

Given an array A[] and a number x, check for pair in A[] with sum as x

Write a C program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

METHOD 1 (Use Sorting)

Algorithm:

```
hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
   (a) Initialize first to the leftmost index: l = 0
   (b) Initialize second the rightmost index: r = ar_size-1
3) Loop while l < r.
   (a) If (A[l] + A[r] == sum) then return 1
   (b) Else if( A[l] + A[r] < sum ) then l++
   (c) Else r--
4) No candidates in whole array - return 0
```

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then $O(n \log n)$ in worst case. If we use Quick Sort then $O(n^2)$ in worst case.

Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is $O(n)$ for merge sort and $O(1)$ for Heap Sort.

Example:

Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16

Sort the array

A = {-8, 1, 4, 6, 10, 45}

Initialize l=0, r=5

A[l] + A[r] (-8 + 45) > 16 => decrement r. Now r = 10

A[l] + A[r] (-8 + 10) < 2 => increment l. Now l = 1

A[l] + A[r] (1 + 10) < 16 => increment l. Now l = 2

A[l] + A[r] (4 + 10) < 14 => increment l. Now l = 3

A[l] + A[r] (6 + 10) == 16 => Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

Implementation:

C

```
# include <stdio.h>
# define bool int

void quickSort(int *, int, int);

bool hasArrayTwoCandidates(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

    /* Now look for the two candidates in the sorted
       array*/
    l = 0;
    r = arr_size-1;
    while (l < r)
    {
        if(A[l] + A[r] == sum)
            return 1;
        else if(A[l] + A[r] < sum)
            l++;
        else // A[i] + A[j] > sum
            r--;
    }
    return 0;
}

/* Driver program to test above function */
int main()
{
```

```

int A[] = {1, 4, 45, 6, 10, -8};
int n = 16;
int arr_size = 6;

if( hasArrayTwoCandidates(A, arr_size, n))
    printf("Array has two elements with sum 16");
else
    printf("Array doesn't have two elements with sum 16 ");

getchar();
return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
PURPOSE */

void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a    = *b;
    *b    = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

```

Python

```

# Python program to check for the sum condition to be satisfied
def hasArrayTwoCandidates(A,arr_size,sum):

    # sort the array
    quickSort(A,0,arr_size-1)
    l = 0
    r = arr_size-1

    # traverse the array for the two elements
    while l < r:
        if (A[l] + A[r] == sum):
            return 1
        elif (A[l] + A[r] < sum):
            l += 1
        else:
            r -= 1
    return 0

# Implementation of Quick Sort
# A[] --> Array to be sorted
# si --> Starting index

```

```

# ei --> Ending index
def quickSort(A, si, ei):
    if si < ei:
        pi=partition(A,si,ei)
        quickSort(A,si,pi-1)
        quickSort(A,pi+1,ei)

# Utility function for partitioning the array(used in quick sort)
def partition(A, si, ei):
    x = A[ei]
    i = (si-1)
    for j in range(si,ei):
        if A[j] <= x:
            i += 1

        # This operation is used to swap two variables in python
        A[i], A[j] = A[j], A[i]

    A[i+1], A[ei] = A[ei], A[i+1]

    return i+1

# Driver program to test the functions
A = [1,4,45,6,10,-8]
n = 16
if (hasArrayTwoCandidates(A, len(A), n)):
    print("Array has two elements with the given sum")
else:
    print("Array doesn't have two elements with the given sum")

## This code is contributed by __Devesh Agrawal__

```

Array has two elements with the given sum

METHOD 2 (Use Hash Map)

Thanks to Bindu for suggesting this method and thanks to [Shekhu](#) for providing code.

This method works in O(n) time if range of numbers is known.

Let sum be the given sum and A[] be the array in which we need to find pair.

- 1) Initialize Binary Hash Map M[] = {0, 0, }
- 2) Do following for each element A[i] in A[]
 - (a) If M[x - A[i]] is set then print the pair (A[i], x - A[i])
 - (b) Set M[A[i]]

Implementation:

C/C++

```

#include <stdio.h>
#define MAX 100000

void printPairs(int arr[], int arr_size, int sum)
{
    int i, temp;
    bool binMap[MAX] = {0}; /*initialize hash map as 0*/

    for (i = 0; i < arr_size; i++)
    {
        temp = sum - arr[i];
        if (temp >= 0 && binMap[temp] == 1)
            printf("Pair with given sum %d is (%d, %d) \n",
                   sum, arr[i], temp);
        binMap[arr[i]] = 1;
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int n = 16;
    int arr_size = sizeof(A)/sizeof(A[0]);
}

```

```

printPairs(A, arr_size, n);

getchar();
return 0;
}

```

Java

```

// Java implementation using Hashing
import java.io.*;

class PairSum
{
    private static final int MAX = 100000; // Max size of Hashmap

    static void printpairs(int arr[],int sum)
    {
        //Declares and initializes the whole array as false
        boolean[] binmap = new boolean[MAX];

        for (int i=0; i<arr.length; ++i)
        {
            int temp = sum-arr[i];

            //checking for condition
            if (temp>=0 && binmap[temp])
            {
                System.out.println("Pair with given sum " +
                    sum + " is (" + arr[i] +
                    ", " +temp+ ")");
            }
            binmap[arr[i]] = true;
        }
    }

    // Main to test the above function
    public static void main (String[] args)
    {
        int A[] = {1, 4, 45, 6, 10, 8};
        int n = 16;
        printpairs(A, n);
    }
}

// This article is contributed by Aakash Hasija

```

Python

```

# Python program to find if there are two elements wtih given sum
CONST_MAX = 100000

# function to check for the given sum in the array
def printPairs(arr, arr_size, sum):

    # initialize hash map as 0
    binmap = [0]*CONST_MAX

    for i in range(0,arr_size):
        temp = sum-arr[i]
        if (temp>=0 and binmap[temp]==1):
            print "Pair with the given sum is", arr[i], "and", temp
            binmap[arr[i]]=1

    # driver program to check the above function
A = [1,4,45,6,10,-8]
n = 16
printPairs(A, len(A), n)

# This code is contributed by __Devesh Agrawal__

```

Pair with given sum 16 is (10, 6)

Auxiliary Space: O(R) where R is range of integers.

If range of numbers include negative numbers then also it works. All we have to do for negative numbers is to make everything positive by adding

the absolute value of smallest negative integer to all numbers.

Majority Element

Majority Element: A majority element in an array A[] of size n is an element that appears more than $n/2$ times (and hence there is at most one such element).

Write a function which takes an array and emits the majority element (if it exists), otherwise prints NONE as follows:

I/P : 3 3 4 2 4 4 2 4 4
O/P : 4

I/P : 3 3 4 2 4 4 2 4
O/P : NONE

METHOD 1 (Basic)

The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than $n/2$ then break the loops and return the element having maximum count. If maximum count doesn't become more than $n/2$ then majority element doesn't exist.

Time Complexity: $O(n^2)$.

Auxiliary Space : $O(1)$.

METHOD 2 (Using Binary Search Tree)

Thanks to [Sachin Midha](#) for suggesting this solution.

Node of the Binary Search Tree (used in this approach) will be as follows.

```
struct tree
{
    int element;
    int count;
}BST;
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than $n/2$ then return.

The method works well for the cases where $n/2+1$ occurrences of the majority element is present in the starting of the array, for example {1, 1, 1, 1, 2, 3, 4}.

Time Complexity: If a binary search tree is used then time complexity will be $O(n^2)$. If a [self-balancing-binary-search](#) tree is used then $O(n \log n)$

Auxiliary Space: $O(n)$

METHOD 3 (Using Moores Voting Algorithm)

This is a two step process.

1. Get an element occurring most of the time in the array. This phase will make sure that if there is a majority element then it will return that only.
2. Check if the element obtained from above step is majority element.

1. Finding a Candidate:

The algorithm for first phase that works in $O(n)$ is known as Moores Voting Algorithm. Basic idea of the algorithm is if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element.

```
findCandidate(a[], size)
1. Initialize index and count of majority element
   maj_index = 0, count = 1
2. Loop for i = 1 to size - 1
   (a) If a[maj_index] == a[i]
       count++
   (b) Else
       count--;
   (c) If count == 0
       maj_index = i;
       count = 1
3. Return a[maj_index]
```

Above algorithm loops through each element and maintains a count of $a[maj_index]$. If next element is same then increments the count, if next element is not same then decrements the count, and if the count reaches 0 then changes the maj_index to the current element and sets count to 1. First Phase algorithm gives us a candidate element. In second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in $O(n)$. We just need to check if count of the candidate element is greater than $n/2$.

Example:

A[] = 2, 2, 3, 5, 2, 2, 6

Initialize:

maj_index = 0, count = 1 > candidate 2?

2, 2, 3, 5, 2, 2, 6

Same as a[maj_index] => count = 2

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 1

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 0

Since count = 0, change candidate for majority element to 5 => maj_index = 3, count = 1

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 0

Since count = 0, change candidate for majority element to 2 => maj_index = 4

2, 2, 3, 5, 2, 2, 6

Same as a[maj_index] => count = 2

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 1

Finally candidate for majority element is 2.

First step uses Moores Voting Algorithm to get a candidate for majority element.

2. Check if the element obtained in step 1 is majority

```
printMajority (a[], size)
1. Find the candidate for majority
2. If candidate is majority. i.e., appears more than n/2 times.
   Print the candidate
3. Else
   Print "NONE"
```

Implementation of method 3:

```
/* Program for finding out majority element in an array */
# include<stdio.h>
# define bool int

int findCandidate(int *, int);
bool isMajority(int *, int, int);

/* Function to print Majority Element */
void printMajority(int a[], int size)
{
    /* Find the candidate for Majority*/
    int cand = findCandidate(a, size);

    /* Print the candidate if it is Majority*/
    if(isMajority(a, size, cand))
        printf(" %d ", cand);
    else
        printf("NO Majority Element");
}

/* Function to find the candidate for Majority */
int findCandidate(int a[], int size)
{
    int maj_index = 0, count = 1;
    int i;
    for(i = 1; i < size; i++)
    {
        if(a[maj_index] == a[i])
            count++;
        else
            count--;
        if(count == 0)
        {
            maj_index = i;
            count = 1;
        }
    }
}
```

```

    }
    return a[maj_index];
}

/* Function to check if the candidate occurs more than n/2 times */
bool isMajority(int a[], int size, int cand)
{
    int i, count = 0;
    for (i = 0; i < size; i++)
        if(a[i] == cand)
            count++;
    if (count > size/2)
        return 1;
    else
        return 0;
}

/* Driver function to test above functions */
int main()
{
    int a[] = {1, 3, 3, 1, 2};
    printMajority(a, 5);
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Auxiliary Space : O(1)

Now give a try to below question

Given an array of $2n$ elements of which n elements are same and the remaining n elements are all different. Write a C program to find out the value which is present n times in the array. There is no restriction on the elements in the array. They are random (In particular they not sequential).

Find the Number Occurring Odd Number of Times

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time & constant space.

Example:

I/P = [1, 2, 3, 2, 3, 1, 3]
O/P = 3

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and inner loop counts number of occurrences of the element picked by outer loop. Time complexity of this solution is $O(n^2)$.

A **Better Solution** is to use Hashing. Use array elements as key and their counts as value. Create an empty hash table. One by one traverse the given array elements and store counts. Time complexity of this solution is $O(n)$. But it requires extra space for hashing.

The **Best Solution** is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring element. Please note that XOR of two elements is 0 if both elements are same and XOR of a number x with 0 is x.

Below are implementations of this best approach.

Program:

C/C++

```
#include <stdio.h>

int getOddOccurrence(int ar[], int ar_size)
{
    int i;
    int res = 0;
    for (i=0; i < ar_size; i++)
        res = res ^ ar[i];

    return res;
}

/* Diver function to test above function */
int main()
{
    int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
    int n = sizeof(ar)/sizeof(ar[0]);
    printf("%d", getOddOccurrence(ar, n));
    return 0;
}
```

Python

```
# Python program to find the element occurring odd number of times

def getOddOccurrence(arr):
    # Initialize result
    res = 0

    # Traverse the array
    for element in arr:
        # XOR with the result
        res = res ^ element

    return res

# Test array
arr = [2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2]

print "%d" % getOddOccurrence(arr)
```

Output:

Largest Sum Contiguous Subarray

Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

Kadanes Algorithm:

```
Initialize:  
    max_so_far = 0  
    max_ending_here = 0  
  
Loop for each element of the array  
(a) max_ending_here = max_ending_here + a[i]  
(b) if(max_ending_here < 0)  
        max_ending_here = 0  
(c) if(max_so_far < max_ending_here)  
        max_so_far = max_ending_here  
return max_so_far
```

Explanation:

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

Lets take the example:

```
{-2, -3, 4, -1, -2, 1, 5, -3}
```

```
max_so_far = max_ending_here = 0
```

```
for i=0, a[0] = -2  
max_ending_here = max_ending_here + (-2)  
Set max_ending_here = 0 because max_ending_here < 0
```

```
for i=1, a[1] = -3  
max_ending_here = max_ending_here + (-3)  
Set max_ending_here = 0 because max_ending_here < 0
```

```
for i=2, a[2] = 4  
max_ending_here = max_ending_here + (4)  
max_ending_here = 4  
max_so_far is updated to 4 because max_ending_here greater  
than max_so_far which was 0 till now
```

```
for i=3, a[3] = -1  
max_ending_here = max_ending_here + (-1)  
max_ending_here = 3
```

```
for i=4, a[4] = -2  
max_ending_here = max_ending_here + (-2)  
max_ending_here = 1
```

```
for i=5, a[5] = 1  
max_ending_here = max_ending_here + (1)  
max_ending_here = 2
```

```
for i=6, a[6] = 5  
max_ending_here = max_ending_here + (5)  
max_ending_here = 7  
max_so_far is updated to 7 because max_ending_here is  
greater than max_so_far
```

```
for i=7, a[7] = -3  
max_ending_here = max_ending_here + (-3)  
max_ending_here = 4
```

Program:

C++

```
// C++ program to print largest contiguous array sum  
#include<iostream>  
using namespace std;  
  
int maxSubArraySum(int a[], int size)  
{  
    int max_so_far = 0, max_ending_here = 0;  
  
    for (int i = 0; i < size; i++)  
    {
```

```

        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is \n" << max_sum;
    return 0;
}

```

Python

```

# Python program to find maximum contiguous subarray

# Function to find the maximum contiguous subarray
def maxSubArraySum(a,size):

    max_so_far = 0
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0

        if (max_so_far < max_ending_here):
            max_so_far = max_ending_here

    return max_so_far

# Driver function to check the above function
a = [-2, -3, 4, -1, -2, 1, 5, -3]
print("Maximum contiguous sum is", maxSubArraySum(a,len(a)))

#This code is contributed by _Devesh Agrawal_

```

Maximum contiguous sum is 7

Notes:

Algorithm doesn't work for all negative numbers. It simply returns 0 if all numbers are negative. For handling this we can add an extra phase before actual implementation. The phase will look if all numbers are negative, if they are it will return maximum of them (or smallest in terms of absolute value). There may be other ways to handle it though.

Above program can be optimized further, if we compare max_so_far with max_ending_here only if max_ending_here is greater than 0.

C++

```

int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;
    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;

        /* Do not compare for all elements. Compare only
           when max_ending_here > 0 */
        else if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

```

Python

```
def maxSubArraySum(a,size):  
    max_so_far = 0  
    max_ending_here = 0  
  
    for i in range(0, size):  
        max_ending_here = max_ending_here + a[i]  
        if max_ending_here < 0:  
            max_ending_here = 0  
  
        # Do not compare for all elements. Compare only  
        # when max_ending_here > 0  
        elif (max_so_far < max_ending_here):  
            max_so_far = max_ending_here  
  
    return max_so_far
```

Time Complexity: O(n)

Algorithmic Paradigm: Dynamic Programming

Following is another simple implementation suggested by **Mohit Kumar**. The implementation handles the case when all numbers in array are negative.

Find the Missing Number

You are given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Example:

I/P [1, 2, 4, ,6, 3, 7, 8]
O/P 5

METHOD 1(Use sum formula)

Algorithm:

1. Get the sum of numbers
total = $n*(n+1)/2$
- 2 Subtract all the numbers from sum and
you will get the missing number.

Program:

```
#include<stdio.h>

/* getMissingNo takes array and size of array as arguments*/
int getMissingNo (int a[], int n)
{
    int i, total;
    total = (n+1)*(n+2)/2;
    for ( i = 0; i < n; i++)
        total -= a[i];
    return total;
}

/*program to test above function */
int main()
{
    int a[] = {1,2,4,5,6};
    int miss = getMissingNo(a,5);
    printf("%d", miss);
    getchar();
}
```

Time Complexity: O(n)

METHOD 2(Use XOR)

- 1) XOR all the array elements, let the result of XOR be X1.
- 2) XOR all numbers from 1 to n, let XOR be X2.
- 3) XOR of X1 and X2 gives the missing number.

```
#include<stdio.h>

/* getMissingNo takes array and size of array as arguments*/
int getMissingNo(int a[], int n)
{
    int i;
    int x1 = a[0]; /* For xor of all the elements in arary */
    int x2 = 1; /* For xor of all the elements from 1 to n+1 */

    for ( i = 1; i < n; i++)
        x1 = x1^a[i];

    for ( i = 2; i <= n+1; i++)
        x2 = x2^i;

    return (x1^x2);
}

/*program to test above function */
int main()
{
    int a[] = {1, 2, 4, 5, 6};
    int miss = getMissingNo(a, 5);
    printf("%d", miss);
    getchar();
}
```

Time Complexity: O(n)

In method 1, if the sum of the numbers goes beyond maximum allowed integer, then there can be integer overflow and we may not get correct answer. Method 2 has no such problems.

Search an element in a sorted and rotated array

An element in a sorted array can be found in $O(\log n)$ time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in $O(\log n)$ time.

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 5 | 1 | 2 |
|---|---|---|---|---|

All solutions provided here assume that all elements in array are distinct.

The idea is to find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.

Using above criteria and binary search methodology we can get pivot element in $O(\log n)$ time

```
Input arr[] = {3, 4, 5, 1, 2}
Element to Search = 1
1) Find out pivot point and divide the array in two
   sub-arrays. (pivot = 2) /*Index of 5*/
2) Now call binary search for one of the two sub-arrays.
   (a) If element is greater than 0th element then
       search in left array
   (b) Else Search in right array
       (1 will go in else as 1 < 0th element(3))
3) If element is found in selected sub-array then return index
   Else return -1.
```

Implementation:

```
/* Program to search an element in a sorted and pivoted array*/
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element key in a pivoted sorted array arrp[]
   of size n */
int pivotedBinarySearch(int arr[], int n, int key)
{
    int pivot = findPivot(arr, 0, n-1);

    // If we didn't find a pivot, then array is not rotated at all
    if (pivot == -1)
        return binarySearch(arr, 0, n-1, key);

    // If we found a pivot, then first compare with pivot and then
    // search in two subarrays around pivot
    if (arr[pivot] == key)
        return pivot;
    if (arr[0] <= key)
        return binarySearch(arr, 0, pivot-1, key);
    return binarySearch(arr, pivot+1, n-1, key);
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it returns
   3 (index of 6) */
int findPivot(int arr[], int low, int high)
{
    // base cases
    if (high < low)  return -1;
    if (high == low) return low;

    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (mid < high && arr[mid] > arr[mid + 1])
        return mid;
    if (mid > low && arr[mid] < arr[mid - 1])
        return (mid-1);
    if (arr[low] >= arr[mid])
        return findPivot(arr, low, mid-1);
    return findPivot(arr, mid + 1, high);
}
```

```

/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int key)
{
    if (high < low)
        return -1;
    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (key == arr[mid])
        return mid;
    if (key > arr[mid])
        return binarySearch(arr, (mid + 1), high, key);
    return binarySearch(arr, low, (mid -1), key);
}

/* Driver program to check above functions */
int main()
{
    // Let us search 3 in below array
    int arr1[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    int key = 3;
    printf("Index: %d\n", pivotedBinarySearch(arr1, n, key));
    return 0;
}

```

Output:

Index of the element is 8

Time Complexity O(logn). Thanks to Ajay Mishra for initial solution.

Improved Solution:

We can search an element in one pass of Binary Search. The idea is to search

- 1) Find middle point mid = (l + h)/2
- 2) **If** key is present at middle point, return mid.
- 3) **Else If** arr[l..mid] is sorted
 - a) **If** key to be searched lies in range from arr[l] to arr[mid], recur for arr[l..mid].
 - b) **Else** recur for arr[mid+1..r]
- 4) **Else** (arr[mid+1..r] must be sorted)
 - a) **If** key to be searched lies in range from arr[mid+1] to arr[r], recur for arr[mid+1..r].
 - b) **Else** recur for arr[l..mid]

Below is C++ implementation of above idea.

```

// Search an element in sorted and rotated array using
// single pass of Binary Search
#include<bits/stdc++.h>
using namespace std;

// Returns index of key in arr[l..h] if key is present,
// otherwise returns -1
int search(int arr[], int l, int h, int key)
{
    if (l > h) return -1;

    int mid = (l+h)/2;
    if (arr[mid] == key) return mid;

    /* If arr[l...mid] is sorted */
    if (arr[l] <= arr[mid])
    {
        /* As this subarray is sorted, we can quickly
         * check if key lies in half or other half */
        if (key >= arr[l] && key <= arr[mid])
            return search(arr, l, mid-1, key);

        return search(arr, mid+1, h, key);
    }

    /* If arr[l..mid] is not sorted, then arr[mid... r]
     * must be sorted*/
    if (key >= arr[mid] && key <= arr[h])
        return search(arr, mid+1, h, key);

    return search(arr, l, mid-1, key);
}

```

```
// Driver program
int main()
{
    int arr[] = {4, 5, 6, 7, 8, 9, 1, 2, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 6;
    int i = search(arr, 0, n-1, key);
    if (i != -1) cout << "Index: " << i << endl;
    else cout << "Key not found\n";
}
```

Output:

Index: 2

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

How to handle duplicates?

It doesn't look possible to search in $O(\log n)$ time in all cases when duplicates are allowed. For example consider searching 0 in {2, 2, 2, 2, 2, 2, 2, 0, 2} and {2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to decide whether to recur for left half or right half by doing constant number of comparisons at the middle.

Similar Articles:

[Find the minimum element in a sorted and rotated array](#)

[Given a sorted and rotated array, find if there is a pair with a given sum](#)

Merge an array of size n into another array of size m+n

Asked by Binod

Question:

There are two sorted arrays. First one is of size m+n containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size m+n such that the output is sorted.

Input: array with m+n elements (mPlusN[]).

| | | | | | | |
|---|----|---|----|----|----|----|
| 2 | NA | 7 | NA | NA | 10 | NA |
|---|----|---|----|----|----|----|

NA => Value is not filled/available in array mPlusN[]. There should be n such array blocks.

Input: array with n elements (N[]).

| | | | |
|---|---|----|----|
| 5 | 8 | 12 | 14 |
|---|---|----|----|

Output: N[] merged into mPlusN[] (Modified mPlusN[])

| | | | | | | |
|---|---|---|---|----|----|----|
| 2 | 5 | 7 | 8 | 10 | 12 | 14 |
|---|---|---|---|----|----|----|

Algorithm:

Let first array be mPlusN[] and other array be N[]

- 1) Move m elements of mPlusN[] to end.
- 2) Start from nth element of mPlusN[] and 0th element of N[] and merge them into mPlusN[].

Implementation:

```
#include <stdio.h>

/* Assuming -1 is filled for the places where element
   is not available */
#define NA -1

/* Function to move m elements at the end of array mPlusN[] */
void moveToEnd(int mPlusN[], int size)
{
    int i = 0, j = size - 1;
    for (i = size-1; i >= 0; i--)
        if (mPlusN[i] != NA)
    {
        mPlusN[j] = mPlusN[i];
        j--;
    }
}

/* Merges array N[] of size n into array mPlusN[]
   of size m+n*/
int merge(int mPlusN[], int N[], int m, int n)
{
    int i = n; /* Current index of i/p part of mPlusN[]*/
    int j = 0; /* Current index of N[]*/
    int k = 0; /* Current index of of output mPlusN[]*/
    while (k < (m+n))
    {
        /* Take an element from mPlusN[] if
           a) value of the picked element is smaller and we have
              not reached end of it
           b) We have reached end of N[] */
        if ((i < (m+n) && mPlusN[i] <= N[j]) || (j == n))
        {
            mPlusN[k] = mPlusN[i];
            k++;
            i++;
        }
        else // Otherwise take element from N[]
        {
            mPlusN[k] = N[j];
            k++;
            j++;
        }
    }
}
```

```

}

/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    /* Initialize arrays */
    int mPlusN[] = {2, 8, NA, NA, NA, 13, NA, 15, 20};
    int N[] = {5, 7, 9, 25};
    int n = sizeof(N)/sizeof(N[0]);
    int m = sizeof(mPlusN)/sizeof(mPlusN[0]) - n;

    /*Move the m elements at the end of mPlusN*/
    moveToEnd(mPlusN, m+n);

    /*Merge N[] into mPlusN[] */
    merge(mPlusN, N, m, n);

    /* Print the resultant mPlusN */
    printArray(mPlusN, m+n);

    return 0;
}

```

Output:

2 5 7 8 9 13 15 20 25

Time Complexity: O(m+n)

Please write comment if you find any bug in the above program or a better way to solve the same problem.

Median of two sorted arrays

Question: There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays(i.e. array of length 2n). The complexity should be O(log(n))

Median: In probability theory and statistics, a median is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half.

The median of a finite list of numbers can be found by arranging all the numbers from lowest value to highest value and picking the middle one.

For getting the median of input array { 12, 11, 15, 10, 20 }, first sort the array. We get { 10, 11, 12, 15, 20 } after sorting. Median is the middle element of the sorted array which is 12.

There are different conventions to take median of an array with even number of elements, one can take the mean of the two middle values, or first middle value, or second middle value.

Let us see different methods to get the median of two sorted arrays of size n each. Since size of the set for which we are looking for median is even (2n), we are taking average of middle two numbers in all below solutions.

Method 1 (Simply count while Merging)

Use merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n(For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array. See the below implementation.

Implementation:

```
#include <stdio.h>

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int i = 0; /* Current index of i/p array ar1[] */
    int j = 0; /* Current index of i/p array ar2[] */
    int count;
    int m1 = -1, m2 = -1;

    /* Since there are 2n elements, median will be average
    of elements at index n-1 and n in the array obtained after
    merging ar1 and ar2 */
    for (count = 0; count <= n; count++)
    {
        /*Below is to handle case where all elements of ar1[] are
        smaller than smallest(or first) element of ar2[]*/
        if (i == n)
        {
            m1 = m2;
            m2 = ar2[0];
            break;
        }

        /*Below is to handle case where all elements of ar2[] are
        smaller than smallest(or first) element of ar1[]*/
        else if (j == n)
        {
            m1 = m2;
            m2 = ar1[0];
            break;
        }

        if (ar1[i] < ar2[j])
        {
            m1 = m2; /* Store the prev median */
            m2 = ar1[i];
            i++;
        }
        else
        {
            m1 = m2; /* Store the prev median */
            m2 = ar2[j];
            j++;
        }
    }

    return (m1 + m2)/2;
}
```

```

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 12, 15, 26, 38};
    int ar2[] = {2, 13, 17, 30, 45};

    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Method 2 (By comparing the medians of two arrays)

This method works by first getting medians of the two sorted arrays and then comparing them.

Let ar1 and ar2 be the input arrays.

Algorithm:

- 1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.
- 2) If m1 and m2 both are equal then we are done.
 return m1 (or m2)
- 3) If m1 is greater than m2, then median is present in one of the below two subarrays.
 - a) From first element of ar1 to m1 (ar1[0...|_n/2_|])
 - b) From m2 to last element of ar2 (ar2[|_n/2_|...n-1])
- 4) If m2 is greater than m1, then median is present in one of the below two subarrays.
 - a) From m1 to last element of ar1 (ar1[|_n/2_|...n-1])
 - b) From first element of ar2 to m2 (ar2[0...|_n/2_|])
- 5) Repeat the above process until size of both the subarrays becomes 2.
- 6) If size of the two arrays is 2 then use below formula to get the median.

$$\text{Median} = (\max(\text{ar1}[0], \text{ar2}[0]) + \min(\text{ar1}[1], \text{ar2}[1]))/2$$

Example:

```

ar1[] = {1, 12, 15, 26, 38}
ar2[] = {2, 13, 17, 30, 45}

```

For above two arrays m1 = 15 and m2 = 17

For the above ar1[] and ar2[], m1 is smaller than m2. So median is present in one of the following two subarrays.

[15, 26, 38] and [2, 13, 17]

Let us repeat the process for above two subarrays:

m1 = 26 m2 = 13.

m1 is greater than m2. So the subarrays become

```

[15, 26] and [13, 17]
Now size is 2, so median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2
= (max(15, 13) + min(26, 17))/2
= (15 + 17)/2
= 16

```

Implementation:

```

#include<stdio.h>

int max(int, int); /* to get maximum of two integers */
int min(int, int); /* to get minimum of two integers */
int median(int [], int); /* to get median of a sorted array */

```

```

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int m1; /* For median of ar1 */
    int m2; /* For median of ar2 */

    /* return -1 for invalid input */
    if (n <= 0)
        return -1;

    if (n == 1)
        return (ar1[0] + ar2[0])/2;

    if (n == 2)
        return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;

    m1 = median(ar1, n); /* get the median of the first array */
    m2 = median(ar2, n); /* get the median of the second array */

    /* If medians are equal then return either m1 or m2 */
    if (m1 == m2)
        return m1;

    /* if m1 < m2 then median must exist in ar1[m1....] and ar2[....m2] */
    if (m1 < m2)
    {
        if (n % 2 == 0)
            return getMedian(ar1 + n/2 - 1, ar2, n - n/2 +1);
        else
            return getMedian(ar1 + n/2, ar2, n - n/2);
    }

    /* if m1 > m2 then median must exist in ar1[....m1] and ar2[m2...] */
    else
    {
        if (n % 2 == 0)
            return getMedian(ar2 + n/2 - 1, ar1, n - n/2 + 1);
        else
            return getMedian(ar2 + n/2, ar1, n - n/2);
    }
}

/* Function to get median of a sorted array */
int median(int arr[], int n)
{
    if (n%2 == 0)
        return (arr[n/2] + arr[n/2-1])/2;
    else
        return arr[n/2];
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 2, 3, 6};
    int ar2[] = {4, 6, 8, 10};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");

    getchar();
    return 0;
}

/* Utility functions */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x > y? y : x;
}

```

Time Complexity: O(logn)

Algorithmic Paradigm: Divide and Conquer

Method 3 (By doing binary search for the median):

The basic idea is that if you are given two arrays ar1[] and ar2[] and know the length of each, you can check whether an element ar1[i] is the median in constant time. Suppose that the median is ar1[i]. Since the array is sorted, it is greater than exactly i values in array ar1[]. Then if it is the median, it is also greater than exactly j = n - i - 1 elements in ar2[].

It requires constant time to check if ar2[j] <= ar1[i] <= ar2[j + 1]. If ar1[i] is not the median, then depending on whether ar1[i] is greater or less than ar2[j] and ar2[j + 1], you know that ar1[i] is either greater than or less than the median. Thus you can binary search for median in O(lg n) worst-case time. For two arrays ar1 and ar2, first do binary search in ar1[]. If you reach at the end (left or right) of the first array and don't find median, start searching in the second array ar2[].

- 1) Get the middle element of ar1[] using array indexes left and right.
Let index of the middle element be i.
- 2) Calculate the corresponding index j of ar2[]
 $j = n - i - 1$
- 3) If $ar1[i] \geq ar2[j]$ and $ar1[i] \leq ar2[j+1]$ then $ar1[i]$ and $ar2[j]$ are the middle elements.
return average of $ar2[j]$ and $ar1[i]$
- 4) If $ar1[i]$ is greater than both $ar2[j]$ and $ar2[j+1]$ then
do binary search in left half (i.e., arr[left ... i-1])
- 5) If $ar1[i]$ is smaller than both $ar2[j]$ and $ar2[j+1]$ then
do binary search in right half (i.e., arr[i+1...right])
- 6) If you reach at any corner of ar1[] then do binary search in ar2[]

Example:

```
ar1[] = {1, 5, 7, 10, 13}
ar2[] = {11, 15, 23, 30, 45}
```

Middle element of ar1[] is 7. Let us compare 7 with 23 and 30, since 7 is smaller than both 23 and 30, move to right in ar1[]. Do binary search in {10, 13}, this step will pick 10. Now compare 10 with 15 and 23. Since 10 is smaller than both 15 and 23, again move to right. Only 13 is there in right side now. Since 13 is greater than 11 and smaller than 15, terminate here. We have got the median as 12 (average of 11 and 13)

Implementation:

```
#include<stdio.h>

int getMedianRec(int ar1[], int ar2[], int left, int right, int n);

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    return getMedianRec(ar1, ar2, 0, n-1, n);
}

/* A recursive function to get the median of ar1[] and ar2[]
using binary search */
int getMedianRec(int ar1[], int ar2[], int left, int right, int n)
{
    int i, j;

    /* We have reached at the end (left or right) of ar1[] */
    if (left > right)
        return getMedianRec(ar2, ar1, 0, n-1, n);

    i = (left + right)/2;
    j = n - i - 1; /* Index of ar2[] */

    /* Recursion terminates here.*/
    if (ar1[i] > ar2[j] && (j == n-1 || ar1[i] <= ar2[j+1]))
    {
        /* ar1[i] is decided as median 2, now select the median 1
        (element just before ar1[i] in merged array) to get the
        average of both*/
        if (i == 0 || ar2[j] > ar1[i-1])
            return (ar1[i] + ar2[j])/2;
        else
            return (ar1[i] + ar1[i-1])/2;
    }
    /*Search in left half of ar1[]*/
```

```

else if (ar1[i] > ar2[j] && j != n-1 && ar1[i] > ar2[j+1])
    return getMedianRec(ar1, ar2, left, i-1, n);

/*Search in right half of ar1[]*/
else /* ar1[i] is smaller than both ar2[j] and ar2[j+1]*/
    return getMedianRec(ar1, ar2, i+1, right, n);
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 12, 15, 26, 38};
    int ar2[] = {2, 13, 17, 30, 45};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");

    getchar();
    return 0;
}

```

Time Complexity: O(logn)

Algorithmic Paradigm: Divide and Conquer

The above solutions can be optimized for the cases when all elements of one array are smaller than all elements of other array. For example, in method 3, we can change the getMedian() function to following so that these cases can be handled in O(1) time. Thanks to [nutcracker](#) for suggesting this optimization.

```

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    // If all elements of array 1 are smaller then
    // median is average of last element of ar1 and
    // first element of ar2
    if (ar1[n-1] < ar2[0])
        return (ar1[n-1]+ar2[0])/2;

    // If all elements of array 1 are smaller then
    // median is average of first element of ar1 and
    // last element of ar2
    if (ar2[n-1] < ar1[0])
        return (ar2[n-1]+ar1[0])/2;

    return getMedianRec(ar1, ar2, 0, n-1, n);
}

```

References:

<http://en.wikipedia.org/wiki/Median>

<http://ocw.alfaisal.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/30C68118-E436-4FE3-8C79-6BAFBB07D935/0/ps9sol.pdf>

Asked by Snehal

Write a program to reverse an array or string

Iterative way:

- 1) Initialize start and end indexes.
start = 0, end = n-1
- 2) In a loop, swap arr[start] with arr[end] and change start and end as follows.
start = start +1; end = end - 1

C

```
// Iterative C program to reverse an array
#include<stdio.h>

/* Function to reverse arr[] from start to end*/
void rvereseArray(int arr[], int start, int end)
{
    int temp;
    while (start < end)
    {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    printArray(arr, 6);
    rvereseArray(arr, 0, 5);
    printf("Reversed array is \n");
    printArray(arr, 6);
    return 0;
}
```

Java

```
// Java program to reverse an array
import java.io.*;

class ReverseArray {

    /* Function to reverse arr[] from start to end*/
    static void rvereseArray(int arr[], int start, int end)
    {
        int temp;
        if (start >= end)
            return;
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        rvereseArray(arr, start+1, end-1);
    }

    /* Utility that prints out an array on a line */
    static void printArray(int arr[], int size)
    {
        int i;
        for (i=0; i < size; i++)
            System.out.print(arr[i] + " ");
        System.out.println("");
    }

    /*Driver function to check for above functions*/
}
```

```

public static void main (String[] args) {
    int arr[] = {1, 2, 3, 4, 5, 6};
    printArray(arr, 6);
    rvereseArray(arr, 0, 5);
    System.out.println("Reversed array is ");
    printArray(arr, 6);
}
/*This code is contributed by Devesh Agrawal*/

```

1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1

Time Complexity: O(n)

Recursive Way:

- 1) Initialize start and end indexes
start = 0, end = n-1
- 2) Swap arr[start] with arr[end]
- 3) Recursively call reverse for rest of the array.

C

```

// Recursive C program to reverse an array
#include <stdio.h>

/* Function to reverse arr[] from start to end*/
void rvereseArray(int arr[], int start, int end)
{
    int temp;
    if (start >= end)
        return;
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    rvereseArray(arr, start+1, end-1);
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    printArray(arr, 5);
    rvereseArray(arr, 0, 4);
    printf("Reversed array is \n");
    printArray(arr, 5);
    return 0;
}

```

Java

```

// Recursive Java Program to reverse an array
import java.io.*;

class ReverseArray {

    /* Function to reverse arr[] from start to end*/
    static void rvereseArray(int arr[], int start, int end)
    {
        int temp;
        if (start >= end)
            return;

```

```

temp = arr[start];
arr[start] = arr[end];
arr[end] = temp;
rvereseArray(arr, start+1, end-1);
}

/* Utility that prints out an array on a line */
static void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        System.out.print(arr[i] + " ");
    System.out.println("");
}

/*Driver function to check for above functions*/
public static void main (String[] args) {
    int arr[] = {1, 2, 3, 4, 5, 6};
    printArray(arr, 6);
    rvereseArray(arr, 0, 5);
    System.out.println("Reversed array is ");
    printArray(arr, 6);
}
}

/*This article is contributed by Devesh Agrawal*/

```

1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1

Time Complexity: O(n)

Program for array rotation

Write a function `rotate(ar[], d, n)` that rotates `arr[]` of size `n` by `d` elements.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

Rotation of the above array by 2 will make array

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 7 | 1 | 2 |
|---|---|---|---|---|---|---|

METHOD 1 (Use temp array)

Input `arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n = 7`
1) Store `d` elements in a temp array
 `temp[] = [1, 2]`
2) Shift rest of the `arr[]`
 `arr[] = [3, 4, 5, 6, 7, 6, 7]`
3) Store back the `d` elements
 `arr[] = [3, 4, 5, 6, 7, 1, 2]`

Time complexity $O(n)$

Auxiliary Space: $O(d)$

METHOD 2 (Rotate one by one)

```
leftRotate(arr[], d, n)
start
    For i = 0 to i < d
        Left rotate all elements of arr[] by one
end
```

To rotate by one, store `arr[0]` in a temporary variable `temp`, move `arr[1]` to `arr[0]`, `arr[2]` to `arr[1]` and finally `temp` to `arr[n-1]`

Let us take the same example `arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2`

Rotate `arr[]` by one 2 times

We get `[2, 3, 4, 5, 6, 7, 1]` after first rotation and `[3, 4, 5, 6, 7, 1, 2]` after second rotation.

```
/*Function to left Rotate arr[] of size n by 1*/
void leftRotatebyOne(int arr[], int n);

/*Function to left rotate arr[] of size n by d*/
void leftRotate(int arr[], int d, int n)
{
    int i;
    for (i = 0; i < d; i++)
        leftRotatebyOne(arr, n);
}

void leftRotatebyOne(int arr[], int n)
{
    int i, temp;
    temp = arr[0];
    for (i = 0; i < n-1; i++)
        arr[i] = arr[i+1];
    arr[i] = temp;
}

/* utility function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    leftRotate(arr, 2, 7);
}
```

```

printArray(arr, 7);
getchar();
return 0;
}

```

Time complexity: O(n*d)

Auxiliary Space: O(1)

METHOD 3 (A Juggling Algorithm)

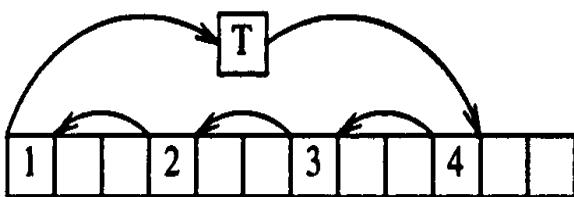
This is an extension of method 2. Instead of moving one by one, divide the array in different sets where number of sets is equal to GCD of n and d and move the elements within sets.

If GCD is 1 as is for the above example array (n = 7 and d = 2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[I+d] to arr[I] and finally store temp at the right place.

Here is an example for n=12 and d = 3. GCD is 3 and

Let arr[] be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

a) Elements are first moved in first set (See below diagram for this movement)



arr[] after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}

b) Then in second set.

arr[] after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}

c) Finally in third set.

arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}

```
/* function to print an array */
void printArray(int arr[], int size);
```

```
/*Function to get gcd of a and b*/
int gcd(int a,int b);
```

```
/*Function to left rotate arr[] of siz n by d*/
void leftRotate(int arr[], int d, int n)
{
```

```
    int i, j, k, temp;
    for (i = 0; i < gcd(d, n); i++)
    {
        /* move i-th values of blocks */
        temp = arr[i];
        j = i;
        while(1)
        {
            k = j + d;
            if (k >= n)
                k = k - n;
            if (k == i)
                break;
            arr[j] = arr[k];
            j = k;
        }
        arr[j] = temp;
    }
}
```

```
/*UTILITY FUNCTIONS*/
```

```
/* function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
}
```

```

/*Function to get gcd of a and b*/
int gcd(int a,int b)
{
    if(b==0)
        return a;
    else
        return gcd(b, a%b);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    leftRotate(arr, 2, 7);
    printArray(arr, 7);
    getchar();
    return 0;
}

```

Time complexity: O(n)

Auxiliary Space: O(1)

Please see following posts for other methods of array rotation:

[Block swap algorithm for array rotation](#)

[Reversal algorithm for array rotation](#)

References:

<http://www.cs.bell-labs.com/cm/cs/pearls/s02b.pdf>

Reversal algorithm for array rotation

Write a function rotate(arr[], d, n) that rotates arr[] of size n by d elements.

Example:

Input: arr[] = [1, 2, 3, 4, 5, 6, 7]
d = 2

Output: arr[] = [3, 4, 5, 6, 7, 1, 2]

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

Rotation of the above array by 2 will make array

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 7 | 1 | 2 |
|---|---|---|---|---|---|---|

Method 4(The Reversal Algorithm)

Please read [this](#) for first three methods of array rotation.

Algorithm:

```
rotate(arr[], d, n)
    reverse(arr[], 1, d) ;
    reverse(arr[], d + 1, n);
    reverse(arr[], 1, n);
```

Let AB are the two parts of the input array where A = arr[0..d-1] and B = arr[d..n-1]. The idea of the algorithm is:

Reverse A to get ArB. /* Ar is reverse of A */

Reverse B to get ArBr. /* Br is reverse of B */

Reverse all to get (ArBr)r = BA.

For arr[] = [1, 2, 3, 4, 5, 6, 7], d=2 and n = 7

A = [1, 2] and B = [3, 4, 5, 6, 7]

Reverse A, we get ArB = [2, 1, 3, 4, 5, 6, 7]

Reverse B, we get ArBr = [2, 1, 7, 6, 5, 4, 3]

Reverse all, we get (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

Implementation:

C/C++

```
// C/C++ program for reversal algorithm of array rotation
#include<stdio.h>

/*Utility function to print an array */
void printArray(int arr[], int size);

/* Utility function to reverse arr[] from start to end */
void rvereseArray(int arr[], int start, int end);

/* Function to left rotate arr[] of size n by d */
void leftRotate(int arr[], int d, int n)
{
    rvereseArray(arr, 0, d-1);
    rvereseArray(arr, d, n-1);
    rvereseArray(arr, 0, n-1);
}

/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

/*Function to reverse arr[] from index start to end*/
```

```

void rvereseArray(int arr[], int start, int end)
{
    int temp;
    while (start < end)
    {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    int d = 2;
    leftRotate(arr, d, n);
    printArray(arr, n);
    return 0;
}

```

Java

```

// Java program for reversal algorithm of array rotation
import java.io.*;

class LeftRotate
{
    /* Function to left rotate arr[] of size n by d */
    static void leftRotate(int arr[], int d)
    {
        int n = arr.length;
        rvereseArray(arr, 0, d-1);
        rvereseArray(arr, d, n-1);
        rvereseArray(arr, 0, n-1);
    }

    /*Function to reverse arr[] from index start to end*/
    static void rvereseArray(int arr[], int start, int end)
    {
        int temp;
        while (start < end)
        {
            temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }

    /*UTILITY FUNCTIONS*/
    /* function to print an array */
    static void printArray(int arr[])
    {
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }

    /* Driver program to test above functions */
    public static void main (String[] args)
    {
        int arr[] = {1, 2, 3, 4, 5, 6, 7};
        leftRotate(arr, 2); // Rotate array by 2
        printArray(arr);
    }
}
/*This code is contributed by Devesh Agrawal*/

```

Python

```

# Python program for reversal algorithm of array rotation

# Function to reverse arr[] from index start to end
def rvereseArray(arr, start, end):
    while (start < end):

```

```

temp = arr[start]
arr[start] = arr[end]
arr[end] = temp
start += 1
end = end-1

# Function to left rotate arr[] of size n by d
def leftRotate(arr, d):
    n = len(arr)
    rverseArray(arr, 0, d-1)
    rverseArray(arr, d, n-1)
    rverseArray(arr, 0, n-1)

# Function to print an array
def printArray(arr):
    for i in range(0, len(arr)):
        print arr[i],

# Driver function to test above functions
arr = [1, 2, 3, 4, 5, 6, 7]
leftRotate(arr, 2) # Rotate array by 2
printArray(arr)

# This code is contributed by Devesh Agrawal

```

3 4 5 6 7 1 2

Time Complexity: O(n)

References:

<http://www.cs.bell-labs.com/cm/cs/pearls/s02b.pdf>

Block swap algorithm for array rotation

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

Rotation of the above array by 2 will make array

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 7 | 1 | 2 |
|---|---|---|---|---|---|---|

Algorithm:

- Initialize A = arr[0..d-1] and B = arr[d..n-1]
- 1) Do following until size of A is equal to size of B
 - a) If A is shorter, divide B into Bl and Br such that Br is of same length as A. Swap A and Br to change AB_lB_r into B_rB_lA. Now A is at its final place, so recur on pieces of B.
 - b) If A is longer, divide A into Al and Ar such that Al is of same length as B Swap Al and B to change A_lA_rB into B_{Ar}A_l. Now B is at its final place, so recur on pieces of A.
 - 2) Finally when A and B are of equal size, block swap them.

Recursive Implementation:

```
#include<stdio.h>

/*Prototype for utility functions */
void printArray(int arr[], int size);
void swap(int arr[], int fi, int si, int d);

void leftRotate(int arr[], int d, int n)
{
    /* Return If number of elements to be rotated is
       zero or equal to array size */
    if(d == 0 || d == n)
        return;

    /*If number of elements to be rotated is exactly
       half of array size */
    if(n-d == d)
    {
        swap(arr, 0, n-d, d);
        return;
    }

    /* If A is shorter*/
    if(d < n-d)
    {
        swap(arr, 0, n-d, d);
        leftRotate(arr, d, n-d);
    }
    else /* If B is shorter*/
    {
        swap(arr, 0, d, n-d);
        leftRotate(arr+n-d, 2*d-n, d); /*This is tricky*/
    }
}

/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/*This function swaps d elements starting at index fi
   with d elements starting at index si */
void swap(int arr[], int fi, int si, int d)
```

```

{
    int i, temp;
    for(i = 0; i<d; i++)
    {
        temp = arr[fi + i];
        arr[fi + i] = arr[si + i];
        arr[si + i] = temp;
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    leftRotate(arr, 2, 7);
    printArray(arr, 7);
    getchar();
    return 0;
}

```

Iterative Implementation:

Here is iterative implementation of the same algorithm. Same utility function swap() is used here.

```

void leftRotate(int arr[], int d, int n)
{
    int i, j;
    if(d == 0 || d == n)
        return;
    i = d;
    j = n - d;
    while (i != j)
    {
        if(i < j) /*A is shorter*/
        {
            swap(arr, d-i, d+j-i, i);
            j -= i;
        }
        else /*B is shorter*/
        {
            swap(arr, d-i, d, j);
            i -= j;
        }
        // printArray(arr, 7);
    }
    /*Finally, block swap A and B*/
    swap(arr, d-i, d, i);
}

```

Time Complexity: O(n)

Please see following posts for other methods of array rotation:

<http://geeksforgeeks.org/?p=2398>

<http://geeksforgeeks.org/?p=2838>

References:

<http://www.cs.bell-labs.com/cm/cs/pearls/s02b.pdf>

Maximum sum such that no two elements are adjacent

Question: Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7). Answer the question in most efficient way.

Algorithm:

Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.

Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).

At the end of the loop return max of incl and excl.

Example:

```
arr[] = {5, 5, 10, 40, 50, 35}

inc = 5
exc = 0

For i = 1 (current element is 5)
incl = (excl + arr[i]) = 5
excl = max(5, 0) = 5

For i = 2 (current element is 10)
incl = (excl + arr[i]) = 15
excl = max(5, 5) = 5

For i = 3 (current element is 40)
incl = (excl + arr[i]) = 45
excl = max(5, 15) = 15

For i = 4 (current element is 50)
incl = (excl + arr[i]) = 65
excl = max(45, 15) = 45

For i = 5 (current element is 35)
incl = (excl + arr[i]) = 80
excl = max(5, 15) = 65
```

And 35 is the last element. So, answer is max(incl, excl) = 80

Thanks to [Debanjan](#) for providing code.

Implementation:

```
#include<stdio.h>

/*Function to return max sum such that no two elements
are adjacent */
int FindMaxSum(int arr[], int n)
{
    int incl = arr[0];
    int excl = 0;
    int excl_new;
    int i;

    for (i = 1; i < n; i++)
    {
        /* current max excluding i */
        excl_new = (incl > excl)? incl: excl;

        /* current max including i */
        incl = excl + arr[i];
        excl = excl_new;
    }

    /* return max of incl and excl */
    return ((incl > excl)? incl : excl);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {5, 5, 10, 100, 10, 5};
```

```
printf("%d \n", FindMaxSum(arr, 6));
getchar();
return 0;
}
```

Time Complexity: O(n)

Now try the same problem for array with negative numbers also.

Leaders in an array

Write a program to print all the LEADERS in the array. An element is leader if it is greater than all the elements to its right side. And the rightmost element is always a leader. For example int the array {16, 17, 4, 3, 5, 2}, leaders are 17, 5 and 2.

Let the input array be `arr[]` and size of the array be `size`.

Method 1 (Simple)

Use two loops. The outer loop runs from 0 to `size - 1` and one by one picks all elements from left to right. The inner loop compares the picked element to all the elements to its right side. If the picked element is greater than all the elements to its right side, then the picked element is the leader.

C++

```
#include<iostream>
using namespace std;

/*C++ Function to print leaders in an array */
void printLeaders(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        int j;
        for (j = i+1; j < size; j++)
        {
            if (arr[i] <= arr[j])
                break;
        }
        if (j == size) // the loop didn't break
            cout << arr[i] << " ";
    }
}

/* Driver program to test above function */
int main()
{
    int arr[] = {16, 17, 4, 3, 5, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printLeaders(arr, n);
    return 0;
}
```

Python

```
# Python Function to print leaders in array

def printLeaders(arr,size):

    for i in range(0, size):
        for j in range(i+1, size):
            if arr[i]<=arr[j]:
                break
        if j == size-1: # If loop didn't break
            print arr[i],

# Driver function
arr=[16, 17, 4, 3, 5, 2]
printLeaders(arr, len(arr))

# This code is contributed by _Devesh Agrawal_
```

17 5 2

Time Complexity: $O(n^2)$

Method 2 (Scan from right)

Scan all the elements from right to left in array and keep track of maximum till now. When maximum changes its value, print it.

C++

```

#include <iostream>
using namespace std;

/* C++ Function to print leaders in an array */
void printLeaders(int arr[], int size)
{
    int max_from_right = arr[size-1];

    /* Rightmost element is always leader */
    cout << max_from_right << " ";

    for (int i = size-2; i >= 0; i--)
    {
        if (max_from_right < arr[i])
        {
            max_from_right = arr[i];
            cout << max_from_right << " ";
        }
    }
}

/* Driver program to test above function*/
int main()
{
    int arr[] = {16, 17, 4, 3, 5, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printLeaders(arr, n);
    return 0;
}

```

Python

```

# Python function to print leaders in array
def printLeaders(arr, size):

    max_from_right = arr[size-1]
    print max_from_right,
    for i in range( size-2, 0, -1):
        if max_from_right < arr[i]:
            print arr[i],
            max_from_right = arr[i]

# Driver function
arr = [16, 17, 4, 3, 5, 2]
printLeaders(arr, len(arr))

# This code contributed by _Devesh Agrawal_

```

2 5 17

Time Complexity: O(n)

Sort elements by frequency | Set 1

Print the elements of an array in the decreasing frequency if 2 numbers have same frequency then print the one which came first.

Examples:

```
Input: arr[] = {2, 5, 2, 8, 5, 6, 8, 8}
Output: arr[] = {8, 8, 8, 2, 2, 5, 6}
```

```
Input: arr[] = {2, 5, 2, 6, -1, 9999999, 5, 8, 8, 8}
Output: arr[] = {8, 8, 8, 2, 2, 5, 5, 6, -1, 9999999}
```

METHOD 1 (Use Sorting)

- 1) Use a sorting algorithm to sort the elements $O(n \log n)$
- 2) Scan the sorted array and construct a 2D array of element and count $O(n)$.
- 3) Sort the 2D array according to count $O(n \log n)$.

Example:

```
Input 2 5 2 8 5 6 8 8
```

```
After sorting we get
2 2 5 5 6 8 8 8
```

```
Now construct the 2D array as
```

```
2, 2
5, 2
6, 1
8, 3
```

```
Sort by count
```

```
8, 3
2, 2
5, 2
6, 1
```

How to maintain order of elements if frequency is same?

The above approach doesn't make sure order of elements if frequency is same. To handle this, we should use indexes in step 3, if two counts are same then we should first process(or print) the element with lower index. In step 1, we should store the indexes instead of elements.

```
Input 5 2 2 8 5 6 8 8
```

```
After sorting we get
Element 2 2 5 5 6 8 8 8
Index   1 2 0 4 5 3 6 7
```

```
Now construct the 2D array as
```

```
Index, Count
1,      2
0,      2
5,      1
3,      3
```

```
Sort by count (consider indexes in case of tie)
```

```
3, 3
0, 2
1, 2
5, 1
```

```
Print the elements using indexes in the above 2D array.
```

Below is C++ implementation of above approach.

```
// Sort elements by frequency. If two elements have same
// count, then put the elements that appears first
#include<bits/stdc++.h>
using namespace std;

// Used for sorting
struct ele
{
    int count, index, val;
};

// Used for sorting by value
```

```

bool mycomp(struct ele a, struct ele b) {
    return (a.val < b.val);
}

// Used for sorting by frequency. And if frequency is same,
// then by appearance
bool mycomp2(struct ele a, struct ele b) {
    if (a.count != b.count) return (a.count < b.count);
    else return a.index > b.index;
}

void sortByFrequency(int arr[], int n)
{
    struct ele element[n];
    for (int i = 0; i < n; i++)
    {
        element[i].index = i; /* Fill Indexes */
        element[i].count = 0; /* Initialize counts as 0 */
        element[i].val = arr[i]; /* Fill values in structure
                                   elements */
    }

    /* Sort the structure elements according to value,
       we used stable sort so relative order is maintained. */
    stable_sort(element, element+n, mycomp);

    /* initialize count of first element as 1 */
    element[0].count = 1;

    /* Count occurrences of remaining elements */
    for (int i = 1; i < n; i++)
    {
        if (element[i].val == element[i-1].val)
        {
            element[i].count += element[i-1].count+1;

            /* Set count of previous element as -1 , we are
               doing this because we'll again sort on the
               basis of counts (if counts are equal than on
               the basis of index)*/
            element[i-1].count = -1;

            /* Retain the first index (Remember first index
               is always present in the first duplicate we
               used stable sort. */
            element[i].index = element[i-1].index;
        }
        else
        {
            /* Else If previous element is not equal to current
               so set the count to 1 */
            element[i].count = 1;
        }
    }

    /* Now we have counts and first index for each element so now
       sort on the basis of count and in case of tie use index
       to sort.*/
    stable_sort(element, element+n, mycomp2);
    for (int i = n-1, index=0; i >= 0; i--)
        if (element[i].count != -1)
            for (int j=0; j<element[i].count; j++)
                arr[index++] = element[i].val;
    }

    // Driver program
    int main()
    {
        int arr[] = {2, 5, 2, 6, -1, 9999999, 5, 8, 8, 8};
        int n = sizeof(arr)/sizeof(arr[0]);

        sortByFrequency(arr, n);

        for (int i=0; i<n; i++)
            cout << arr[i] << " ";
        return 0;
    }
}

```

Output:

8 8 8 2 2 5 5 6 -1 9999999

Thanks to [Gaurav Ahirwar](#) for providing above implementation.

METHOD 2(Use BST and Sorting)

1. Insert elements in BST one by one and if an element is already present then increment the count of the node. Node of the Binary Search Tree (used in this approach) will be as follows.

```
struct tree
{
    int element;
    int first_index /*To handle ties in counts*/
    int count;
}BST;
```

2. Store the first indexes and corresponding counts of BST in a 2D array.

3 Sort the 2D array according to counts (and use indexes in case of tie).

Time Complexity: $O(n \log n)$ if a [Self Balancing Binary Search Tree](#) is used. This is implemented in [Set 2](#).

METHOD 3(Use Hashing and Sorting)

Using a hashing mechanism, we can store the elements (also first index) and their counts in a hash. Finally, sort the hash elements according to their counts.

Set 2:

[Sort elements by frequency | Set 2](#)

Count Inversions in an array

Inversion Count for an array indicates how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Formally speaking, two elements $a[i]$ and $a[j]$ form an inversion if $a[i] > a[j]$ and $i < j$ **Example:**

The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

METHOD 1 (Simple)

For each element, count number of elements which are on right side of it and are smaller than it.

```
int getInvCount(int arr[], int n)
{
    int inv_count = 0;
    int i, j;

    for(i = 0; i < n - 1; i++)
        for(j = i+1; j < n; j++)
            if(arr[i] > arr[j])
                inv_count++;

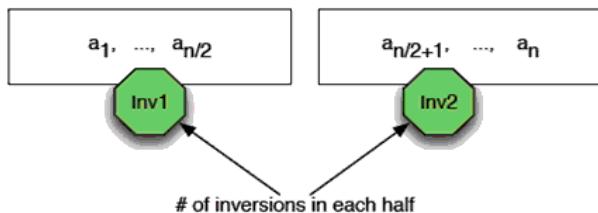
    return inv_count;
}

/* Driver program to test above functions */
int main(int argc, char** argv)
{
    int arr[] = {1, 20, 6, 4, 5};
    printf(" Number of inversions are %d \n", getInvCount(arr, 5));
    getchar();
    return 0;
}
```

Time Complexity: $O(n^2)$

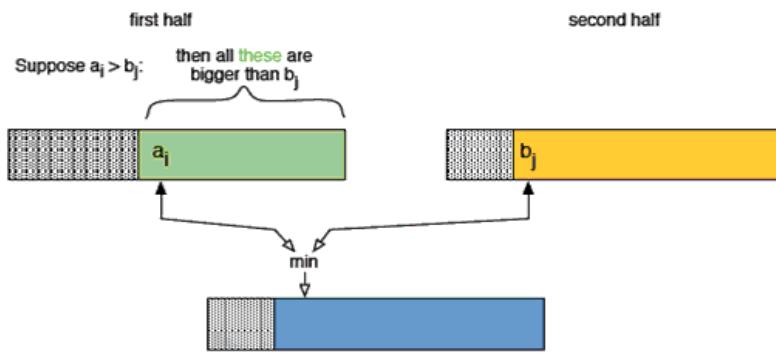
METHOD 2(Enhance Merge Sort)

Suppose we know the number of inversions in the left half and right half of the array (let be $inv1$ and $inv2$), what kinds of inversions are not accounted for in $inv1 + inv2$? The answer is the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().

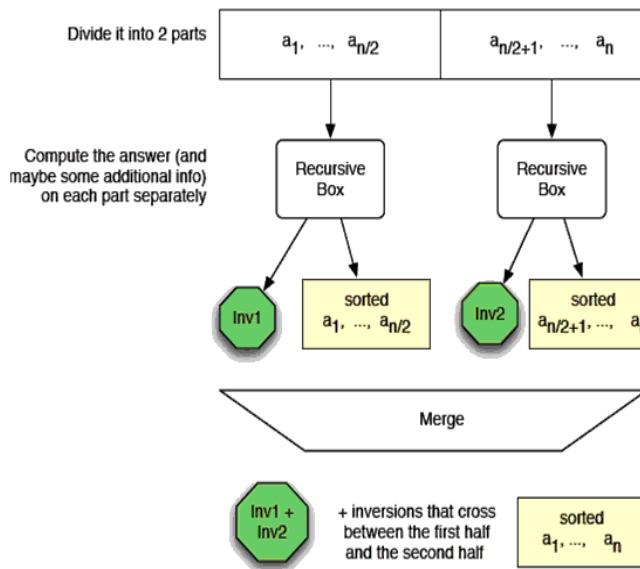


How to get number of inversions in merge()?

In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if $a[i]$ is greater than $a[j]$, then there are $(mid - i)$ inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray ($a[i+1], a[i+2], \dots, a[mid]$) will be greater than $a[j]$.



The complete picture:



Implementation:

```

#include <stdio.h>
#include <stdlib.h>

int _mergeSort(int arr[], int temp[], int left, int right);
int merge(int arr[], int temp[], int left, int mid, int right);

/* This function sorts the input array and returns the
   number of inversions in the array */
int mergeSort(int arr[], int array_size)
{
    int *temp = (int *)malloc(sizeof(int)*array_size);
    return _mergeSort(arr, temp, 0, array_size - 1);
}

/* An auxiliary recursive function that sorts the input array and
   returns the number of inversions in the array. */
int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left)
    {
        /* Divide the array into two parts and call _mergeSortAndCountInv()
           for each of the parts */
        mid = (right + left)/2;

        /* Inversion count will be sum of inversions in left-part, right-part
           and number of inversions in merging */
        inv_count = _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid+1, right);

        /*Merge the two parts*/
        inv_count += merge(arr, temp, left, mid+1, right);
    }
    return inv_count;
}
  
```

```

}

/* This funt merges two sorted arrays and returns inversion count in
   the arrays.*/
int merge(int arr[], int temp[], int left, int mid, int right)
{
    int i, j, k;
    int inv_count = 0;

    i = left; /* i is index for left subarray*/
    j = mid; /* i is index for right subarray*/
    k = left; /* i is index for resultant merged subarray*/
    while ((i <= mid - 1) && (j <= right))
    {
        if (arr[i] <= arr[j])
        {
            temp[k++] = arr[i++];
        }
        else
        {
            temp[k++] = arr[j++];
            /*this is tricky -- see above explanation/diagram for merge()*/
            inv_count = inv_count + (mid - i);
        }
    }

    /* Copy the remaining elements of left subarray
       (if there are any) to temp*/
    while (i <= mid - 1)
        temp[k++] = arr[i++];

    /* Copy the remaining elements of right subarray
       (if there are any) to temp*/
    while (j <= right)
        temp[k++] = arr[j++];

    /*Copy back the merged elements to original array*/
    for (i=left; i <= right; i++)
        arr[i] = temp[i];
}

return inv_count;
}

/* Driver progra to test above functions */
int main(int argc, char** args)
{
    int arr[] = {1, 20, 6, 4, 5};
    printf(" Number of inversions are %d \n", mergeSort(arr, 5));
    getchar();
    return 0;
}

```

Note that above code modifies (or sorts) the input array. If we want to count only inversions then we need to create a copy of original array and call mergeSort() on copy.

Time Complexity: O(nlogn)

Algorithmic Paradigm: Divide and Conquer

References:

<http://www.cs.umd.edu/class/fall2009/cmsc451/lectures/Lec08-inversions.pdf>

<http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>

Two elements whose sum is closest to zero

Question: An Array of integers is given, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

For the below array, program should print -80 and 85.

| | | | | | |
|---|----|-----|----|-----|----|
| 1 | 60 | -10 | 70 | -80 | 85 |
|---|----|-----|----|-----|----|

METHOD 1 (Simple)

For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

Implementation

```
# include <stdio.h>
# include <stdlib.h> /* for abs() */
# include <math.h>
void minAbsSumPair(int arr[], int arr_size)
{
    int inv_count = 0;
    int l, r, min_sum, sum, min_l, min_r;

    /* Array should have at least two elements*/
    if(arr_size < 2)
    {
        printf("Invalid Input");
        return;
    }

    /* Initialization of values */
    min_l = 0;
    min_r = 1;
    min_sum = arr[0] + arr[1];

    for(l = 0; l < arr_size - 1; l++)
    {
        for(r = l+1; r < arr_size; r++)
        {
            sum = arr[l] + arr[r];
            if(abs(min_sum) > abs(sum))
            {
                min_sum = sum;
                min_l = l;
                min_r = r;
            }
        }
    }

    printf(" The two elements whose sum is minimum are %d and %d",
           arr[min_l], arr[min_r]);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 60, -10, 70, -80, 85};
    minAbsSumPair(arr, 6);
    getchar();
    return 0;
}
```

Time complexity: O(n^2)

METHOD 2 (Use Sorting)

Thanks to baskin for suggesting this approach. We recommend to read [this post](#) for background of this approach.

Algorithm

- 1) Sort all the elements of the input array.
- 2) Use two index variables l and r to traverse from left and right ends respectively. Initialize l as 0 and r as n-1.
- 3) sum= a[l] + a[r]
- 4) If sum is -ve, then l++
- 5) If sum is +ve, then r
- 6) Keep track of abs min sum

7) Repeat steps 3, 4, 5 and 6 while $l < r$ **Implementation**

```
# include <stdio.h>
# include <math.h>
# include <limits.h>

void quickSort(int *, int, int);

/* Function to print pair of elements having minimum sum */
void minAbsSumPair(int arr[], int n)
{
    // Variables to keep track of current sum and minimum sum
    int sum, min_sum = INT_MAX;

    // left and right index variables
    int l = 0, r = n-1;

    // variable to keep track of the left and right pair for min_sum
    int min_l = l, min_r = n-1;

    /* Array should have at least two elements*/
    if(n < 2)
    {
        printf("Invalid Input");
        return;
    }

    /* Sort the elements */
    quickSort(arr, l, r);

    while(l < r)
    {
        sum = arr[l] + arr[r];

        /*If abs(sum) is less then update the result items*/
        if(abs(sum) < abs(min_sum))
        {
            min_sum = sum;
            min_l = l;
            min_r = r;
        }
        if(sum < 0)
            l++;
        else
            r--;
    }

    printf(" The two elements whose sum is minimum are %d and %d",
           arr[min_l], arr[min_r]);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 60, -10, 70, -80, 85};
    int n = sizeof(arr)/sizeof(arr[0]);
    minAbsSumPair(arr, n);
    getchar();
    return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
 PURPOSE */
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int si, int ei)
{
    int x = arr[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(arr[j] <= x)
        {
```

```

        i++;
        exchange(&arr[i], &arr[j]);
    }
}

exchange (&arr[i + 1], &arr[ei]);
return (i + 1);
}

/* Implementation of Quick Sort
arr[] --> Array to be sorted
si  --> Starting index
ei  --> Ending index
*/
void quickSort(int arr[], int si, int ei)
{
    int pi;      /* Partitioning index */
    if(si < ei)
    {
        pi = partition(arr, si, ei);
        quickSort(arr, si, pi - 1);
        quickSort(arr, pi + 1, ei);
    }
}

```

Time Complexity: complexity to sort + complexity of finding the optimum pair = $O(n \log n) + O(n) = O(n \log n)$

Asked by Vineet

Find the smallest and second smallest element in an array

Write an efficient C program to find smallest and second smallest element in an array.

Example:

Input: arr[] = {12, 13, 1, 10, 34, 1}
Output: The smallest element is 1 and
second Smallest element is 10

A **Simple Solution** is to sort the array in increasing order. The first two elements in sorted array would be two smallest elements. Time complexity of this solution is $O(n \log n)$.

A **Better Solution** is to scan the array twice. In first traversal find the minimum element. Let this element be x . In second traversal, find the smallest element greater than x . Time complexity of this solution is $O(n)$.

The above solution requires two traversals of input array.

An **Efficient Solution** can find the minimum two elements in one traversal. Below is complete algorithm.

Algorithm:

- 1) Initialize both first and second smallest as `INT_MAX`
`first = second = INT_MAX`
- 2) Loop through all the elements.
 - a) If the current element is smaller than `first`, then update `first` and `second`.
 - b) Else if the current element is smaller than `second` then update `second`

Implementation:

C/C++

```
// C program to find smallest and second smallest elements
#include <stdio.h>
#include <limits.h> /* For INT_MAX */

void print2Smallest(int arr[], int arr_size)
{
    int i, first, second;

    /* There should be atleast two elements */
    if (arr_size < 2)
    {
        printf(" Invalid Input ");
        return;
    }

    first = second = INT_MAX;
    for (i = 0; i < arr_size ; i++)
    {
        /* If current element is smaller than first
         * then update both first and second */
        if (arr[i] < first)
        {
            second = first;
            first = arr[i];
        }

        /* If arr[i] is in between first and second
         * then update second */
        else if (arr[i] < second && arr[i] != first)
            second = arr[i];
    }

    if (second == INT_MAX)
        printf("There is no second smallest element\n");
    else
        printf("The smallest element is %d and second "
               "Smallest element is %d\n", first, second);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {12, 13, 1, 10, 34, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
```

```

    print2Smallest(arr, n);
    return 0;
}

```

Java

```

// Java program to find smallest and second smallest elements
import java.io.*;

class SecondSmallest
{
    /* Function to print first smallest and second smallest
       elements */
    static void print2Smallest(int arr[])
    {
        int first, second, arr_size = arr.length;

        /* There should be atleast two elements */
        if (arr_size < 2)
        {
            System.out.println(" Invalid Input ");
            return;
        }

        first = second = Integer.MAX_VALUE;
        for (int i = 0; i < arr_size ; i++)
        {
            /* If current element is smaller than first
               then update both first and second */
            if (arr[i] < first)
            {
                second = first;
                first = arr[i];
            }

            /* If arr[i] is in between first and second
               then update second */
            else if (arr[i] < second && arr[i] != first)
                second = arr[i];
        }
        if (second == Integer.MAX_VALUE)
            System.out.println("There is no second" +
                               "smallest element");
        else
            System.out.println("The smallest element is " +
                               first + " and second Smallest" +
                               " element is " + second);
    }

    /* Driver program to test above functions */
    public static void main (String[] args)
    {
        int arr[] = {12, 13, 1, 10, 34, 1};
        print2Smallest(arr);
    }
}
/*This code is contributed by Devesh Agrawal*/

```

Python

```

# Python program to find smallest and second smallest elements
import sys

def print2Smallest(arr):

    # There should be atleast two elements
    arr_size = len(arr)
    if arr_size < 2:
        print "Invalid Input"
        return

    first = second = sys.maxint
    for i in range(0, arr_size):

        # If current element is smaller than first then
        # update both first and second
        if arr[i] < first:
            second = first
            first = arr[i]

```

```
# If arr[i] is in between first and second then
# update second
elif (arr[i] < second and arr[i] != first):
    second = arr[i];

if (second == sys.maxint):
    print "No second smallest element"
else:
    print 'The smallest element is',first,'and' \
          ' second smallest element is',second

# Driver function to test above function
arr = [12, 13, 1, 10, 34, 1]
print2Smallest(arr)

# This code is contributed by Devesh Agrawal
```

The smallest element is 1 and second Smallest element is 10

The same approach can be used to find the largest and second largest elements in an array.

Time Complexity: O(n)

Check for Majority Element in a sorted array

Question: Write a C function to find if a given integer x appears more than $n/2$ times in a sorted array of n integers.

Basically, we need to write a function say `isMajority()` that takes an array (`arr[]`), arrays size (n) and a number to be searched (x) as parameters and returns true if x is a [majority element](#) (present more than $n/2$ times).

Examples:

Input: `arr[] = {1, 2, 3, 3, 3, 10}, x = 3`
Output: True (x appears more than $n/2$ times in the given array)

Input: `arr[] = {1, 1, 2, 4, 4, 6, 6}, x = 4`
Output: False (x doesn't appear more than $n/2$ times in the given array)

Input: `arr[] = {1, 1, 1, 2, 2}, x = 1`
Output: True (x appears more than $n/2$ times in the given array)

METHOD 1 (Using Linear Search)

Linearly search for the first occurrence of the element, once you find it (let at index i), check element at index $i + n/2$. If element is present at $i+n/2$ then return 1 else return 0.

C

```
/* C Program to check for majority element in a sorted array */
#include <stdio.h>
#include <stdbool.h>

bool isMajority(int arr[], int n, int x)
{
    int i;

    /* get last index according to n (even or odd) */
    int last_index = n%2? (n/2+1): (n/2);

    /* search for first occurrence of x in arr[]*/
    for (i = 0; i < last_index; i++)
    {
        /* check if x is present and is present more than n/2
         * times */
        if (arr[i] == x && arr[i+n/2] == x)
            return 1;
    }
    return 0;
}

/* Driver program to check above function */
int main()
{
    int arr[] ={1, 2, 3, 4, 4, 4, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 4;
    if (isMajority(arr, n, x))
        printf("%d appears more than %d times in arr[]",
               x, n/2);
    else
        printf("%d does not appear more than %d times in arr[]",
               x, n/2);

    return 0;
}
```

Java

```
/* Program to check for majority element in a sorted array */
import java.io.*;

class Majority {

    static boolean isMajority(int arr[], int n, int x)
    {
        int i, last_index = 0;

        /* get last index according to n (even or odd) */
        last_index = (n%2==0)? n/2: n/2+1;

        /* search for first occurrence of x in arr[]*/
    }
}
```

```

        for (i = 0; i < last_index; i++)
    {
        /* check if x is present and is present more
         * than n/2 times */
        if (arr[i] == x && arr[i+n/2] == x)
            return true;
    }
    return false;
}

/* Driver function to check for above functions*/
public static void main (String[] args) {
    int arr[] = {1, 2, 3, 4, 4, 4, 4};
    int n = arr.length;
    int x = 4;
    if (isMajority(arr, n, x)==true)
        System.out.println(x+" appears more than "+
                           n/2+" times in arr[]");
    else
        System.out.println(x+" does not appear more than "+
                           n/2+" times in arr[]");
}
}
/*This article is contributed by Devesh Agrawal*/

```

4 appears more than 3 times in arr[]

Time Complexity: O(n)

METHOD 2 (Using Binary Search)

Use binary search methodology to find the first occurrence of the given number. The criteria for binary search is important here.

C

```

/* Program to check for majority element in a sorted array */
# include <stdio.h>
# include <stdbool.h>

/* If x is present in arr[low...high] then returns the index of
first occurrence of x, otherwise returns -1 */
int _binarySearch(int arr[], int low, int high, int x);

/* This function returns true if the x is present more than n/2
times in arr[] of size n */
bool isMajority(int arr[], int n, int x)
{
    /* Find the index of first occurrence of x in arr[] */
    int i = _binarySearch(arr, 0, n-1, x);

    /* If element is not present at all, return false*/
    if (i == -1)
        return false;

    /* check if the element is present more than n/2 times */
    if (((i + n/2) <= (n -1)) && arr[i + n/2] == x)
        return true;
    else
        return false;
}

/* If x is present in arr[low...high] then returns the index of
first occurrence of x, otherwise returns -1 */
int _binarySearch(int arr[], int low, int high, int x)
{
    if (high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/

        /* Check if arr[mid] is the first occurrence of x.
        arr[mid] is first occurrence if x is one of the following
        is true:
        (i) mid == 0 and arr[mid] == x
        (ii) arr[mid-1] < x and arr[mid] == x
        */
        if ( (mid == 0 || x > arr[mid-1]) && (arr[mid] == x) )

```

```

        return mid;
    else if (x > arr[mid])
        return _binarySearch(arr, (mid + 1), high, x);
    else
        return _binarySearch(arr, low, (mid -1), x);
    }

    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 2, 3, 3, 3, 3, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    if (isMajority(arr, n, x))
        printf("%d appears more than %d times in arr[]",
               x, n/2);
    else
        printf("%d does not appear more than %d times in arr[]",
               x, n/2);
    return 0;
}

```

Java

```

/* Program to check for majority element in a sorted array */
import java.io.*;

class Majority {

    /* If x is present in arr[low...high] then returns the index of
       first occurrence of x, otherwise returns -1 */
    static int _binarySearch(int arr[], int low, int high, int x)
    {
        if (high >= low)
        {
            int mid = (low + high)/2; /*low + (high - low)/2;*/

            /* Check if arr[mid] is the first occurrence of x.
               arr[mid] is first occurrence if x is one of the following
               is true:
               (i)  mid == 0 and arr[mid] == x
               (ii) arr[mid-1] < x and arr[mid] == x
            */
            if ( (mid == 0 || x > arr[mid-1]) && (arr[mid] == x) )
                return mid;
            else if (x > arr[mid])
                return _binarySearch(arr, (mid + 1), high, x);
            else
                return _binarySearch(arr, low, (mid -1), x);
        }

        return -1;
    }

    /* This function returns true if the x is present more than n/2
       times in arr[] of size n */
    static boolean isMajority(int arr[], int n, int x)
    {
        /* Find the index of first occurrence of x in arr[] */
        int i = _binarySearch(arr, 0, n-1, x);

        /* If element is not present at all, return false*/
        if (i == -1)
            return false;

        /* check if the element is present more than n/2 times */
        if (((i + n/2) <= (n -1)) && arr[i + n/2] == x)
            return true;
        else
            return false;
    }

    /*Driver function to check for above functions*/
    public static void main (String[] args) {

        int arr[] = {1, 2, 3, 3, 3, 3, 10};

```

```
int n = arr.length;
int x = 3;
if (isMajority(arr, n, x)==true)
    System.out.println(x + " appears more than "+
                        n/2 + " times in arr[]");
else
    System.out.println(x + " does not appear more than " +
                        n/2 + " times in arr[]");
}
/*This code is contributed by Devesh Agrawal*/
```

3 appears more than 3 times in arr[]

Time Complexity: O(Logn)

Algorithmic Paradigm: Divide and Conquer

Maximum and minimum of an array using minimum number of comparisons

Write a C function to return minimum and maximum in an array. Your program should make minimum number of comparisons.

First of all, how do we return multiple values from a C function? We can do it either using structures or pointers.

We have created a structure named pair (which contains min and max) to return multiple values.

```
struct pair
{
    int min;
    int max;
};
```

And the function declaration becomes: struct pair getMinMax(int arr[], int n) where arr[] is the array of size n whose minimum and maximum are needed.

METHOD 1 (Simple Linear Search)

Initialize values of min and max as minimum and maximum of the first two elements respectively. Starting from 3rd, compare each element with max and min, and change max and min accordingly (i.e., if the element is smaller than min then change min, else if the element is greater than max then change max, else ignore the element)

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
    int min;
    int max;
};

struct pair getMinMax(int arr[], int n)
{
    struct pair minmax;
    int i;

    /* If there is only one element then return it as min and max both*/
    if (n == 1)
    {
        minmax.max = arr[0];
        minmax.min = arr[0];
        return minmax;
    }

    /* If there are more than one elements, then initialize min
       and max*/
    if (arr[0] > arr[1])
    {
        minmax.max = arr[0];
        minmax.min = arr[1];
    }
    else
    {
        minmax.max = arr[1];
        minmax.min = arr[0];
    }

    for (i = 2; i < n; i++)
    {
        if (arr[i] > minmax.max)
            minmax.max = arr[i];

        else if (arr[i] < minmax.min)
            minmax.min = arr[i];
    }

    return minmax;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1000, 11, 445, 1, 330, 3000};
    int arr_size = 6;
    struct pair minmax = getMinMax (arr, arr_size);
    printf("\nMinimum element is %d", minmax.min);
    printf("\nMaximum element is %d", minmax.max);
    getchar();
}
```

Time Complexity: O(n)

In this method, total number of comparisons is $1 + 2(n-2)$ in worst case and $1 + n/2$ in best case.

In the above implementation, worst case occurs when elements are sorted in descending order and best case occurs when elements are sorted in ascending order.

METHOD 2 (Tournament Method)

Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.

```
Pair MaxMin(array, array_size)
if array_size = 1
    return element as both max and min
else if array_size = 2
    one comparison to determine max and min
    return that pair
else /* array_size > 2 */
    recur for max and min of left half
    recur for max and min of right half
    one comparison determines true max of the two candidates
    one comparison determines true min of the two candidates
    return the pair of max and min
```

Implementation

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
    int min;
    int max;
};

struct pair getMinMax(int arr[], int low, int high)
{
    struct pair minmax, mml, mmr;
    int mid;

    /* If there is only one element */
    if (low == high)
    {
        minmax.max = arr[low];
        minmax.min = arr[low];
        return minmax;
    }

    /* If there are two elements */
    if (high == low + 1)
    {
        if (arr[low] > arr[high])
        {
            minmax.max = arr[low];
            minmax.min = arr[high];
        }
        else
        {
            minmax.max = arr[high];
            minmax.min = arr[low];
        }
        return minmax;
    }

    /* If there are more than 2 elements */
    mid = (low + high)/2;
    mml = getMinMax(arr, low, mid);
    mmr = getMinMax(arr, mid+1, high);

    /* compare minimums of two parts*/
    if (mml.min < mmr.min)
        minmax.min = mml.min;
    else
        minmax.min = mmr.min;

    /* compare maximums of two parts*/
    if (mml.max > mmr.max)
        minmax.max = mml.max;
    else
        minmax.max = mmr.max;
}
```

```

minmax.max = mmr.max;
return minmax;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1000, 11, 445, 1, 330, 3000};
    int arr_size = 6;
    struct pair minmax = getMinMax(arr, 0, arr_size-1);
    printf("\nMinimum element is %d", minmax.min);
    printf("\nMaximum element is %d", minmax.max);
    getchar();
}

```

Time Complexity: O(n)

Total number of comparisons: let number of comparisons be T(n). T(n) can be written as follows:

Algorithmic Paradigm: Divide and Conquer

$$\begin{aligned}
 T(n) &= T(\text{floor}(n/2)) + T(\text{ceil}(n/2)) + 2 \\
 T(2) &= 1 \\
 T(1) &= 0
 \end{aligned}$$

If n is a power of 2, then we can write T(n) as:

$$T(n) = 2T(n/2) + 2$$

After solving above recursion, we get

$$T(n) = 3/2n - 2$$

Thus, the approach does $3/2n - 2$ comparisons if n is a power of 2. And it does more than $3/2n - 2$ comparisons if n is not a power of 2.

METHOD 3 (Compare in Pairs)

If n is odd then initialize min and max as first element.

If n is even then initialize min and max as minimum and maximum of the first two elements respectively.

For rest of the elements, pick them in pairs and compare their maximum and minimum with max and min respectively.

```

#include<stdio.h>

/* structure is used to return two values from minMax() */
struct pair
{
    int min;
    int max;
};

struct pair getMinMax(int arr[], int n)
{
    struct pair minmax;
    int i;

    /* If array has even number of elements then
       initialize the first two elements as minimum and
       maximum */
    if (n%2 == 0)
    {
        if (arr[0] > arr[1])
        {
            minmax.max = arr[0];
            minmax.min = arr[1];
        }
        else
        {
            minmax.min = arr[0];
            minmax.max = arr[1];
        }
        i = 2; /* set the starting index for loop */
    }

    /* If array has odd number of elements then
       initialize the first element as minimum and
       maximum */

```

```

else
{
    minmax.min = arr[0];
    minmax.max = arr[0];
    i = 1; /* set the startung index for loop */
}

/* In the while loop, pick elements in pair and
   compare the pair with max and min so far */
while (i < n-1)
{
    if (arr[i] > arr[i+1])
    {
        if(arr[i] > minmax.max)
            minmax.max = arr[i];
        if(arr[i+1] < minmax.min)
            minmax.min = arr[i+1];
    }
    else
    {
        if (arr[i+1] > minmax.max)
            minmax.max = arr[i+1];
        if (arr[i] < minmax.min)
            minmax.min = arr[i];
    }
    i += 2; /* Increment the index by 2 as two
               elements are processed in loop */
}

return minmax;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1000, 11, 445, 1, 330, 3000};
    int arr_size = 6;
    struct pair minmax = getMinMax (arr, arr_size);
    printf("\nMinimum element is %d", minmax.min);
    printf("\nMaximum element is %d", minmax.max);
    getchar();
}

```

Time Complexity: O(n)

Total number of comparisons: Different for even and odd n, see below:

```

If n is odd:      3*(n-1)/2
If n is even:    1 Initial comparison for initializing min and max,
                  and 3(n-2)/2 comparisons for rest of the elements
                  = 1 + 3*(n-2)/2 = 3n/2 -2

```

Second and third approaches make equal number of comparisons when n is a power of 2.

In general, method 3 seems to be the best.

Segregate 0s and 1s in an array

Asked by [kapil](#).

You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
Output array = [0, 0, 0, 0, 1, 1, 1, 1, 1]

Method 1 (Count 0s or 1s)

Thanks to [Naveen](#) for suggesting this method.

- 1) Count the number of 0s. Let count be C.
- 2) Once we have count, we can put C 0s at the beginning and 1s at the remaining n - C positions in array.

Time Complexity: O(n)

The method 1 traverses the array two times. Method 2 does the same in a single pass.

Method 2 (Use two indexes to traverse)

Maintain two indexes. Initialize first index *left* as 0 and second index *right* as n-1.

Do following while *left* < *right*

- a) Keep incrementing index *left* while there are 0s at it
- b) Keep decrementing index *right* while there are 1s at it
- c) If *left* < *right* then exchange arr[left] and arr[right]

```
// C program to sort a binary array in one pass
#include<stdio.h>

/*Function to put all 0s on left and all 1s on right*/
void segregate0and1(int arr[], int size)
{
    /* Initialize left and right indexes */
    int left = 0, right = size-1;

    while (left < right)
    {
        /* Increment left index while we see 0 at left */
        while (arr[left] == 0 && left < right)
            left++;

        /* Decrement right index while we see 1 at right */
        while (arr[right] == 1 && left < right)
            right--;

        /* If left is smaller than right then there is a 1 at left
         * and a 0 at right. Exchange arr[left] and arr[right]*/
        if (left < right)
        {
            arr[left] = 0;
            arr[right] = 1;
            left++;
            right--;
        }
    }
}

/* driver program to test */
int main()
{
    int arr[] = {0, 1, 0, 1, 1, 1};
    int i, arr_size = sizeof(arr)/sizeof(arr[0]);

    segregate0and1(arr, arr_size);

    printf("array after segregation ");
    for (i = 0; i < 6; i++)
        printf("%d ", arr[i]);

    getchar();
    return 0;
}
```

Time Complexity: O(n)

k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Method 1 (Use Bubble k times)

Thanks to Shailendra for suggesting this approach.

- 1) Modify [Bubble Sort](#) to run the outer loop at most k times.
- 2) Print the last k elements of the array obtained in step 1.

Time Complexity: $O(nk)$

Like Bubble sort, other sorting algorithms like [Selection Sort](#) can also be modified to get the k largest elements.

Method 2 (Use temporary array)

K largest elements from arr[0..n-1]

- 1) Store the first k elements in a temporary array temp[0..k-1].
- 2) Find the smallest element in temp[], let the smallest element be *min*.
- 3) For each element *x* in arr[k] to arr[n-1]
If *x* is greater than the *min* then remove *min* from temp[] and insert *x*.
- 4) Print final k elements of temp[]

Time Complexity: $O((n-k)*k)$. If we want the output sorted then $O((n-k)*k + k\log k)$

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)

- 1) Sort the elements in descending order in $O(n\log n)$
- 2) Print the first k numbers of the sorted array $O(k)$.

Time complexity: $O(n\log n)$

Method 4 (Use Max Heap)

- 1) Build a Max Heap tree in $O(n)$
- 2) Use [Extract Max](#) k times to get k maximum elements from the Max Heap $O(k\log n)$

Time complexity: $O(n + k\log n)$

Method 5(Use Order Statistics)

- 1) Use order statistic algorithm to find the kth largest element. Please [see the topic selection in worst-case linear time](#) $O(n)$
- 2) Use [QuickSort](#) Partition algorithm to partition around the kth largest number $O(n)$.
- 3) Sort the k-1 elements (elements greater than the kth largest element) $O(k\log k)$. This step is needed only if sorted output is required.

Time complexity: $O(n)$ if we dont need the sorted output, otherwise $O(n+k\log k)$

Thanks to [Shilpi](#) for suggesting the first two approaches.

Method 6 (Use Min Heap)

This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.

Thanks to [geek4u](#) for suggesting this method.

- 1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. $O(k)$
- 2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.
 - a) If the element is greater than the root then make it root and call [heapify](#) for MH
 - b) Else ignore it.

// The step 2 is $O((n-k)\log k)$
- 3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: $O(k + (n-k)\log k)$ without sorted output. If sorted output is needed then $O(k + (n-k)\log k + k\log k)$

All of the above methods can also be used to find the kth largest (or smallest) element.

References:

http://en.wikipedia.org/wiki/Selection_algorithm

Asked by [geek4u](#)

Maximum difference between two elements such that larger element appears after the smaller number

Given an array arr[] of integers, find out the difference between any two elements **such that larger element appears after the smaller number** in arr[].

Examples: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Diff between 10 and 2). If array is [7, 9, 5, 6, 3, 2] then returned value should be 2 (Diff between 7 and 9)

Method 1 (Simple)

Use two loops. In the outer loop, pick elements one by one and in the inner loop calculate the difference of the picked element with every other element in the array and compare the difference with the maximum difference calculated so far.

```
#include<stdio.h>

/* The function assumes that there are at least two
elements in array.
The function returns a negative value if the array is
sorted in decreasing order.
Returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
    int max_diff = arr[1] - arr[0];
    int i, j;
    for(i = 0; i < arr_size; i++)
    {
        for(j = i+1; j < arr_size; j++)
        {
            if(arr[j] - arr[i] > max_diff)
                max_diff = arr[j] - arr[i];
        }
    }
    return max_diff;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 2, 90, 10, 110};
    printf("Maximum difference is %d", maxDiff(arr, 5));
    getchar();
    return 0;
}
```

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Method 2 (Tricky and Efficient)

In this method, instead of taking difference of the picked element with every other element, we take the difference with the minimum element found so far. So we need to keep track of 2 things:

- 1) Maximum difference found so far (`max_diff`).
- 2) Minimum number visited so far (`min_element`).

```
#include<stdio.h>

/* The function assumes that there are at least two
elements in array.
The function returns a negative value if the array is
sorted in decreasing order.
Returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
    int max_diff = arr[1] - arr[0];
    int min_element = arr[0];
    int i;
    for(i = 1; i < arr_size; i++)
    {
        if (arr[i] - min_element > max_diff)
            max_diff = arr[i] - min_element;
        if (arr[i] < min_element)
            min_element = arr[i];
    }
    return max_diff;
}

/* Driver program to test above function */
```

```

int main()
{
    int arr[] = {1, 2, 6, 80, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum difference is %d", maxDiff(arr, size));
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Auxiliary Space: O(1)

Like min element, we can also keep track of max element from right side. See below code suggested by Katamaran

```

int maxDiff(int arr[], int n)
{
    int maxDiff = -1; // Initialize Result

    int maxRight = arr[n-1]; // Initialize max element from right side

    for (int i = n-2; i >= 0; i--)
    {
        if (arr[i] > maxRight)
            maxRight = arr[i];
        else
        {
            int diff = maxRight - arr[i];
            if (diff > maxDiff)
            {
                maxDiff = diff;
            }
        }
    }
    return maxDiff;
}

```

Method 3 (Another Tricky Solution)

First find the difference between the adjacent elements of the array and store all differences in an auxiliary array diff[] of size n-1. Now this problem turns into finding the maximum sum subarray of this difference array.

Thanks to Shubham Mittal for suggesting this solution.

```

#include<stdio.h>

int maxDiff(int arr[], int n)
{
    // Create a diff array of size n-1. The array will hold
    // the difference of adjacent elements
    int diff[n-1];
    for (int i=0; i < n-1; i++)
        diff[i] = arr[i+1] - arr[i];

    // Now find the maximum sum subarray in diff array
    int max_diff = diff[0];
    for (int i=1; i < n-1; i++)
    {
        if (diff[i-1] > 0)
            diff[i] += diff[i-1];
        if (max_diff < diff[i])
            max_diff = diff[i];
    }
    return max_diff;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {80, 2, 6, 3, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum difference is %d", maxDiff(arr, size));
    return 0;
}

```

Output:

98

This method is also O(n) time complexity solution, but it requires O(n) extra space

Time Complexity: O(n)

Auxiliary Space: O(n)

We can modify the above method to work in O(1) extra space. Instead of creating an auxiliary array, we can calculate diff and max sum in same loop. Following is the space optimized version.

```
int maxDiff (int arr[], int n)
{
    // Initialize diff, current sum and max sum
    int diff = arr[1]-arr[0];
    int curr_sum = diff;
    int max_sum = curr_sum;

    for(int i=1; i<n-1; i++)
    {
        // Calculate current diff
        diff = arr[i+1]-arr[i];

        // Calculate current sum
        if (curr_sum > 0)
            curr_sum += diff;
        else
            curr_sum = diff;

        // Update max sum, if needed
        if (curr_sum > max_sum)
            max_sum = curr_sum;
    }

    return max_sum;
}
```

Time Complexity: O(n)

Auxiliary Space: O(1)

Union and Intersection of two sorted arrays

Given two sorted arrays, find their union and intersection.

For example, if the input arrays are:

arr1[] = {1, 3, 4, 5, 7}
arr2[] = {2, 3, 5, 6}

Then your program should print Union as {1, 2, 3, 4, 5, 6, 7} and Intersection as {3, 5}.

Algorithm Union(arr1[], arr2[]):

For union of two arrays, follow the following merge procedure.

- 1) Use two index variables i and j, initial values i = 0, j = 0
- 2) If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i.
- 3) If arr1[i] is greater than arr2[j] then print arr2[j] and increment j.
- 4) If both are same then print any of them and increment both i and j.
- 5) Print remaining elements of the larger array.

```
#include<stdio.h>

/* Function prints union of arr1[] and arr2[]
   m is the number of elements in arr1[]
   n is the number of elements in arr2[] */
int printUnion(int arr1[], int arr2[], int m, int n)
{
    int i = 0, j = 0;
    while (i < m && j < n)
    {
        if (arr1[i] < arr2[j])
            printf(" %d ", arr1[i++]);
        else if (arr2[j] < arr1[i])
            printf(" %d ", arr2[j++]);
        else
        {
            printf(" %d ", arr2[j++]);
            i++;
        }
    }

    /* Print remaining elements of the larger array */
    while(i < m)
        printf(" %d ", arr1[i++]);
    while(j < n)
        printf(" %d ", arr2[j++]);
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {1, 2, 4, 5, 6};
    int arr2[] = {2, 3, 5, 7};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);
    printUnion(arr1, arr2, m, n);
    getchar();
    return 0;
}
```

Time Complexity: O(m+n)

Algorithm Intersection(arr1[], arr2[]):

For Intersection of two arrays, print the element only if the element is present in both arrays.

- 1) Use two index variables i and j, initial values i = 0, j = 0
- 2) If arr1[i] is smaller than arr2[j] then increment i.
- 3) If arr1[i] is greater than arr2[j] then increment j.
- 4) If both are same then print any of them and increment both i and j.

```
#include<stdio.h>

/* Function prints Intersection of arr1[] and arr2[]
   m is the number of elements in arr1[]
   n is the number of elements in arr2[] */
int printIntersection(int arr1[], int arr2[], int m, int n)
{
    int i = 0, j = 0;
```

```

if (arr1[i] < arr2[j])
    i++;
else if (arr2[j] < arr1[i])
    j++;
else /* if arr1[i] == arr2[j] */
{
    printf(" %d ", arr2[j++]);
    i++;
}
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {1, 2, 4, 5, 6};
    int arr2[] = {2, 3, 5, 7};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);
    printIntersection(arr1, arr2, m, n);
    getchar();
    return 0;
}

```

Time Complexity: $O(m+n)$

Another approach that is useful when difference between sizes of two given arrays is significant.

The idea is to iterate through the shorter array and do a binary search for every element of short array in big array (note that arrays are sorted). Time complexity of this solution is $O(\min(m\log n, n\log m))$. This solution works better than the above approach when ratio of larger length to smaller is more than logarithmic order.

See following post for unsorted arrays.

[Find Union and Intersection of two unsorted arrays](#)

Floor and Ceiling in a sorted array

Given a sorted array and a value x, the ceiling of x is the smallest element in array greater than or equal to x, and the floor is the greatest element smaller than or equal to x. Assume that the array is sorted in non-decreasing order. Write efficient functions to find floor and ceiling of x.

```
For example, let the input array be {1, 2, 8, 10, 10, 12, 19}
For x = 0:    floor doesn't exist in array, ceil = 1
For x = 1:    floor = 1, ceil = 1
For x = 5:    floor = 2, ceil = 8
For x = 20:   floor = 19, ceil doesn't exist in array
```

In below methods, we have implemented only ceiling search functions. Floor search can be implemented in the same way.

Method 1 (Linear Search)

Algorithm to search ceiling of x:

- 1) If x is smaller than or equal to the first element in array then return 0(index of first element)
- 2) Else Linearly search for an index i such that x lies between arr[i] and arr[i+1].
- 3) If we do not find an index i in step 2, then return -1

```
#include<stdio.h>

/* Function to get index of ceiling of x in arr[low..high] */
int ceilSearch(int arr[], int low, int high, int x)
{
    int i;

    /* If x is smaller than or equal to first element,
       then return the first element */
    if(x <= arr[low])
        return low;

    /* Otherwise, linearly search for ceil value */
    for(i = low; i < high; i++)
    {
        if(arr[i] == x)
            return i;

        /* if x lies between arr[i] and arr[i+1] including
           arr[i+1], then return arr[i+1] */
        if(arr[i] < x && arr[i+1] >= x)
            return i+1;
    }

    /* If we reach here then x is greater than the last element
       of the array, return -1 in this case */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 2, 8, 10, 10, 12, 19};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int index = ceilSearch(arr, 0, n-1, x);
    if(index == -1)
        printf("Ceiling of %d doesn't exist in array ", x);
    else
        printf("ceiling of %d is %d", x, arr[index]);
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Method 2 (Binary Search)

Instead of using linear search, binary search is used here to find out the index. Binary search reduces time complexity to O(Logn).

```
#include<stdio.h>

/* Function to get index of ceiling of x in arr[low..high]*/
int ceilSearch(int arr[], int low, int high, int x)
{
    int mid;

    /* If x is smaller than or equal to the first element,
       then return the first element */
    if(x <= arr[low])
        return low;
```

```

if(x <= arr[low])
    return low;

/* If x is greater than the last element, then return -1 */
if(x > arr[high])
    return -1;

/* get the index of middle element of arr[low..high]*/
mid = (low + high)/2; /* low + (high - low)/2 */

/* If x is same as middle element, then return mid */
if(arr[mid] == x)
    return mid;

/* If x is greater than arr[mid], then either arr[mid + 1]
   is ceiling of x or ceiling lies in arr[mid+1...high] */
else if(arr[mid] < x)
{
    if(mid + 1 <= high && x <= arr[mid+1])
        return mid + 1;
    else
        return ceilSearch(arr, mid+1, high, x);
}

/* If x is smaller than arr[mid], then either arr[mid]
   is ceiling of x or ceiling lies in arr[mid-1...high] */
else
{
    if(mid - 1 >= low && x > arr[mid-1])
        return mid;
    else
        return ceilSearch(arr, low, mid - 1, x);
}
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 2, 8, 10, 10, 12, 19};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 20;
    int index = ceilSearch(arr, 0, n-1, x);
    if(index == -1)
        printf("Ceiling of %d doesn't exist in array ", x);
    else
        printf("ceiling of %d is %d", x, arr[index]);
    getchar();
    return 0;
}

```

Time Complexity: O(Logn)

A Product Array Puzzle

Given an array arr[] of n integers, construct a Product Array prod[] (of same size) such that prod[i] is equal to the product of all the elements of arr[] except arr[i]. Solve it **without division operator and in O(n)**.

Example:

arr[] = {10, 3, 5, 6, 2}
prod[] = {180, 600, 360, 300, 900}

Algorithm:

- 1) Construct a temporary array left[] such that left[i] contains product of all elements on left of arr[i] excluding arr[i].
- 2) Construct another temporary array right[] such that right[i] contains product of all elements on on right of arr[i] excluding arr[i].
- 3) To get prod[], multiply left[] and right[].

Implementation:

```
#include<stdio.h>
#include<stdlib.h>

/* Function to print product array for a given array
   arr[] of size n */
void productArray(int arr[], int n)
{
    /* Allocate memory for temporary arrays left[] and right[] */
    int *left = (int *)malloc(sizeof(int)*n);
    int *right = (int *)malloc(sizeof(int)*n);

    /* Allocate memory for the product array */
    int *prod = (int *)malloc(sizeof(int)*n);

    int i, j;

    /* Left most element of left array is always 1 */
    left[0] = 1;

    /* Rightmost most element of right array is always 1 */
    right[n-1] = 1;

    /* Construct the left array */
    for(i = 1; i < n; i++)
        left[i] = arr[i-1]*left[i-1];

    /* Construct the right array */
    for(j = n-2; j >=0; j--)
        right[j] = arr[j+1]*right[j+1];

    /* Construct the product array using
       left[] and right[] */
    for (i=0; i<n; i++)
        prod[i] = left[i] * right[i];

    /* print the constructed prod array */
    for (i=0; i<n; i++)
        printf("%d ", prod[i]);

    return;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {10, 3, 5, 6, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("The product array is: \n");
    productArray(arr, n);
    getchar();
}
```

Time Complexity: O(n)

Space Complexity: O(n)

Auxiliary Space: O(n)

The above method can be optimized to work in space complexity O(1). Thanks to Dileep for suggesting the below solution.

```
void productArray(int arr[], int n)
{
    int i, temp = 1;
```

```

/* Allocate memory for the product array */
int *prod = (int *)malloc(sizeof(int)*n);

/* Initialize the product array as 1 */
memset(prod, 1, n);

/* In this loop, temp variable contains product of
elements on left side excluding arr[i] */
for(i=0; i<n; i++)
{
    prod[i] = temp;
    temp *= arr[i];
}

/* Initialize temp to 1 for product on right side */
temp = 1;

/* In this loop, temp variable contains product of
elements on right side excluding arr[i] */
for(i= n-1; i>=0; i--)
{
    prod[i] *= temp;
    temp *= arr[i];
}

/* print the constructed prod array */
for (i=0; i<n; i++)
    printf("%d ", prod[i]);

return;
}

```

Time Complexity: O(n)

Space Complexity: O(n)

Auxiliary Space: O(1)

Segregate Even and Odd numbers

Given an array A[], write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers.

Example

```
Input = {12, 34, 45, 9, 8, 90, 3}
Output = {12, 34, 8, 90, 45, 9, 3}
```

In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.

The problem is very similar to our old post [Segregate 0s and 1s in an array](#), and both of these problems are variation of famous [Dutch national flag problem](#).

Algorithm: segregateEvenOdd()

- 1) Initialize two index variables left and right:
left = 0, right = size -1
- 2) Keep incrementing left index until we see an odd number.
- 3) Keep decrementing right index until we see an even number.
- 4) If left < right then swap arr[left] and arr[right]

Implementation:

C/C++

```
// C program to segregate even and odd elements of array
#include<stdio.h>

/* Function to swap *a and *b */
void swap(int *a, int *b);

void segregateEvenOdd(int arr[], int size)
{
    /* Initialize left and right indexes */
    int left = 0, right = size-1;
    while (left < right)
    {
        /* Increment left index while we see 0 at left */
        while (arr[left]%2 == 0 && left < right)
            left++;

        /* Decrement right index while we see 1 at right */
        while (arr[right]%2 == 1 && left < right)
            right--;

        if (left < right)
        {
            /* Swap arr[left] and arr[right]*/
            swap(&arr[left], &arr[right]);
            left++;
            right--;
        }
    }
}

/* UTILITY FUNCTIONS */
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

/* driver program to test */
int main()
{
    int arr[] = {12, 34, 45, 9, 8, 90, 3};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    int i = 0;

    segregateEvenOdd(arr, arr_size);

    printf("Array after segregation ");
    for (i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

```
}
```

Java

```
// Java program to segregate even and odd elements of array
import java.io.*;

class SegregateOddEven
{
    static void segregateEvenOdd(int arr[])
    {
        /* Initialize left and right indexes */
        int left = 0, right = arr.length - 1;
        while (left < right)
        {
            /* Increment left index while we see 0 at left */
            while (arr[left] % 2 == 0 && left < right)
                left++;

            /* Decrement right index while we see 1 at right */
            while (arr[right] % 2 == 1 && left < right)
                right--;

            if (left < right)
            {
                /* Swap arr[left] and arr[right]*/
                int temp = arr[left];
                arr[left] = arr[right];
                arr[right] = temp;
                left++;
                right--;
            }
        }
    }

    /* Driver program to test above functions */
    public static void main (String[] args)
    {
        int arr[] = {12, 34, 45, 9, 8, 90, 3};

        segregateEvenOdd(arr);

        System.out.print("Array after segregation ");
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}
/*This code is contributed by Devesh Agrawal*/
```

Python

```
# Python program to segregate even and odd elements of array

def segregateEvenOdd(arr):

    # Initialize left and right indexes
    left, right = 0, len(arr) - 1

    while left < right:

        # Increment left index while we see 0 at left
        while (arr[left] % 2 == 0 and left < right):
            left += 1

        # Decrement right index while we see 1 at right
        while (arr[right] % 2 == 1 and left < right):
            right -= 1

        if (left < right):
            # Swap arr[left] and arr[right]*/
            arr[left], arr[right] = arr[right], arr[left]
            left += 1
            right = right - 1

# Driver function to test above function
arr = [12, 34, 45, 9, 8, 90, 3]
segregateEvenOdd(arr)
print ("Array after segregation "),
```

```
for i in range(0,len(arr)):  
    print arr[i],  
# This code is contributed by Devesh Agrawal
```

Array after segregation 12 34 90 8 9 45 3

Time Complexity: O(n)

References:

<http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Flag/>

Find the two repeating elements in a given array

You are given an array of $n+2$ elements. All elements of the array are in range 1 to n . And all elements occur once except two numbers which occur twice. Find the two repeating numbers.

For example, array = {4, 2, 4, 5, 2, 3, 1} and $n = 5$

The above array has $n + 2 = 7$ elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

Method 1 (Basic)

Use two loops. In the outer loop, pick elements one by one and count the number of occurrences of the picked element in the inner loop.

This method doesn't use the other useful data provided in questions like range of numbers is between 1 to n and there are only two repeating elements.

```
#include<stdio.h>
#include<stdlib.h>
void printRepeating(int arr[], int size)
{
    int i, j;
    printf(" Repeating elements are ");
    for(i = 0; i < size; i++)
        for(j = i+1; j < size; j++)
            if(arr[i] == arr[j])
                printf(" %d ", arr[i]);
}

int main()
{
    int arr[] = {4, 2, 4, 5, 2, 3, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Method 2 (Use Count array)

Traverse the array once. While traversing, keep track of count of all elements in the array using a temp array $count[]$ of size n , when you see an element whose count is already set, print it as duplicate.

This method uses the range given in the question to restrict the size of $count[]$, but doesn't use the data that there are only two repeating elements.

```
#include<stdio.h>
#include<stdlib.h>

void printRepeating(int arr[], int size)
{
    int *count = (int *)calloc(sizeof(int), (size - 2));
    int i;

    printf(" Repeating elements are ");
    for(i = 0; i < size; i++)
    {
        if(count[arr[i]] == 1)
            printf(" %d ", arr[i]);
        else
            count[arr[i]]++;
    }
}

int main()
{
    int arr[] = {4, 2, 4, 5, 2, 3, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$

Method 3 (Make two equations)

Let the numbers which are being repeated are X and Y. We make two equations for X and Y and the simple task left is to solve the two equations.

We know the sum of integers from 1 to n is $n(n+1)/2$ and product is $n!$. We calculate the sum of input array, when this sum is subtracted from $n(n+1)/2$, we get $X + Y$ because X and Y are the two numbers missing from set $[1..n]$. Similarly calculate product of input array, when this product is divided from $n!$, we get $X*Y$. Given sum and product of X and Y, we can find easily out X and Y.

Let summation of all numbers in array be S and product be P

$$X + Y = S - \frac{n(n+1)}{2}$$

$$XY = P/n!$$

Using above two equations, we can find out X and Y. For array = 4 2 4 5 2 3 1, we get S = 21 and P as 960.

$$X + Y = 21 - 6 = 15$$

$$XY = 960/5! = 8$$

$$X Y = \sqrt{(X+Y)^2 - 4*XY} = \sqrt{4} = 2$$

Using below two equations, we easily get $X = (6 + 2)/2$ and $Y = (6-2)/2$

$$X + Y = 6$$

$$X Y = 2$$

Thanks to [geek4u](#) for suggesting this method. As pointed by [Beginer](#), there can be addition and multiplication overflow problem with this approach.

The methods 3 and 4 use all useful information given in the question 😊

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

/* function to get factorial of n */
int fact(int n);

void printRepeating(int arr[], int size)
{
    int S = 0; /* S is for sum of elements in arr[] */
    int P = 1; /* P is for product of elements in arr[] */
    int x, y; /* x and y are two repeating elements */
    int D; /* D is for difference of x and y, i.e., x-y*/
    int n = size - 2, i;

    /* Calculate Sum and Product of all elements in arr[] */
    for(i = 0; i < size; i++)
    {
        S = S + arr[i];
        P = P*arr[i];
    }

    S = S - n*(n+1)/2; /* S is x + y now */
    P = P/fact(n); /* P is x*y now */

    D = sqrt(S*S - 4*P); /* D is x - y now */

    x = (D + S)/2;
    y = (S - D)/2;

    printf("The two Repeating elements are %d & %d", x, y);
}

int fact(int n)
{
    return (n == 0) ? 1 : n*fact(n-1);
}

int main()
{
    int arr[] = {4, 2, 4, 5, 2, 3, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Auxiliary Space: O(1)

Method 4 (Use XOR)

Thanks to neophyte for suggesting this method.

The approach used here is similar to method 2 of [this post](#).

Let the repeating numbers be X and Y, if we xor all the elements in the array and all integers from 1 to n, then the result is X xor Y.

The 1s in binary representation of X xor Y is corresponding to the different bits between X and Y. Suppose that the kth bit of X xor Y is 1, we can xor all the elements in the array and all integers from 1 to n, whose kth bits are 1. The result will be one of X and Y.

```
void printRepeating(int arr[], int size)
{
    int xor = arr[0]; /* Will hold xor of all elements */
    int set_bit_no; /* Will have only single set bit of xor */
    int i;
    int n = size - 2;
    int x = 0, y = 0;

    /* Get the xor of all elements in arr[] and {1, 2 .. n} */
    for(i = 1; i < size; i++)
        xor ^= arr[i];
    for(i = 1; i <= n; i++)
        xor ^= i;

    /* Get the rightmost set bit in set_bit_no */
    set_bit_no = xor & ~(xor-1);

    /* Now divide elements in two sets by comparing rightmost set
       bit of xor with bit at same position in each element. */
    for(i = 0; i < size; i++)
    {
        if(arr[i] & set_bit_no)
            x = x ^ arr[i]; /*XOR of first set in arr[] */
        else
            y = y ^ arr[i]; /*XOR of second set in arr[] */
    }
    for(i = 1; i <= n; i++)
    {
        if(i & set_bit_no)
            x = x ^ i; /*XOR of first set in arr[] and {1, 2, ...n }*/
        else
            y = y ^ i; /*XOR of second set in arr[] and {1, 2, ...n }*/
    }

    printf("\n The two repeating elements are %d & %d ", x, y);
}

int main()
{
    int arr[] = {4, 2, 4, 5, 2, 3, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

Method 5 (Use array elements as index)

Thanks to Manish K. Aasawat for suggesting this method.

Traverse the array. Do following for every index i of A[].

```
{
check for sign of A[abs(A[i])] ;
if positive then
    make it negative by A[abs(A[i])] = -A[abs(A[i])];
else // i.e., A[abs(A[i])] is negative
    this element (ith element of list) is a repetition
}
```

Example: A[] = {1, 1, 2, 3, 2}

i=0;

Check sign of A[abs(A[0])] which is A[1]. A[1] is positive, so make it negative.

Array now becomes {1, -1, 2, 3, 2}

i=1;

Check sign of A[abs(A[1])] which is A[1]. A[1] is negative, so A[1] is a repetition.

i=2;

Check sign of A[abs(A[2])] which is A[2]. A[2] is positive, so make it negative. '

Array now becomes {1, -1, -2, 3, 2}

i=3;

Check sign of A[abs(A[3])] which is A[3]. A[3] is positive, so make it negative.
Array now becomes {1, -1, -2, -3, 2}

i=4;
Check sign of A[abs(A[4])] which is A[2]. A[2] is negative, so A[4] is a repetition.

Note that this method modifies the original array and may not be a recommended method if we are not allowed to modify the array.

```
#include <stdio.h>
#include <stdlib.h>

void printRepeating(int arr[], int size)
{
    int i;

    printf("\n The repeating elements are");

    for(i = 0; i < size; i++)
    {
        if(arr[abs(arr[i])] > 0)
            arr[abs(arr[i])] = -arr[abs(arr[i])];
        else
            printf(" %d ", abs(arr[i]));
    }
}

int main()
{
    int arr[] = {1, 3, 2, 2, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

The problem can be solved in linear time using other method also, please see [this](#) and [this](#) comments

Sort an array of 0s, 1s and 2s

Given an array A[] consisting 0s, 1s and 2s, write a function that sorts A[]. The functions should put all 0s first, then all 1s and all 2s in last.

Example

Input = {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1};
Output = {0, 0, 0, 0, 1, 1, 1, 1, 2, 2}

The problem is similar to our old post [Segregate 0s and 1s in an array](#), and both of these problems are variation of famous [Dutch national flag problem](#).

The problem was posed with three colours, here '0?', '1?' and '2?'. The array is divided into four sections:

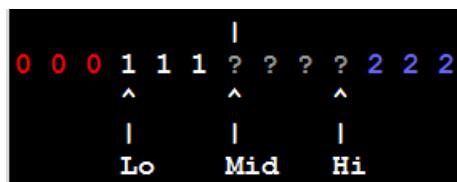
1. a[1..Lo-1] zeroes (red)
2. a[Lo..Mid-1] ones (white)
3. a[Mid..Hi] unknown
4. a[Hi+1..N] twos (blue)

The unknown region is shrunk while maintaining these conditions

1. Lo := 1; Mid := 1; Hi := N;
2. while Mid <= Hi do
 1. Invariant: a[1..Lo-1]=0 and a[Lo..Mid-1]=1 and a[Hi+1..N]=2; a[Mid..Hi] are unknown.
 2. case a[Mid] in
 - 0: swap a[Lo] and a[Mid]; Lo++; Mid++
 - 1: Mid++
 - 2: swap a[Mid] and a[Hi]; Hi

Dutch National Flag Algorithm, or 3-way Partitioning

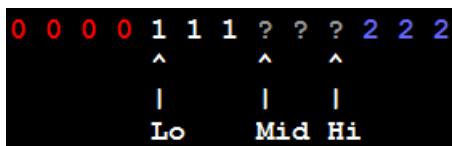
Part way through the process, some red, white and blue elements are known and are in the right place. The section of unknown elements, a[Mid..Hi], is shrunk by examining a[Mid]:



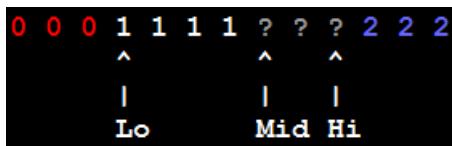
Examine a[Mid]. There are three possibilities:

a[Mid] is (0) red, (1) white or (2) blue.

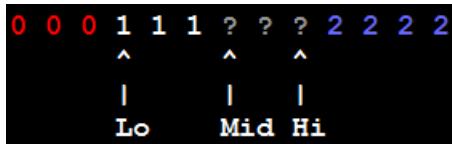
Case (0) a[Mid] is red, swap a[Lo] and a[Mid]; Lo++; Mid++



Case (1) a[Mid] is white, Mid++



Case (2) a[Mid] is blue, swap a[Mid] and a[Hi]; Hi--



Continue until Mid>Hi.

Below is C implementation of above algorithm.

C

```
// C program to sort an array with 0,1 and 2
// in a single pass
#include<stdio.h>

/* Function to swap *a and *b */
void swap(int *a, int *b);

// Sort the input array, the array is assumed to
// have values in {0, 1, 2}
void sort012(int a[], int arr_size)
{
    int lo = 0;
    int hi = arr_size - 1;
    int mid = 0;

    while (mid <= hi)
    {
        switch (a[mid])
        {
        case 0:
            swap(&a[lo++], &a[mid++]);
            break;
        case 1:
            mid++;
            break;
        case 2:
            swap(&a[mid], &a[hi--]);
            break;
        }
    }
}

/* UTILITY FUNCTIONS */
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

/* Utility function to print array arr[] */
void printArray(int arr[], int arr_size)
{
    int i;
    for (i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* driver program to test */
int main()
{
    int arr[] = {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    int i;

    sort012(arr, arr_size);

    printf("array after segregation ");
    printArray(arr, arr_size);

    getchar();
    return 0;
}
```

Java

```
// Java program to sort an array of 0, 1 and 2
import java.io.*;

class countzot {

    // Sort the input array, the array is assumed to
    // have values in {0, 1, 2}
    static void sort012(int a[], int arr_size)
    {
        int lo = 0;
```

```

int hi = arr_size - 1;
int mid = 0,temp=0;
while (mid <= hi)
{
    switch (a[mid])
    {
        case 0:
        {
            temp = a[lo];
            a[lo] = a[mid];
            a[mid] = temp;
            lo++;
            mid++;
            break;
        }
        case 1:
            mid++;
            break;
        case 2:
        {
            temp = a[mid];
            a[mid] = a[hi];
            a[hi] = temp;
            hi--;
            break;
        }
    }
}
/* Utility function to print array arr[] */
static void printArray(int arr[], int arr_size)
{
    int i;
    for (i = 0; i < arr_size; i++)
        System.out.print(arr[i]+" ");
    System.out.println("");
}
/*Driver function to check for above functions*/
public static void main (String[] args)
{
    int arr[] = {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1};
    int arr_size = arr.length;
    sort012(arr, arr_size);
    System.out.println("Array after segregation ");
    printArray(arr, arr_size);
}
}
/*This code is contributed by Devesh Agrawal*/

```

Output:

array after segregation 0 0 0 0 1 1 1 1 1 2 2

Time Complexity: O(n)

The above code performs unnecessary swaps for inputs like 0 0 0 0 1 1 1 2 2 2 2 : lo=4 and mid=7 and hi=11. In present code: first 7 exchanged with 11 and hi become 10 and mid is still pointing to 7. again same operation is till the mid <= hi. But it is really not required. We can change the swap function to do a check that the values being swapped are same or not, if not same, then only swap values. Thanks to Ankur Roy for suggesting this optimization. Source: <http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Flag/>

Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted

Given an unsorted array arr[0..n-1] of size n, find the minimum length subarray arr[s..e] such that sorting this subarray makes the whole array sorted.

Examples:

1) If the input array is [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60], your program should be able to find that the subarray lies between the indexes 3 and 8.

2) If the input array is [0, 1, 15, 25, 6, 7, 30, 40, 50], your program should be able to find that the subarray lies between the indexes 2 and 5.

Solution:

1) Find the candidate unsorted subarray

a) Scan from left to right and find the first element which is greater than the next element. Let s be the index of such an element. In the above example 1, s is 3 (index of 30).

b) Scan from right to left and find the first element (first in right to left order) which is smaller than the next element (next in right to left order). Let e be the index of such an element. In the above example 1, e is 7 (index of 31).

2) Check whether sorting the candidate unsorted subarray makes the complete array sorted or not. If not, then include more elements in the subarray.

a) Find the minimum and maximum values in $\text{arr}[s..e]$. Let minimum and maximum values be min and max . min and max for [30, 25, 40, 32, 31] are 25 and 40 respectively.

b) Find the first element (if there is any) in $\text{arr}[0..s-1]$ which is greater than min , change s to index of this element. There is no such element in above example 1.

c) Find the last element (if there is any) in $\text{arr}[e+1..n-1]$ which is smaller than max , change e to index of this element. In the above example 1, e is changed to 8 (index of 35)

3) Print s and e .

Implementation:

```
#include<stdio.h>

void printUnsorted(int arr[], int n)
{
    int s = 0, e = n-1, i, max, min;

    // step 1(a) of above algo
    for (s = 0; s < n-1; s++)
    {
        if (arr[s] > arr[s+1])
            break;
    }
    if (s == n-1)
    {
        printf("The complete array is sorted");
        return;
    }

    // step 1(b) of above algo
    for(e = n - 1; e > 0; e--)
    {
        if(arr[e] < arr[e-1])
            break;
    }

    // step 2(a) of above algo
    max = arr[s]; min = arr[s];
    for(i = s + 1; i <= e; i++)
    {
        if(arr[i] > max)
            max = arr[i];
        if(arr[i] < min)
            min = arr[i];
    }

    // step 2(b) of above algo
    for( i = 0; i < s; i++)
    {
        if(arr[i] > min)
```

```

    {
        s = i;
        break;
    }
}

// step 2(c) of above algo
for( i = n -1; i >= e+1; i--)
{
    if(arr[i] < max)
    {
        e = i;
        break;
    }
}

// step 3 of above algo
printf(" The unsorted subarray which makes the given array "
       " sorted lies between the indees %d and %d", s, e);
return;
}

int main()
{
    int arr[] = {10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printUnsorted(arr, arr_size);
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Find duplicates in O(n) time and O(1) extra space

Given an array of n elements which contains elements from 0 to n-1, with any of these numbers appearing any number of times. Find these repeating numbers in O(n) and using only constant memory space.

For example, let n be 7 and array be {1, 2, 3, 1, 3, 6, 6}, the answer should be 1, 3 and 6.

This problem is an extended version of following problem.

[Find the two repeating elements in a given array](#)

Method 1 and Method 2 of the above link are not applicable as the question says O(n) time complexity and O(1) constant space. Also, Method 3 and Method 4 cannot be applied here because there can be more than 2 repeating elements in this problem. Method 5 can be extended to work for this problem. Below is the solution that is similar to the Method 5.

Algorithm:

```
traverse the list for i= 0 to n-1 elements
{
    check for sign of A[abs(A[i])] ;
    if positive then
        make it negative by A[abs(A[i])] = -A[abs(A[i])];
    else // i.e., A[abs(A[i])] is negative
        this element (ith element of list) is a repetition
}
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

void printRepeating(int arr[], int size)
{
    int i;
    printf("The repeating elements are: \n");
    for (i = 0; i < size; i++)
    {
        if (arr[abs(arr[i])] >= 0)
            arr[abs(arr[i])] = -arr[abs(arr[i])];
        else
            printf(" %d ", abs(arr[i]));
    }
}

int main()
{
    int arr[] = {1, 2, 3, 1, 3, 6, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

Note: The above program doesn't handle 0 case (If 0 is present in array). The program can be easily modified to handle that also. It is not handled to keep the code simple.

Output:

The repeating elements are:

1 3 6

Time Complexity: O(n)

Auxiliary Space: O(1)

Equilibrium index of an array

Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in an array A:

A[0] = -7, A[1] = 1, A[2] = 5, A[3] = 2, A[4] = -4, A[5] = 3, A[6] = 0

3 is an equilibrium index, because:

A[0] + A[1] + A[2] = A[4] + A[5] + A[6]

6 is also an equilibrium index, because sum of zero elements is zero, i.e., A[0] + A[1] + A[2] + A[3] + A[4] + A[5] = 0

7 is not an equilibrium index, because it is not a valid index of array A.

Write a function `int equilibrium(int[] arr, int n)`; that given a sequence arr[] of size n, returns an equilibrium index (if any) or -1 if no equilibrium indexes exist.

Method 1 (Simple but inefficient)

Use two loops. Outer loop iterates through all the elements and inner loop finds out whether the current index picked by the outer loop is equilibrium index or not. Time complexity of this solution is O(n^2).

```
#include <stdio.h>

int equilibrium(int arr[], int n)
{
    int i, j;
    int leftsum, rightsum;

    /* Check for indexes one by one until an equilibrium
       index is found */
    for (i = 0; i < n; ++i)
    {
        leftsum = 0; // initialize left sum for current index i
        rightsum = 0; // initialize right sum for current index i

        /* get left sum */
        for (j = 0; j < i; j++)
            leftsum += arr[j];

        /* get right sum */
        for (j = i+1; j < n; j++)
            rightsum += arr[j];

        /* if leftsum and rightsum are same, then we are done */
        if (leftsum == rightsum)
            return i;
    }

    /* return -1 if no equilibrium index is found */
    return -1;
}

int main()
{
    int arr[] = {-7, 1, 5, 2, -4, 3, 0};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printf("%d\n", equilibrium(arr, arr_size));

    getchar();
    return 0;
}
```

Time Complexity: O(n^2)

Method 2 (Tricky and Efficient)

The idea is to get total sum of array first. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get right sum by subtracting the elements one by one. Thanks to Sambasiva for suggesting this solution and providing code for this.

- 1) Initialize leftsum as 0
- 2) Get the total sum of the array as sum
- 3) Iterate through the array and for each index i, do following.
 - a) Update sum to get the right sum.
 $sum = sum - arr[i]$

```

    // sum is now right sum
    b) If leftsum is equal to sum, then return current index.
    c) leftsum = leftsum + arr[i] // update leftsum for next iteration.
4) return -1 // If we come out of loop without returning then
               // there is no equilibrium index

#include <stdio.h>

int equilibrium(int arr[], int n)
{
    int sum = 0;           // initialize sum of whole array
    int leftsum = 0; // initialize leftsum
    int i;

    /* Find sum of the whole array */
    for (i = 0; i < n; ++i)
        sum += arr[i];

    for( i = 0; i < n; ++i)
    {
        sum -= arr[i]; // sum is now right sum for index i

        if(leftsum == sum)
            return i;

        leftsum += arr[i];
    }

    /* If no equilibrium index found, then return 0 */
    return -1;
}

int main()
{
    int arr[] = {-7, 1, 5, 2, -4, 3, 0};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printf("First equilibrium index is %d\n", equilibrium(arr, arr_size));

    getchar();
    return 0;
}

```

Time Complexity: O(n)

As pointed out by Sameer, we can remove the return statement and add a print statement to print all equilibrium indexes instead of returning only one.

Linked List vs Array

Difficulty Level: Rookie

Both Arrays and [Linked List](#) can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Linked lists have following drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Arrays have better cache locality that can make a pretty big difference in performance.

Please also see [this](#) thread.

References:

<http://cslibrary.stanford.edu/103/LinkedListBasics.pdf>

Which sorting algorithm makes minimum number of memory writes?

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with [EEPROMs](#) or [Flash memory](#), where each write reduces the lifespan of the memory.

Among the sorting algorithms that we generally study in our data structure and algorithm courses, [Selection Sort](#) makes least number of writes (it makes $O(n)$ swaps). But, [Cycle Sort](#) almost always makes less number of writes compared to Selection Sort. In Cycle Sort, each value is either written zero times, if its already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

Sources:

http://en.wikipedia.org/wiki/Cycle_sort

http://en.wikipedia.org/wiki/Selection_sort

Turn an image by 90 degree

Given an image, how will you turn it by 90 degrees? A vague question. Minimize the browser and try your solution before going further.

An image can be treated as 2D matrix which can be stored in a buffer. We are provided with matrix dimensions and its base address. How can we turn it?

For example see the below picture,

```
* * * ^ * * *
* * * | * * *
* * * | * * *
* * * | * * *
```

After rotating right, it appears (observe arrow direction)

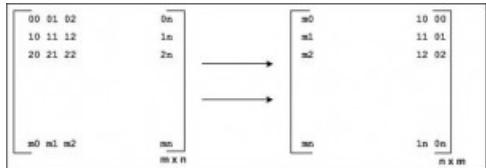
```
* * * *
* * * *
* * * *
-- - - >
* * * *
* * * *
* * * *
```

The idea is simple. Transform each row of source matrix into required column of final image. We will use an auxiliary buffer to transform the image.

From the above picture, we can observe that

```
first row of source -----> last column of destination
second row of source -----> last but-one column of destination
so ... on
last row of source-----> first column of destination
```

In pictorial form, we can represent the above transformations of an ($m \times n$) matrix into ($n \times m$) matrix,



Transformations

If you have not attempted, atleast try your pseudo code now.

It will be easy to write our pseudo code. In C/C++ we will usually traverse matrix on row major order. Each row is transformed into different column of final image. We need to construct columns of final image. See the following algorithm (transformation)

```
for(r = 0; r < m; r++)
{
    for(c = 0; c < n; c++)
    {
        // Hint: Map each source element indices into
        // indices of destination matrix element.
        dest_buffer [ c ] [ m - r - 1 ] = source_buffer [ r ] [ c ];
    }
}
```

Note that there are various ways to implement the algorithm based on traversal of matrix, row major or column major order. We have two matrices and two ways (row and column major) to traverse each matrix. Hence, there can atleast be 4 different ways of transformation of source matrix into final matrix.

Code:

```
#include <stdio.h>
#include <stdlib.h>

void displayMatrix(unsigned int const *p, unsigned int row, unsigned int col);
void rotate(unsigned int *pS, unsigned int *pD, unsigned int row, unsigned int col);

int main()
{
    // declarations
    unsigned int image[][][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
    unsigned int *pSource;
    unsigned int *pDestination;
```

```

unsigned int m, n;

// setting initial values and memory allocation
m = 3, n = 4, pSource = (unsigned int *)image;
pDestination = (unsigned int *)malloc(sizeof(int)*m*n);

// process each buffer
displayMatrix(pSource, m, n);

rotate(pSource, pDestination, m, n);

displayMatrix(pDestination, n, m);

free(pDestination);

getchar();
return 0;
}

void displayMatrix(unsigned int const *p, unsigned int r, unsigned int c)
{
    unsigned int row, col;
    printf("\n\n");

    for(row = 0; row < r; row++)
    {
        for(col = 0; col < c; col++)
        {
            printf("%d\t", *(p + row * c + col));
        }
        printf("\n");
    }

    printf("\n\n");
}

void rotate(unsigned int *pS, unsigned int *pD, unsigned int row, unsigned int col)
{
    unsigned int r, c;
    for(r = 0; r < row; r++)
    {
        for(c = 0; c < col; c++)
        {
            *(pD + c * row + (row - r - 1)) = *(pS + r * col + c);
        }
    }
}

```

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

- a) For any array, rightmost element always has next greater element as -1.
- b) For an array which is sorted in decreasing order, all elements have next greater element as -1.
- c) For the input array {4, 5, 2, 25}, the next greater elements for each element are as follows.

| Element | NGE |
|---------|--------|
| 4 | --> 5 |
| 5 | --> 25 |
| 2 | --> 25 |
| 25 | --> -1 |

- d) For the input array {13, 7, 6, 12}, the next greater elements for each element are as follows.

| Element | NGE |
|---------|--------|
| 13 | --> -1 |
| 7 | --> 12 |
| 6 | --> 12 |
| 12 | --> -1 |

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Thanks to [Sachin](#) for providing following code.

C/C++

```
// Simple C program to print next greater elements
// in a given array
#include<stdio.h>

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -- %d\n", arr[i], next);
    }
}

int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Python

```
# Function to print element and NGE pair for all elements of list
def printNGE(arr):

    for i in range(0, len(arr), 1):

        next = -1
        for j in range(i+1, len(arr), 1):
```

```

if arr[i] < arr[j]:
    next = arr[j]
    break

print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function
arr = [11,13,21,3]
printNGE(arr)

# This code is contributed by Sunny Karira

```

11 -- 13
 13 -- 21
 21 -- -1
 3 -- -1

Time Complexity: $O(n^2)$. The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

Thanks to [pcchild](#) for suggesting following approach.

- 1) Push the first element to stack.
- 2) Pick rest of the elements one by one and follow following steps in loop.
 - a) Mark the current element as *next*.
 - b) If stack is not empty, then pop an element from stack and compare it with *next*.
 - c) If *next* is greater than the popped element, then *next* is the next greater element for the popped element.
 - d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements
 - g) If *next* is smaller than the popped element, then push the popped element back.
- 3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

Check if array elements are consecutive | Added Method 3

Given an unsorted array of numbers, write a function that returns true if array consists of consecutive numbers.

Examples:

- a) If array is {5, 2, 3, 1, 4}, then the function should return true because the array has consecutive numbers from 1 to 5.
- b) If array is {83, 78, 80, 81, 79, 82}, then the function should return true because the array has consecutive numbers from 78 to 83.
- c) If the array is {34, 23, 52, 12, 3 }, then the function should return false because the elements are not consecutive.
- d) If the array is {7, 6, 5, 5, 3, 4}, then the function should return false because 5 and 5 are not consecutive.

Method 1 (Use Sorting)

1) Sort all the elements.

2) Do a linear scan of the sorted array. If the difference between current element and next element is anything other than 1, then return false. If all differences are 1, then return true.

Time Complexity: O(nLogn)

Method 2 (Use visited array)

The idea is to check for following two conditions. If following two conditions are true, then return true.

- 1) $\max \min + 1 = n$ where max is the maximum element in array, min is minimum element in array and n is the number of elements in array.
- 2) All elements are distinct.

To check if all elements are distinct, we can create a visited[] array of size n. We can map the ith element of input array arr[] to visited array by using arr[i] min as index in visited[].

```
#include<stdio.h>
#include<stdlib.h>

/* Helper functions to get minimum and maximum in an array */
int getMin(int arr[], int n);
int getMax(int arr[], int n);

/* The function checks if the array elements are consecutive
   If elements are consecutive, then returns true, else returns
   false */
bool areConsecutive(int arr[], int n)
{
    if ( n < 1 )
        return false;

    /* 1) Get the minimum element in array */
    int min = getMin(arr, n);

    /* 2) Get the maximum element in array */
    int max = getMax(arr, n);

    /* 3) max - min + 1 is equal to n, then only check all elements */
    if (max - min + 1 == n)
    {
        /* Create a temp array to hold visited flag of all elements.
           Note that, calloc is used here so that all values are initialized
           as false */
        bool *visited = (bool *) calloc (n, sizeof(bool));
        int i;
        for (i = 0; i < n; i++)
        {
            /* If we see an element again, then return false */
            if ( visited[arr[i] - min] != false )
                return false;

            /* If visited first time, then mark the element as visited */
            visited[arr[i] - min] = true;
        }

        /* If all elements occur once, then return true */
        return true;
    }

    return false; // if (max - min + 1 != n)
}

/* UTILITY FUNCTIONS */
int getMin(int arr[], int n)
```

```

{
    int min = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] < min)
            min = arr[i];
    return min;
}

int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {5, 4, 2, 3, 1, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if(areConsecutive(arr, n) == true)
        printf(" Array elements are consecutive ");
    else
        printf(" Array elements are not consecutive ");
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Extra Space: O(n)

Method 3 (Mark visited array elements as negative)

This method is O(n) time complexity and O(1) extra space, but it changes the original array and it works only if all numbers are positive. We can get the original array by adding an extra step though. It is an extension of method 2 and it has the same two steps.

- 1) $\max \min + 1 = n$ where max is the maximum element in array, min is minimum element in array and n is the number of elements in array.
- 2) All elements are distinct.

In this method, the implementation of step 2 differs from method 2. Instead of creating a new array, we modify the input array arr[] to keep track of visited elements. The idea is to traverse the array and for each index i (where $0 \leq i < n$), make $\text{arr}[\text{arr}[i] - \min]$ as a negative value. If we see a negative value again then there is repetition.

```

#include<stdio.h>
#include<stdlib.h>

/* Helper functions to get minimum and maximum in an array */
int getMin(int arr[], int n);
int getMax(int arr[], int n);

/* The function checks if the array elements are consecutive
   If elements are consecutive, then returns true, else returns
   false */
bool areConsecutive(int arr[], int n)
{

    if (n < 1)
        return false;

    /* 1) Get the minimum element in array */
    int min = getMin(arr, n);

    /* 2) Get the maximum element in array */
    int max = getMax(arr, n);

    /* 3) max - min + 1 is equal to n then only check all elements */
    if (max - min + 1 == n)
    {
        int i;
        for(i = 0; i < n; i++)
        {
            int j;

            if (arr[i] < 0)
                j = -arr[i] - min;
            else
                j = arr[i] - min;

            // if the value at index j is negative then

```

```

        // there is repetition
        if (arr[j] > 0)
            arr[j] = -arr[j];
        else
            return false;
    }

    /* If we do not see a negative value then all elements
     * are distinct */
    return true;
}

return false; // if (max - min + 1 != n)
}

/* UTILITY FUNCTIONS */
int getMin(int arr[], int n)
{
    int min = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] < min)
            min = arr[i];
    return min;
}

int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 4, 5, 3, 2, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if(areConsecutive(arr, n) == true)
        printf(" Array elements are consecutive ");
    else
        printf(" Array elements are not consecutive ");
    getchar();
    return 0;
}

```

Note that this method might not work for negative numbers. For example, it returns false for {2, 1, 0, -3, -1, -2}.

Time Complexity: O(n)

Extra Space: O(1)

Source: <http://geeksforgeeks.org/forum/topic/amazon-interview-question-for-software-engineerdeveloper-fresher-9>

Find the smallest missing number

Given a **sorted** array of n integers where each integer is in the range from 0 to m-1 and m>n. Find the smallest number that is missing from the array.

Examples

Input: {0, 1, 2, 6, 9}, n=5, m=10

Output: 3

Input: {4, 5, 10, 11}, n=4, m=12

Output: 0

Input: {0, 1, 2, 3}, n=4, m=5

Output: 4

Input: {0, 1, 2, 3, 4, 5, 6, 7, 10}, n=9, m=11

Output: 8

Thanks to [Ravichandra](#) for suggesting following two methods.

Method 1 (Use Binary Search)

For i=0 to m-1, do binary search for i in the array. If i is not present in the array then return i.

Time Complexity: O(m log n)

Method 2 (Linear Search)

If arr[0] is not 0, return 0. Otherwise traverse the input array starting from index 1, and for each pair of elements a[i] and a[i+1], find the difference between them. if the difference is greater than 1 then a[i]+1 is the missing number.

Time Complexity: O(n)

Method 3 (Use Modified Binary Search)

Thanks to [yasein](#) and [Jams](#) for suggesting this method.

In the standard Binary Search process, the element to be searched is compared with the middle element and on the basis of comparison result, we decide whether to search is over or to go to left half or right half.

In this method, we modify the standard Binary Search algorithm to compare the middle element with its index and make decision on the basis of this comparison.

- 1) If the first element is not same as its index then return first index
- 2) Else get the middle index say mid
- a) If arr[mid] greater than mid then the required element lies in left half.
- b) Else the required element lies in right half.

```
#include<stdio.h>

int findFirstMissing(int array[], int start, int end) {

    if(start > end)
        return end + 1;

    if (start != array[start])
        return start;

    int mid = (start + end) / 2;

    if (array[mid] > mid)
        return findFirstMissing(array, start, mid);
    else
        return findFirstMissing(array, mid + 1, end);
}

// driver program to test above function
int main()
{
    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf(" First missing element is %d",
          findFirstMissing(arr, 0, n-1));
    getchar();
    return 0;
}
```

Note: This method doesn't work if there are duplicate elements in the array.

Time Complexity: O(Logn)

Source: <http://geeksforgeeks.org/forum/topic/comnvault-interview-question-for-software-engineerdeveloper-2-5-years-about-algorithms>

Count the number of occurrences in a sorted array

Given a sorted array arr[] and a number x, write a function that counts the occurrences of x in arr[]. Expected time complexity is O(Logn)

Examples:

```
Input: arr[] = {1, 1, 2, 2, 2, 2, 3}, x = 2
Output: 4 // x (or 2) occurs 4 times in arr[]
```

```
Input: arr[] = {1, 1, 2, 2, 2, 2, 3}, x = 3
Output: 1
```

```
Input: arr[] = {1, 1, 2, 2, 2, 2, 3}, x = 1
Output: 2
```

```
Input: arr[] = {1, 1, 2, 2, 2, 2, 3}, x = 4
Output: -1 // 4 doesn't occur in arr[]
```

Method 1 (Linear Search)

Linearly search for x, count the occurrences of x and return the count.

Time Complexity: O(n)

Method 2 (Use Binary Search)

- 1) Use Binary search to get index of the first occurrence of x in arr[]. Let the index of the first occurrence be i.
- 2) Use Binary search to get index of the last occurrence of x in arr[]. Let the index of the last occurrence be j.
- 3) Return (j-i+1);

```
/* if x is present in arr[] then returns the count of occurrences of x,
   otherwise returns -1. */
int count(int arr[], int x, int n)
{
    int i; // index of first occurrence of x in arr[0..n-1]
    int j; // index of last occurrence of x in arr[0..n-1]

    /* get the index of first occurrence of x */
    i = first(arr, 0, n-1, x, n);

    /* If x doesn't exist in arr[] then return -1 */
    if(i == -1)
        return i;

    /* Else get the index of last occurrence of x. Note that we
       are only looking in the subarray after first occurrence */
    j = last(arr, i, n-1, x, n);

    /* return count */
    return j-i+1;
}

/* if x is present in arr[] then returns the index of FIRST occurrence
   of x in arr[0..n-1], otherwise returns -1 */
int first(int arr[], int low, int high, int x, int n)
{
    if(high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/
        if( (mid == 0 || x > arr[mid-1]) && arr[mid] == x)
            return mid;
        else if(x > arr[mid])
            return first(arr, (mid + 1), high, x, n);
        else
            return first(arr, low, (mid - 1), x, n);
    }
    return -1;
}

/* if x is present in arr[] then returns the index of LAST occurrence
   of x in arr[0..n-1], otherwise returns -1 */
int last(int arr[], int low, int high, int x, int n)
{
    if(high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/
        if( (mid == n-1 || x < arr[mid+1]) && arr[mid] == x )
            return mid;
        else if(x < arr[mid])

```

```
        return last(arr, low, (mid -1), x, n);
    else
        return last(arr, (mid + 1), high, x, n);
}
return -1;
}

/* driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 2, 3, 3, 3, 3};
    int x = 3; // Element to be counted in arr[]
    int n = sizeof(arr)/sizeof(arr[0]);
    int c = count(arr, x, n);
    printf("%d occurs %d times ", x, c);
    getchar();
    return 0;
}
```

Time Complexity: O(Logn)

Programming Paradigm: Divide & Conquer

Interpolation search vs Binary search

[Interpolation search](#) works better than Binary Search for a sorted and uniformly distributed array.

On average the interpolation search makes about $\log(\log(n))$ comparisons (if the elements are uniformly distributed), where n is the number of elements to be searched. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to $O(n)$ comparisons.

Sources:

http://en.wikipedia.org/wiki/Interpolation_search

Given an array arr[], find the maximum j i such that arr[j] > arr[i]

Given an array arr[], find the maximum j i such that arr[j] > arr[i].

Examples:

Input: {34, 8, 10, 3, 2, 80, 30, 33, 1}
Output: 6 (j = 7, i = 1)

Input: {9, 2, 3, 4, 5, 6, 7, 8, 18, 0}
Output: 8 (j = 8, i = 0)

Input: {1, 2, 3, 4, 5, 6}
Output: 5 (j = 5, i = 0)

Input: {6, 5, 4, 3, 2, 1}
Output: -1

Method 1 (Simple but Inefficient)

Run two loops. In the outer loop, pick elements one by one from left. In the inner loop, compare the picked element with the elements starting from right side. Stop the inner loop when you see an element greater than the picked element and keep updating the maximum j-i so far.

```
#include <stdio.h>
/* For a given array arr[], returns the maximum j - i such that
   arr[j] > arr[i] */
int maxIndexDiff(int arr[], int n)
{
    int maxDiff = -1;
    int i, j;

    for (i = 0; i < n; ++i)
    {
        for (j = n-1; j > i; --j)
        {
            if(arr[j] > arr[i] && maxDiff < (j - i))
                maxDiff = j - i;
        }
    }

    return maxDiff;
}

int main()
{
    int arr[] = {9, 2, 3, 4, 5, 6, 7, 8, 18, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int maxDiff = maxIndexDiff(arr, n);
    printf("\n %d", maxDiff);
    getchar();
    return 0;
}
```

Time Complexity: O(n^2)

Method 2 (Efficient)

To solve this problem, we need to get two optimum indexes of arr[]: left index i and right index j. For an element arr[i], we do not need to consider arr[i] for left index if there is an element smaller than arr[i] on left side of arr[i]. Similarly, if there is a greater element on right side of arr[j] then we do not need to consider this j for right index. So we construct two auxiliary arrays LMin[] and RMax[] such that LMin[i] holds the smallest element on left side of arr[i] including arr[i], and RMax[j] holds the greatest element on right side of arr[j] including arr[j]. After constructing these two auxiliary arrays, we traverse both of these arrays from left to right. While traversing LMin[] and RMax[] if we see that LMin[i] is greater than RMax[j], then we must move ahead in LMin[] (or do i++) because all elements on left of LMin[i] are greater than or equal to LMin[i]. Otherwise we must move ahead in RMax[j] to look for a greater j i value.

Thanks to celicom for suggesting the algorithm for this method.

```
#include <stdio.h>

/* Utility Functions to get max and minimum of two integers */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x < y? x : y;
}
```

```

/* For a given array arr[], returns the maximum j - i such that
   arr[j] > arr[i] */
int maxIndexDiff(int arr[], int n)
{
    int maxDiff;
    int i, j;

    int *LMin = (int *)malloc(sizeof(int)*n);
    int *RMax = (int *)malloc(sizeof(int)*n);

    /* Construct LMin[] such that LMin[i] stores the minimum value
       from (arr[0], arr[1], ... arr[i]) */
    LMin[0] = arr[0];
    for (i = 1; i < n; ++i)
        LMin[i] = min(arr[i], LMin[i-1]);

    /* Construct RMax[] such that RMax[j] stores the maximum value
       from (arr[j], arr[j+1], ..arr[n-1]) */
    RMax[n-1] = arr[n-1];
    for (j = n-2; j >= 0; --j)
        RMax[j] = max(arr[j], RMax[j+1]);

    /* Traverse both arrays from left to right to find optimum j - i
       This process is similar to merge() of MergeSort */
    i = 0, j = 0, maxDiff = -1;
    while (j < n && i < n)
    {
        if (LMin[i] < RMax[j])
        {
            maxDiff = max(maxDiff, j-i);
            j = j + 1;
        }
        else
            i = i+1;
    }

    return maxDiff;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {9, 2, 3, 4, 5, 6, 7, 8, 18, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int maxDiff = maxIndexDiff(arr, n);
    printf("\n %d", maxDiff);
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Auxiliary Space: O(n)

Maximum of all subarrays of size k (Added a O(n) method)

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Examples:

Input :

arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}

k = 3

Output :

3 3 4 5 5 6

Input :

arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}

k = 4

Output :

10 10 10 15 15 90 90

Method 1 (Simple)

Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.

```
#include<stdio.h>

void printKMax(int arr[], int n, int k)
{
    int j, max;

    for (int i = 0; i <= n-k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i+j] > max)
                max = arr[i+j];
        }
        printf("%d ", max);
    }
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Time Complexity: The outer loop runs $n-k+1$ times and the inner loop runs k times for every iteration of outer loop. So time complexity is $O((n-k+1)*k)$ which can also be written as $O(nk)$.

Method 2 (Use Self-Balancing BST)

1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.

2) Run a loop for $i = 0$ to $n-k$

..a) Get the maximum element from the BST, and print it.

..b) Search for $arr[i]$ in the BST and delete it from the BST.

..c) Insert $arr[i+k]$ into the BST.

Time Complexity: Time Complexity of step 1 is $O(k\log k)$. Time Complexity of steps 2(a), 2(b) and 2(c) is $O(\log k)$. Since steps 2(a), 2(b) and 2(c) are in a loop that runs $n-k+1$ times, time complexity of the complete algorithm is $O(k\log k + (n-k+1)*\log k)$ which can also be written as $O(n\log k)$.

Method 3 (A O(n) method: use Dequeue)

We create a [Dequeue](#), Qi of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain Qi to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the Qi is the largest and

element at rear of Q_i is the smallest of current window. Thanks to [Aashish](#) for suggesting this method.

Following is C++ implementation of this method.

```
#include <iostream>
#include <deque>

using namespace std;

// A Dequeue (Double ended queue) based method for printing maximum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front()] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int> Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
    {
        // For every element, the previous smaller elements are useless so
        // remove them from Qi
        while (!Qi.empty() && arr[i] >= arr[Qi.back()])
            Qi.pop_back(); // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while (!Qi.empty() && Qi.front() <= i - k)
            Qi.pop_front(); // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while (!Qi.empty() && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

        // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, 1, 78, 90, 57, 89, 56};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Output:

```
78 90 90 90 89
```

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed at most once. So there are total $2n$ operations.

Auxiliary Space: $O(k)$

Find whether an array is subset of another array | Added Method 3

Given two arrays: arr1[0..m-1] and arr2[0..n-1]. Find whether arr2[] is a subset of arr1[] or not. Both the arrays are not in sorted order. It may be assumed that elements in both array are distinct.

Examples:

Input: arr1[] = {11, 1, 13, 21, 3, 7}, arr2[] = {11, 3, 7, 1}

Output: arr2[] is a subset of arr1[]

Input: arr1[] = {1, 2, 3, 4, 5, 6}, arr2[] = {1, 2, 4}

Output: arr2[] is a subset of arr1[]

Input: arr1[] = {10, 5, 2, 23, 19}, arr2[] = {19, 5, 3}

Output: arr2[] is not a subset of arr1[]

Method 1 (Simple)

Use two loops: The outer loop picks all the elements of arr2[] one by one. The inner loop linearly searches for the element picked by outer loop. If all elements are found then return 1, else return 0.

```
#include<stdio.h>

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;
    int j = 0;
    for (i=0; i<n; i++)
    {
        for (j = 0; j<m; j++)
        {
            if(arr2[i] == arr1[j])
                break;
        }

        /* If the above inner loop was not broken at all then
         arr2[i] is not present in arr1[] */
        if (j == m)
            return 0;
    }

    /* If we reach here then all elements of arr2[]
     are present in arr1[] */
    return 1;
}

int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[] ");

    getchar();
    return 0;
}
```

Time Complexity: O(m*n)

Method 2 (Use Sorting and Binary Search)

- 1) Sort arr1[] O(mLogm)
- 2) For each element of arr2[], do binary search for it in sorted arr1[].
 - a) If the element is not found then return 0.
- 3) If all elements are present then return 1.

```
#include<stdio.h>

/* Function prototypes */
void quickSort(int *arr, int si, int ei);
int binarySearch(int arr[], int low, int high, int x);
```

```

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;

    quickSort(arr1, 0, m-1);
    for (i=0; i<n; i++)
    {
        if (binarySearch(arr1, 0, m-1, arr2[i]) == -1)
            return 0;
    }

    /* If we reach here then all elements of arr2[]
       are present in arr1[] */
    return 1;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SEARCHING AND SORTING PURPOSE */
/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int x)
{
    if(high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/

        /* Check if arr[mid] is the first occurrence of x.
           arr[mid] is first occurrence if x is one of the following
           is true:
           (i)  mid == 0 and arr[mid] == x
           (ii) arr[mid-1] < x and arr[mid] == x
        */
        if(( mid == 0 || x > arr[mid-1]) && (arr[mid] == x))
            return mid;
        else if(x > arr[mid])
            return binarySearch(arr, (mid + 1), high, x);
        else
            return binarySearch(arr, low, (mid -1), x);
    }
    return -1;
}

void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a    = *b;
    *b    = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

```

```

}

/*Driver program to test above functions */
int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[] ");

    getchar();
    return 0;
}

```

Time Complexity: $O(m\log m + n\log n)$. Please note that this will be the complexity if an $m\log m$ algorithm is used for sorting which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is $O(n^2)$

Method 3 (Use Sorting and Merging)

- 1) Sort both arrays: $arr1[]$ and $arr2[]$ $O(m\log m + n\log n)$
- 2) Use Merge type of process to see if all elements of sorted $arr2[]$ are present in sorted $arr1[]$.

Thanks to [Parthsarthi](#) for suggesting this method.

```

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0, j = 0;

    if(m < n)
        return 0;

    quickSort(arr1, 0, m-1);
    quickSort(arr2, 0, n-1);
    while( i < n && j < m )
    {
        if( arr1[j] < arr2[i] )
            j++;
        else if( arr1[j] == arr2[i] )
        {
            j++;
            i++;
        }
        else if( arr1[j] > arr2[i] )
            return 0;
    }

    if( i < n )
        return 0;
    else
        return 1;
}

```

Time Complexity: $O(m\log m + n\log n)$ which is better than method 2. Please note that this will be the complexity if an $n\log n$ algorithm is used for sorting both arrays which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is $O(n^2)$

Method 4 (Use Hashing)

- 1) Create a Hash Table for all the elements of $arr1[]$.
- 2) Traverse $arr2[]$ and search for each element of $arr2[]$ in the Hash Table. If element is not found then return 0.
- 3) If all elements are found then return 1.

Note that method 1, method 2 and method 4 don't handle the cases when we have duplicates in $arr2[]$. For example, $\{1, 4, 4, 2\}$ is not a subset of $\{1, 4, 2\}$, but these methods will print it as a subset.

Source: <http://geeksforgeeks.org/forum/topic/if-an-array-is-subset-of-another>

Find the minimum distance between two numbers

Given an unsorted array arr[] and two numbers x and y, find the minimum distance between x and y in arr[]. The array might also contain duplicates. You may assume that both x and y are different and present in arr[].

Examples:

Input: arr[] = {1, 2}, x = 1, y = 2

Output: Minimum distance between 1 and 2 is 1.

Input: arr[] = {3, 4, 5}, x = 3, y = 5

Output: Minimum distance between 3 and 5 is 2.

Input: arr[] = {3, 5, 4, 2, 6, 5, 6, 5, 4, 8, 3}, x = 3, y = 6

Output: Minimum distance between 3 and 6 is 4.

Input: arr[] = {2, 5, 3, 5, 4, 4, 2, 3}, x = 3, y = 2

Output: Minimum distance between 3 and 2 is 1.

Method 1 (Simple)

Use two loops: The outer loop picks all the elements of arr[] one by one. The inner loop picks all the elements after the element picked by outer loop. If the elements picked by outer and inner loops have same values as x or y then if needed update the minimum distance calculated so far.

```
#include <stdio.h>
#include <stdlib.h> // for abs()
#include <limits.h> // for INT_MAX

int minDist(int arr[], int n, int x, int y)
{
    int i, j;
    int min_dist = INT_MAX;
    for (i = 0; i < n; i++)
    {
        for (j = i+1; j < n; j++)
        {
            if( (x == arr[i] && y == arr[j] || y == arr[i] && x == arr[j]) && min_dist > abs(i-j))
            {
                min_dist = abs(i-j);
            }
        }
    }
    return min_dist;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {3, 5, 4, 2, 6, 5, 6, 5, 4, 8, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int y = 6;

    printf("Minimum distance between %d and %d is %d\n", x, y,
           minDist(arr, n, x, y));
    return 0;
}
```

Output: Minimum distance between 3 and 6 is 4

Time Complexity: O(n^2)

Method 2 (Tricky)

- 1) Traverse array from left side and stop if either x or y is found. Store index of this first occurrence in a variable say *prev*
- 2) Now traverse arr[] after the index *prev*. If the element at current index *i* matches with either x or y then check if it is different from arr[*prev*]. If it is different then update the minimum distance if needed. If it is same then update *prev* i.e., make *prev* = *i*.

Thanks to [wgpshashank](#) for suggesting this approach.

```
#include <stdio.h>
#include <limits.h> // For INT_MAX

int minDist(int arr[], int n, int x, int y)
{
    int i = 0;
    int min_dist = INT_MAX;
```

```

int prev;

// Find the first occurrence of any of the two numbers (x or y)
// and store the index of this occurrence in prev
for (i = 0; i < n; i++)
{
    if (arr[i] == x || arr[i] == y)
    {
        prev = i;
        break;
    }
}

// Traverse after the first occurrence
for ( ; i < n; i++)
{
    if (arr[i] == x || arr[i] == y)
    {
        // If the current element matches with any of the two then
        // check if current element and prev element are different
        // Also check if this value is smaller than minimum distance so far
        if (arr[prev] != arr[i] && (i - prev) < min_dist )
        {
            min_dist = i - prev;
            prev = i;
        }
        else
            prev = i;
    }
}

return min_dist;
}

/* Driver program to test above function */
int main()
{
    int arr[] ={3, 5, 4, 2, 6, 3, 0, 0, 5, 4, 8, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int y = 6;

    printf("Minimum distance between %d and %d is %d\n", x, y,
           minDist(arr, n, x, y));
    return 0;
}

```

Output: Minimum distance between 3 and 6 is 1

Time Complexity: O(n)

Find the repeating and the missing | Added 3 new methods

Given an unsorted array of size n. Array elements are in range from 1 to n. One number from set {1, 2, n} is missing and one number occurs twice in array. Find these two numbers.

Examples:

```
arr[] = {3, 1, 3}
Output: 2, 3 // 2 is missing and 3 occurs twice
```

```
arr[] = {4, 3, 6, 2, 1, 1}
Output: 1, 5 // 5 is missing and 1 occurs twice
```

Method 1 (Use Sorting)

- 1) Sort the input array.
- 2) Traverse the array and check for missing and repeating.

Time Complexity: O(nLogn)

Thanks to LoneShadow for suggesting this method.

Method 2 (Use count array)

- 1) Create a temp array temp[] of size n with all initial values as 0.
- 2) Traverse the input array arr[], and do following for each arr[i]
 - a) if(temp[arr[i]] == 0) temp[arr[i]] = 1;
 - b) if(temp[arr[i]] == 1) output arr[i] //repeating
- 3) Traverse temp[] and output the array element having value as 0 (This is the missing element)

Time Complexity: O(n)

Auxiliary Space: O(n)

Method 3 (Use elements as Index and mark the visited places)

Traverse the array. While traversing, use absolute value of every element as index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

```
#include<stdio.h>
#include<stdlib.h>

void printTwoElements(int arr[], int size)
{
    int i;
    printf("\n The repeating element is");

    for(i = 0; i < size; i++)
    {
        if(arr[abs(arr[i])-1] > 0)
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];
        else
            printf(" %d ", abs(arr[i]));
    }

    printf("\nand the missing element is ");
    for(i=0; i<size; i++)
    {
        if(arr[i]>0)
            printf("%d", i+1);
    }
}

/* Driver program to test above function */
int main()
{
    int arr[] = {7, 3, 4, 5, 5, 6, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printTwoElements(arr, n);
    return 0;
}
```

Time Complexity: O(n)

Thanks to Manish Mishra for suggesting this method.

Method 4 (Make two equations)

Let x be the missing and y be the repeating element.

1) Get sum of all numbers.

Sum of array computed $S = n(n+1)/2$

2) Get product of all numbers.

Product of array computed $P = 1*2*3**n$

3) The above two steps give us two equations, we can solve the equations and get the values of x and y.

Time Complexity: O(n)

Thanks to disappearedng for suggesting this solution.

This method can cause arithmetic overflow as we calculate product and sum of all array elements. See [this](#) for changes suggested by [john](#) to reduce the chances of overflow.

Method 5 (Use XOR)

Let x and y be the desired output elements.

Calculate XOR of all the array elements.

```
xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]
```

XOR the result with all numbers from 1 to n

```
xor1 = xor1^1^2^.....^n
```

In the result $xor1$, all elements would nullify each other except x and y. All the bits that are set in $xor1$ will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of $xor1$ and divide the elements of the array in two sets one set of elements with same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x, and by doing same in other set we will get y.

```
#include <stdio.h>
#include <stdlib.h>

/* The output of this function is stored at *x and *y */
void getTwoElements(int arr[], int n, int *x, int *y)
{
    int xor1; /* Will hold xor of all elements and numbers from 1 to n */
    int set_bit_no; /* Will have only single set bit of xor1 */
    int i;
    *x = 0;
    *y = 0;

    xor1 = arr[0];

    /* Get the xor of all array elements elements */
    for(i = 1; i < n; i++)
        xor1 = xor1^arr[i];

    /* XOR the previous result with numbers from 1 to n*/
    for(i = 1; i <= n; i++)
        xor1 = xor1^i;

    /* Get the rightmost set bit in set_bit_no */
    set_bit_no = xor1 & ~(xor1-1);

    /* Now divide elements in two sets by comparing rightmost set
       bit of xor1 with bit at same position in each element. Also, get XORS
       of two sets. The two XORS are the output elements.
       The following two for loops serve the purpose */
    for(i = 0; i < n; i++)
    {
        if(arr[i] & set_bit_no)
            *x = *x ^ arr[i]; /* arr[i] belongs to first set */
        else
            *y = *y ^ arr[i]; /* arr[i] belongs to second set*/
    }
    for(i = 1; i <= n; i++)
    {
        if(i & set_bit_no)
            *x = *x ^ i; /* i belongs to first set */
        else
            *y = *y ^ i; /* i belongs to second set*/
    }

    /* Now *x and *y hold the desired output elements */
}

/* Driver program to test above function */
int main()
```

```
{  
int arr[] = {1, 3, 4, 5, 5, 6, 2};  
int *x = (int *)malloc(sizeof(int));  
int *y = (int *)malloc(sizeof(int));  
int n = sizeof(arr)/sizeof(arr[0]);  
getTwoElements(arr, n, x, y);  
printf(" The two elements are %d and %d", *x, *y);  
getchar();  
}
```

Time Complexity: O(n)

This method doesn't cause overflow, but it doesn't tell which one occurs twice and which one is missing. We can add one more step that checks which one is missing and which one is repeating. This can be easily done in O(n) time.

Median in a stream of integers (running integers)

Given that integers are read from a data stream. Find median of elements read so far in efficient way. For simplicity assume there are no duplicates. For example, let us consider the stream 5, 15, 1, 3

```
After reading 1st element of stream - 5 -> median - 5
After reading 2nd element of stream - 5, 15 -> median - 10
After reading 3rd element of stream - 5, 15, 1 -> median - 5
After reading 4th element of stream - 5, 15, 1, 3 -> median - 4, so on...
```

Making it clear, when the input size is odd, we take the middle element of sorted data. If the input size is even, we pick average of middle two elements in sorted stream.

Note that output is *effective median* of integers read from the stream so far. Such an algorithm is called online algorithm. Any algorithm that can guarantee output of i -elements after processing i -th element, is said to be **online algorithm**. Let us discuss three solutions for the above problem.

Method 1: Insertion Sort

If we can sort the data as it appears, we can easily locate median element. *Insertion Sort* is one such online algorithm that sorts the data appeared so far. At any instance of sorting, say after sorting i -th element, the first i elements of array are sorted. The insertion sort doesn't depend on future data to sort data input till that point. In other words, insertion sort considers data sorted so far while inserting next element. This is the key part of insertion sort that makes it an online algorithm.

However, insertion sort takes $O(n^2)$ time to sort n elements. Perhaps we can use *binary search* on *insertion sort* to find location of next element in $O(\log n)$ time. Yet, we can't do data movement in $O(\log n)$ time. No matter how efficient the implementation is, it takes polynomial time in case of insertion sort.

Interested reader can try implementation of Method 1.

Method 2: Augmented self balanced binary search tree (AVL, RB, etc)

At every node of BST, maintain number of elements in the subtree rooted at that node. We can use a node as root of simple binary tree, whose left child is self balancing BST with elements less than root and right child is self balancing BST with elements greater than root. The root element always holds *effective median*.

If left and right subtrees contain same number of elements, root node holds average of left and right subtree root data. Otherwise, root contains same data as the root of subtree which is having more elements. After processing an incoming element, the left and right subtrees (BST) are differed utmost by 1.

Self balancing BST is costly in managing balancing factor of BST. However, they provide sorted data which we don't need. We need median only. The next method make use of Heaps to trace median.

Method 3: Heaps

Similar to balancing BST in Method 2 above, we can use a max heap on left side to represent elements that are less than *effective median*, and a min heap on right side to represent elements that are greater than *effective median*.

After processing an incoming element, the number of elements in heaps differ utmost by 1 element. When both heaps contain same number of elements, we pick average of heaps root data as *effective median*. When the heaps are not balanced, we select *effective median* from the root of heap containing more elements.

Given below is implementation of above method. For algorithm to build these heaps, please read the highlighted code.

```
#include <iostream>
using namespace std;

// Heap capacity
#define MAX_HEAP_SIZE (128)
#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

//// Utility functions

// exchange a and b
inline
void Exch(int &a, int &b)
{
    int aux = a;
    a = b;
    b = aux;
}

// Greater and Smaller are used as comparators
```

```

bool Greater(int a, int b)
{
    return a > b;
}

bool Smaller(int a, int b)
{
    return a < b;
}

int Average(int a, int b)
{
    return (a + b) / 2;
}

// Signum function
// = 0 if a == b - heaps are balanced
// = -1 if a < b - left contains less elements than right
// = 1 if a > b - left contains more elements than right
int Signum(int a, int b)
{
    if( a == b )
        return 0;

    return a < b ? -1 : 1;
}

// Heap implementation
// The functionality is embedded into
// Heap abstract class to avoid code duplication
class Heap
{
public:
    // Initializes heap array and comparator required
    // in heapification
    Heap(int *b, bool (*c)(int, int)) : A(b), comp(c)
    {
        heapSize = -1;
    }

    // Frees up dynamic memory
    virtual ~Heap()
    {
        if( A )
        {
            delete[] A;
        }
    }

    // We need only these four interfaces of Heap ADT
    virtual bool Insert(int e) = 0;
    virtual int GetTop() = 0;
    virtual int ExtractTop() = 0;
    virtual int GetCount() = 0;

protected:

    // We are also using location 0 of array
    int left(int i)
    {
        return 2 * i + 1;
    }

    int right(int i)
    {
        return 2 * (i + 1);
    }

    int parent(int i)
    {
        if( i <= 0 )
        {
            return -1;
        }

        return (i - 1)/2;
    }

    // Heap array
    int *A;
    // Comparator

```

```

bool (*comp)(int, int);
// Heap size
int heapSize;

// Returns top element of heap data structure
int top(void)
{
    int max = -1;

    if( heapSize >= 0 )
    {
        max = A[0];
    }

    return max;
}

// Returns number of elements in heap
int count()
{
    return heapSize + 1;
}

// Heapification
// Note that, for the current median tracing problem
// we need to heapify only towards root, always
void heapify(int i)
{
    int p = parent(i);

    // comp - differentiate MaxHeap and MinHeap
    // percolates up
    if( p >= 0 && comp(A[i], A[p]) )
    {
        Exch(A[i], A[p]);
        heapify(p);
    }
}

// Deletes root of heap
int deleteTop()
{
    int del = -1;

    if( heapSize > -1)
    {
        del = A[0];

        Exch(A[0], A[heapSize]);
        heapSize--;
        heapify(parent(heapSize+1));
    }

    return del;
}

// Helper to insert key into Heap
bool insertHelper(int key)
{
    bool ret = false;

    if( heapSize < MAX_HEAP_SIZE )
    {
        ret = true;
        heapSize++;
        A[heapSize] = key;
        heapify(heapSize);
    }

    return ret;
}
};

// Specilization of Heap to define MaxHeap
class MaxHeap : public Heap
{
private:

public:
    MaxHeap() : Heap(new int[MAX_HEAP_SIZE], &Greater) { }
}

```

```

~MaxHeap() { }

// Wrapper to return root of Max Heap
int GetTop()
{
    return top();
}

// Wrapper to delete and return root of Max Heap
int ExtractTop()
{
    return deleteTop();
}

// Wrapper to return # elements of Max Heap
int GetCount()
{
    return count();
}

// Wrapper to insert into Max Heap
bool Insert(int key)
{
    return insertHelper(key);
}
};

// Specilization of Heap to define MinHeap
class MinHeap : public Heap
{
private:

public:

    MinHeap() : Heap(new int[MAX_HEAP_SIZE], &Smaller) { }

    ~MinHeap() { }

    // Wrapper to return root of Min Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Min Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Min Heap
    int GetCount()
    {
        return count();
    }

    // Wrapper to insert into Min Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// Function implementing algorithm to find median so far.
int getMedian(int e, int &m, Heap &l, Heap &r)
{
    // Are heaps balanced? If yes, sig will be 0
    int sig = Signum(l.GetCount(), r.GetCount());
    switch(sig)
    {
    case 1: // There are more elements in left (max) heap

        if( e < m ) // current element fits in left (max) heap
        {
            // Remove top element from left heap and
            // insert into right heap
            r.Insert(l.ExtractTop());

            // current element fits in left (max) heap
            l.Insert(e);
        }
    }
}

```

```

        else
        {
            // current element fits in right (min) heap
            r.Insert(e);
        }

        // Both heaps are balanced
        m = Average(l.GetTop(), r.GetTop());

        break;
    }

    case 0: // The left and right heaps contain same number of elements

    if( e < m ) // current element fits in left (max) heap
    {
        l.Insert(e);
        m = l.GetTop();
    }
    else
    {
        // current element fits in right (min) heap
        r.Insert(e);
        m = r.GetTop();
    }

    break;

    case -1: // There are more elements in right (min) heap

    if( e < m ) // current element fits in left (max) heap
    {
        l.Insert(e);
    }
    else
    {
        // Remove top element from right heap and
        // insert into left heap
        l.Insert(r.ExtractTop());

        // current element fits in right (min) heap
        r.Insert(e);
    }

    // Both heaps are balanced
    m = Average(l.GetTop(), r.GetTop());

    break;
}

// No need to return, m already updated
return m;
}

void printMedian(int A[], int size)
{
    int m = 0; // effective median
    Heap *left = new MaxHeap();
    Heap *right = new MinHeap();

    for(int i = 0; i < size; i++)
    {
        m = getMedian(A[i], m, *left, *right);

        cout << m << endl;
    }

    // C++ more flexible, ensure no leaks
    delete left;
    delete right;
}

// Driver code
int main()
{
    int A[] = {5, 15, 1, 3, 2, 8, 7, 9, 10, 6, 11, 4};
    int size = ARRAY_SIZE(A);

    // In lieu of A, we can also use data read from a stream
    printMedian(A, size);

    return 0;
}

```

Time Complexity: If we omit the way how stream was read, complexity of median finding is $O(N \log N)$, as we need to read the stream, and due to heap insertions/deletions.

At first glance the above code may look complex. If you read the code carefully, it is simple algorithm

Find a Fixed Point in a given array

Given an array of n distinct integers sorted in ascending order, write a function that returns a Fixed Point in the array, if there is any Fixed Point present in array, else returns -1. Fixed Point in an array is an index i such that arr[i] is equal to i. Note that integers in array can be negative.

Examples:

```
Input: arr[] = {-10, -5, 0, 3, 7}
Output: 3 // arr[3] == 3
```

```
Input: arr[] = {0, 2, 5, 8, 17}
Output: 0 // arr[0] == 0
```

```
Input: arr[] = {-10, -5, 3, 4, 7, 9}
Output: -1 // No Fixed Point
```

Asked by [rajk](#)

Method 1 (Linear Search)

Linearly search for an index i such that arr[i] == i. Return the first such index found. Thanks to [pm](#) for suggesting this solution.

```
int linearSearch(int arr[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(arr[i] == i)
            return i;
    }

    /* If no fixed point present then return -1 */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Fixed Point is %d", linearSearch(arr, n));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Method 2 (Binary Search)

First check whether middle element is Fixed Point or not. If it is, then return it; otherwise check whether index of middle element is greater than value at the index. If index is greater, then Fixed Point(s) lies on the right side of the middle point (obviously only if there is a Fixed Point). Else the Fixed Point(s) lies on left side.

```
int binarySearch(int arr[], int low, int high)
{
    if(high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/
        if(mid == arr[mid])
            return mid;
        if(mid > arr[mid])
            return binarySearch(arr, (mid + 1), high);
        else
            return binarySearch(arr, low, (mid -1));
    }

    /* Return -1 if there is no Fixed Point */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[10] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Fixed Point is %d", binarySearch(arr, 0, n-1));
}
```

```
    getchar();  
    return 0;  
}
```

Algorithmic Paradigm: Divide & Conquer

Time Complexity: O(Logn)

Maximum Length Bitonic Subarray

Given an array $A[0 \dots n-1]$ containing n positive integers, a subarray $A[i \dots j]$ is bitonic if there is a k with $i \leq k \leq j$ such that $A[i] \leq A[i+1] \dots \leq A[k] \geq A[k+1] \geq \dots A[j-1] \geq A[j]$. Write a function that takes an array as argument and returns the length of the maximum length bitonic subarray.

Expected time complexity of the solution is $O(n)$

Simple Examples

1) $A[] = \{12, 4, 78, 90, 45, 23\}$, the maximum length bitonic subarray is $\{4, 78, 90, 45, 23\}$ which is of length 5.

2) $A[] = \{20, 4, 1, 2, 3, 4, 2, 10\}$, the maximum length bitonic subarray is $\{1, 2, 3, 4, 2\}$ which is of length 5.

Extreme Examples

1) $A[] = \{10\}$, the single element is bitonic, so output is 1.

2) $A[] = \{10, 20, 30, 40\}$, the complete array itself is bitonic, so output is 4.

3) $A[] = \{40, 30, 20, 10\}$, the complete array itself is bitonic, so output is 4.

Solution

Let us consider the array $\{12, 4, 78, 90, 45, 23\}$ to understand the solution.

1) Construct an auxiliary array $inc[]$ from left to right such that $inc[i]$ contains length of the nondecreasing subarray ending at $arr[i]$.
For $A[] = \{12, 4, 78, 90, 45, 23\}$, $inc[]$ is $\{1, 1, 2, 3, 1, 1\}$

2) Construct another array $dec[]$ from right to left such that $dec[i]$ contains length of nonincreasing subarray starting at $arr[i]$.
For $A[] = \{12, 4, 78, 90, 45, 23\}$, $dec[]$ is $\{2, 1, 1, 3, 2, 1\}$.

3) Once we have the $inc[]$ and $dec[]$ arrays, all we need to do is find the maximum value of $(inc[i] + dec[i] - 1)$.
For $\{12, 4, 78, 90, 45, 23\}$, the max value of $(inc[i] + dec[i] - 1)$ is 5 for $i = 3$.

C/C++

```
// C program to find length of the longest bitonic subarray
#include<stdio.h>
#include<stdlib.h>

int bitonic(int arr[], int n)
{
    int inc[n]; // Length of increasing subarray ending at all indexes
    int dec[n]; // Length of decreasing subarray starting at all indexes
    int i, max;

    // length of increasing sequence ending at first index is 1
    inc[0] = 1;

    // length of increasing sequence starting at first index is 1
    dec[n-1] = 1;

    // Step 1) Construct increasing sequence array
    for (i = 1; i < n; i++)
        inc[i] = (arr[i] > arr[i-1])? inc[i-1] + 1: 1;

    // Step 2) Construct decreasing sequence array
    for (i = n-2; i >= 0; i--)
        dec[i] = (arr[i] > arr[i+1])? dec[i+1] + 1: 1;

    // Step 3) Find the length of maximum length bitonic sequence
    max = inc[0] + dec[0] - 1;
    for (i = 1; i < n; i++)
        if (inc[i] + dec[i] - 1 > max)
            max = inc[i] + dec[i] - 1;

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {12, 4, 78, 90, 45, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("\nLength of max length Bitonic Subarray is %d",
          bitonic(arr, n));
    return 0;
}
```

Java

```
// Java program to find length of the longest bitonic subarray
import java.io.*;
import java.util.*;

class Bitonic
{
    static int bitonic(int arr[], int n)
    {
        int[] inc = new int[n]; // Length of increasing subarray ending
                               // at all indexes
        int[] dec = new int[n]; // Length of decreasing subarray starting
                               // at all indexes
        int max;

        // Length of increasing sequence ending at first index is 1
        inc[0] = 1;

        // Length of increasing sequence starting at first index is 1
        dec[n-1] = 1;

        // Step 1) Construct increasing sequence array
        for (int i = 1; i < n; i++)
            inc[i] = (arr[i] > arr[i-1])? inc[i-1] + 1: 1;

        // Step 2) Construct decreasing sequence array
        for (int i = n-2; i >= 0; i--)
            dec[i] = (arr[i] > arr[i+1])? dec[i+1] + 1: 1;

        // Step 3) Find the length of maximum length bitonic sequence
        max = inc[0] + dec[0] - 1;
        for (int i = 1; i < n; i++)
            if (inc[i] + dec[i] - 1 > max)
                max = inc[i] + dec[i] - 1;

        return max;
    }

    /*Driver function to check for above function*/
    public static void main (String[] args)
    {
        int arr[] = {12, 4, 78, 90, 45, 23};
        int n = arr.length;
        System.out.println("Length of max length Bitnoic Subarray is "
                           + bitonic(arr, n));
    }
}
/* This code is contributed by Devesh Agrawal */
```

Length of max length Bitnoic Subarray is 5

Time Complexity: O(n)

Auxiliary Space: O(n)

As an exercise, extend the above implementation to print the longest bitonic subarray also. The above implementation only returns the length of such subarray.

Find the maximum element in an array which is first increasing and then decreasing

Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.

Input: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}
Output: 500

Input: arr[] = {1, 3, 50, 10, 9, 7, 6}
Output: 50

Corner case (No decreasing part)
Input: arr[] = {10, 20, 30, 40, 50}
Output: 50

Corner case (No increasing part)
Input: arr[] = {120, 100, 80, 20, 0}
Output: 120

Method 1 (Linear Search)

We can traverse the array and keep track of maximum element. And finally return the maximum element.

```
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{
    int max = arr[low];
    int i;
    for (i = low; i <= high; i++)
    {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 30, 40, 50, 60, 70, 23, 20};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("The maximum element is %d", findMaximum(arr, 0, n-1));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Method 2 (Binary Search)

We can modify the standard Binary Search algorithm for the given type of arrays.

- i) If the mid element is greater than both of its adjacent elements, then mid is the maximum
- ii) If mid element is greater than its next element and smaller than the previous element then maximum lies on left side of mid. Example array: {3, 50, 10, 9, 7, 6}
- iii) If mid element is smaller than its next element and greater than the previous element then maximum lies on right side of mid. Example array: {2, 4, 6, 8, 10, 3, 1}

```
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{
    /* Base Case: Only one element is present in arr[low..high]*/
    if (low == high)
        return arr[low];

    /* If there are two elements and first is greater then
       the first element is maximum */
    if ((high == low + 1) && arr[low] >= arr[high])
        return arr[low];

    /* If there are two elements and second is greater then
       the second element is maximum */
    if ((high == low + 1) && arr[low] < arr[high])
        return arr[high];

    int mid = (low + high)/2; /*low + (high - low)/2;*/
}
```

```

/* If we reach a point where arr[mid] is greater than both of
   its adjacent elements arr[mid-1] and arr[mid+1], then arr[mid]
   is the maximum element*/
if ( arr[mid] > arr[mid + 1] && arr[mid] > arr[mid - 1])
    return arr[mid];

/* If arr[mid] is greater than the next element and smaller than the previous
   element then maximum lies on left side of mid */
if (arr[mid] > arr[mid + 1] && arr[mid] < arr[mid - 1])
    return findMaximum(arr, low, mid-1);
else // when arr[mid] is greater than arr[mid-1] and smaller than arr[mid+1]
    return findMaximum(arr, mid + 1, high);
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 3, 50, 10, 9, 7, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("The maximum element is %d", findMaximum(arr, 0, n-1));
    getchar();
    return 0;
}

```

Time Complexity: O(Logn)

This method works only for distinct numbers. For example, it will not work for an array like {0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 5, 3, 3, 2, 2, 1, 1}.

Count smaller elements on right side

Write a function to count number of smaller elements on right of each element in an array. Given an unsorted array arr[] of distinct integers, construct another array countSmaller[] such that countSmaller[i] contains count of smaller elements on right side of each element arr[i] in array.

Examples:

Input: arr[] = {12, 1, 2, 3, 0, 11, 4}
Output: countSmaller[] = {6, 1, 1, 1, 0, 1, 0}

(Corner Cases)

Input: arr[] = {5, 4, 3, 2, 1}
Output: countSmaller[] = {4, 3, 2, 1, 0}

Input: arr[] = {1, 2, 3, 4, 5}
Output: countSmaller[] = {0, 0, 0, 0, 0}

Method 1 (Simple)

Use two loops. The outer loop picks all elements from left to right. The inner loop iterates through all the elements on right side of the picked element and updates countSmaller[].

```
void constructLowerArray (int *arr[], int *countSmaller, int n)
{
    int i, j;

    // initialize all the counts in countSmaller array as 0
    for (i = 0; i < n; i++)
        countSmaller[i] = 0;

    for (i = 0; i < n; i++)
    {
        for (j = i+1; j < n; j++)
        {
            if (arr[j] < arr[i])
                countSmaller[i]++;
        }
    }
}

/* Utility function that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, 10, 5, 4, 2, 20, 6, 1, 0, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    int *low = (int *)malloc(sizeof(int)*n);
    constructLowerArray(arr, low, n);
    printArray(low, n);
    return 0;
}
```

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Method 2 (Use Self Balancing BST)

A Self Balancing Binary Search Tree (AVL, Red Black,.. etc) can be used to get the solution in O(nLogn) time complexity. We can augment these trees so that every node N contains size of the subtree rooted with N. We have used AVL tree in the following implementation.

We traverse the array from right to left and insert all elements one by one in an AVL tree. While inserting a new key in an AVL tree, we first compare the key with root. If key is greater than root, then it is greater than all the nodes in left subtree of root. So we add the size of left subtree to the count of smaller element for the key being inserted. We recursively follow the same approach for all nodes down the root.

Following is C implementation.

```
#include<stdio.h>
```

```

#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
    int size; // size of the tree rooted with this node
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree rooted with N
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to size of the tree of rooted with N
int size(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->size;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    node->size   = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Update sizes
    y->size = size(y->left) + size(y->right) + 1;
    x->size = size(x->left) + size(x->right) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;

```

```

x->right = T2;

// Update heights
x->height = max(height(x->left), height(x->right))+1;
y->height = max(height(y->left), height(y->right))+1;

// Update sizes
x->size = size(x->left) + size(x->right) + 1;
y->size = size(y->left) + size(y->right) + 1;

// Return new root
return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Inserts a new key to the tree rotated with node. Also, updates *count
// to contain count of smaller elements for the new key
struct node* insert(struct node* node, int key, int *count)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return (newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key, count);
    else
    {
        node->right = insert(node->right, key, count);

        // UPDATE COUNT OF SMALLER ELEMENTS FOR KEY
        *count = *count + size(node->left) + 1;
    }

    /* 2. Update height and size of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;
    node->size = size(node->left) + size(node->right) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// The following function updates the countSmaller array to contain count of
// smaller elements on right side.
void constructLowerArray (int arr[], int countSmaller[], int n)
{

```

```

int i, j;
struct node *root = NULL;

// initialize all the counts in countSmaller array as 0
for (i = 0; i < n; i++)
    countSmaller[i] = 0;

// Starting from rightmost element, insert all elements one by one in
// an AVL tree and get the count of smaller elements
for (i = n-1; i >= 0; i--)
{
    root = insert(root, arr[i], &countSmaller[i]);
}
}

/* Utility function that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    printf("\n");
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 6, 15, 20, 30, 5, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    int *low = (int *)malloc(sizeof(int)*n);

    constructLowerArray(arr, low, n);

    printf("Following is the constructed smaller count array");
    printArray(low, n);
    return 0;
}

```

Output:

Following is the constructed smaller count array
3 1 2 2 0 0

Time Complexity: O(nLogn)

Auxiliary Space: O(n)

Minimum number of jumps to reach end

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.

Example:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 ->9)
```

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9.

Method 1 (Naive Recursive Approach)

A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.

$\text{minJumps}(start, end) = \text{Min} (\text{minJumps}(k, end)) \text{ for all } k \text{ reachable from start}$

```
#include <stdio.h>
#include <limits.h>

// Returns minimum number of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int l, int h)
{
    // Base case: when source and destination are same
    if (h == l)
        return 0;

    // When nothing is reachable from the given source
    if (arr[l] == 0)
        return INT_MAX;

    // Traverse through all the points reachable from arr[l]. Recursively
    // get the minimum number of jumps needed to reach arr[h] from these
    // reachable points.
    int min = INT_MAX;
    for (int i = l+1; i <= h && i <= l + arr[l]; i++)
    {
        int jumps = minJumps(arr, i, h);
        if(jumps != INT_MAX && jumps + 1 < min)
            min = jumps + 1;
    }

    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-1));
    return 0;
}
```

If we trace the execution of this method, we can see that there will be overlapping subproblems. For example, $\text{minJumps}(3, 9)$ will be called two times as $\text{arr}[3]$ is reachable from $\text{arr}[1]$ and $\text{arr}[2]$. So this problem has both properties ([optimal substructure](#) and [overlapping subproblems](#)) of Dynamic Programming.

Method 2 (Dynamic Programming)

In this method, we build a $\text{jumps}[]$ array from left to right such that $\text{jumps}[i]$ indicates the minimum number of jumps needed to reach $\text{arr}[i]$ from $\text{arr}[0]$. Finally, we return $\text{jumps}[n-1]$.

```
#include <stdio.h>
#include <limits.h>

int min(int x, int y) { return (x < y)? x: y; }

// Returns minimum number of jumps to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    int *jumps = new int[n]; // jumps[n-1] will hold the result
    int i, j;
```

```

if (n == 0 || arr[0] == 0)
    return INT_MAX;

jumps[0] = 0;

// Find the minimum number of jumps to reach arr[i]
// from arr[0], and assign this value to jumps[i]
for (i = 1; i < n; i++)
{
    jumps[i] = INT_MAX;
    for (j = 0; j < i; j++)
    {
        if (i <= j + arr[j] && jumps[j] != INT_MAX)
        {
            jumps[i] = min(jumps[i], jumps[j] + 1);
            break;
        }
    }
}
return jumps[n-1];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 1, 0, 9};
    int size = sizeof(arr)/sizeof(int);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr,size));
    return 0;
}

```

Output:

Minimum number of jumps to reach end is 3

Thanks to [paras](#) for suggesting this method.

Time Complexity: O(n^2)

Method 3 (Dynamic Programming)

In this method, we build jumps[] array from right to left such that jumps[i] indicates the minimum number of jumps needed to reach arr[n-1] from arr[i]. Finally, we return arr[0].

```

int minJumps(int arr[], int n)
{
    int *jumps = new int[n]; // jumps[0] will hold the result
    int min;

    // Minimum number of jumps needed to reach last element
    // from last elements itself is always 0
    jumps[n-1] = 0;

    int i, j;

    // Start from the second element, move from right to left
    // and construct the jumps[] array where jumps[i] represents
    // minimum number of jumps needed to reach arr[m-1] from arr[i]
    for (i = n-2; i >= 0; i--)
    {
        // If arr[i] is 0 then arr[n-1] can't be reached from here
        if (arr[i] == 0)
            jumps[i] = INT_MAX;

        // If we can directly reach to the end point from here then
        // jumps[i] is 1
        else if (arr[i] >= n - i - 1)
            jumps[i] = 1;

        // Otherwise, to find out the minimum number of jumps needed
        // to reach arr[n-1], check all the points reachable from here
        // and jumps[] value for those points
        else
        {
            min = INT_MAX; // initialize min value

            // following loop checks with all reachable points and

```

```

// takes the minimum
for (j = i+1; j < n && j <= arr[i] + i; j++)
{
    if (min > jumps[j])
        min = jumps[j];
}

// Handle overflow
if (min != INT_MAX)
    jumps[i] = min + 1;
else
    jumps[i] = min; // or INT_MAX
}

return jumps[0];
}

```

Time Complexity: $O(n^2)$ in worst case.

Thanks to [Ashish](#) for suggesting this solution.

Implement two stacks in an array

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

push1(int x) > pushes x to first stack

push2(int x) > pushes x to second stack

pop1() > pops an element from first stack and return the popped element

pop2() > pops an element from second stack and return the popped element

Implementation of *twoStack* should be space efficient.

Method 1 (Divide the space in two halves)

A simple way to implement two stacks is to divide the array in two halves and assign the half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[n/2+1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

C++

```
#include<iostream>
#include<stdlib.h>

using namespace std;

class twoStacks
{
    int *arr;
    int size;
    int top1, top2;
public:
    twoStacks(int n) // constructor
    {
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top1++;
            arr[top1] = x;
        }
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top2--;
            arr[top2] = x;
        }
        else
        {
```

```

        cout << "Stack Overflow";
        exit(1);
    }

// Method to pop an element from first stack
int pop1()
{
    if (top1 >= 0 )
    {
        int x = arr[top1];
        top1--;
        return x;
    }
    else
    {
        cout << "Stack UnderFlow";
        exit(1);
    }
}

// Method to pop an element from second stack
int pop2()
{
    if (top2 < size)
    {
        int x = arr[top2];
        top2++;
        return x;
    }
    else
    {
        cout << "Stack UnderFlow";
        exit(1);
    }
}
};

/* Driver program to test twStacks class */
int main()
{
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
    return 0;
}

```

Python

```

# Python Script to Implement two stacks in a list
class twoStacks:

    def __init__(self, n):      #constructor
        self.size = n
        self.arr = [None] * n
        self.top1 = -1
        self.top2 = self.size

    # Method to push an element x to stack1
    def push1(self, x):

        # There is at least one empty space for new element
        if self.top1 < self.top2 - 1 :
            self.top1 = self.top1 + 1
            self.arr[self.top1] = x

        else:
            print("Stack Overflow ")
            exit(1)

    # Method to push an element x to stack2
    def push2(self, x):

```

```

# There is at least one empty space for new element
if self.top1 < self.top2 - 1:
    self.top2 = self.top2 - 1
    self.arr[self.top2] = x

else :
    print("Stack Overflow ")
    exit(1)

# Method to pop an element from first stack
def pop1(self):
    if self.top1 >= 0:
        x = self.arr[self.top1]
        self.top1 = self.top1 -1
        return x
    else:
        print("Stack Underflow ")
        exit(1)

# Method to pop an element from second stack
def pop2(self):
    if self.top2 < self.size:
        x = self.arr[self.top2]
        self.top2 = self.top2 + 1
        return x
    else:
        print("Stack Underflow ")
        exit()

# Driver program to test twoStacks class
ts = twoStacks(5)
ts.push1(5)
ts.push2(10)
ts.push2(15)
ts.push1(11)
ts.push2(7)

print("Popped element from stack1 is " + str(ts.pop1()))
ts.push2(40)
print("Popped element from stack2 is " + str(ts.pop2()))

# This code is contributed by Sunny Karira

```

Popped element from stack1 is 11
Popped element from stack2 is 40

Time complexity of all 4 operations of *twoStack* is O(1).
We will extend to 3 stacks in an array in a separate post.

Find subarray with given sum

Given an unsorted array of nonnegative integers, find a continuous subarray which adds to a given number.

Examples:

Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Output: Sum found between indexes 2 and 4

Input: arr[] = {1, 4, 0, 0, 3, 10, 5}, sum = 7
Output: Sum found between indexes 1 and 4

Input: arr[] = {1, 4}, sum = 0
Output: No subarray found

There may be more than one subarrays with sum as the given sum. The following solutions print first such subarray.

Source: Google Interview Question

Method 1 (Simple)

A simple solution is to consider all subarrays one by one and check the sum of every subarray. Following program implements the simple solution. We run two loops: the outer loop picks a starting point i and the inner loop tries all subarrays starting from i.

```
/* A simple program to print subarray with sum as given sum */  
#include<stdio.h>  
  
/* Returns true if there is a subarray of arr[] with sum equal to 'sum'  
otherwise returns false. Also, prints the result */  
int subArraySum(int arr[], int n, int sum)  
{  
    int curr_sum, i, j;  
  
    // Pick a starting point  
    for (i = 0; i < n; i++)  
    {  
        curr_sum = arr[i];  
  
        // try all subarrays starting with 'i'  
        for (j = i+1; j <= n; j++)  
        {  
            if (curr_sum == sum)  
            {  
                printf ("Sum found between indexes %d and %d", i, j-1);  
                return 1;  
            }  
            if (curr_sum > sum || j == n)  
                break;  
            curr_sum = curr_sum + arr[j];  
        }  
    }  
  
    printf("No subarray found");  
    return 0;  
}  
  
// Driver program to test above function  
int main()  
{  
    int arr[] = {15, 2, 4, 8, 9, 5, 10, 23};  
    int n = sizeof(arr)/sizeof(arr[0]);  
    int sum = 23;  
    subArraySum(arr, n, sum);  
    return 0;  
}
```

Output:

Sum found between indexes 1 and 4

Time Complexity: O(n^2) in worst case.

Method 2 (Efficient)

Initialize a variable curr_sum as first element. curr_sum indicates the sum of current subarray. Start from the second element and add all elements one by one to the curr_sum. If curr_sum becomes equal to sum, then print the solution. If curr_sum exceeds the sum, then remove trailing elements while curr_sum is greater than sum.

Following is C implementation of the above approach.

```
/* An efficient program to print subarray with sum as given sum */
#include<stdio.h>

/* Returns true if there is a subarray of arr[] with sum equal to 'sum'
   otherwise returns false. Also, prints the result */
int subArraySum(int arr[], int n, int sum)
{
    /* Initialize curr_sum as value of first element
       and starting point as 0 */
    int curr_sum = arr[0], start = 0, i;

    /* Add elements one by one to curr_sum and if the curr_sum exceeds the
       sum, then remove starting element */
    for (i = 1; i <= n; i++)
    {
        // If curr_sum exceeds the sum, then remove the starting elements
        while (curr_sum > sum && start < i-1)
        {
            curr_sum = curr_sum - arr[start];
            start++;
        }

        // If curr_sum becomes equal to sum, then return true
        if (curr_sum == sum)
        {
            printf ("Sum found between indexes %d and %d", start, i-1);
            return 1;
        }

        // Add this element to curr_sum
        if (i < n)
            curr_sum = curr_sum + arr[i];
    }

    // If we reach here, then no subarray
    printf("No subarray found");
    return 0;
}

// Driver program to test above function
int main()
{
    int arr[] = {15, 2, 4, 8, 9, 5, 10, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = 23;
    subArraySum(arr, n, sum);
    return 0;
}
```

Output:

```
Sum found between indexes 1 and 4
```

Time complexity of method 2 looks more than $O(n)$, but if we take a closer look at the program, then we can figure out the time complexity is $O(n)$. We can prove it by counting the number of operations performed on every element of arr[] in worst case. There are at most 2 operations performed on every element: (a) the element is added to the curr_sum (b) the element is subtracted from curr_sum. So the upper bound on number of operations is $2n$ which is $O(n)$.

Dynamic Programming | Set 14 (Maximum Sum Increasing Subsequence)

Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then output should be 10.

Solution

This problem is a variation of standard [Longest Increasing Subsequence \(LIS\) problem](#). We need a slight change in the Dynamic Programming solution of [LIS problem](#). All we need to change is to use sum as a criteria instead of length of increasing subsequence.

Following are C/C++ and Python implementations for Dynamic Programming solution of the problem.

C/C++

```
/* Dynamic Programming implementation of Maximum Sum Increasing
Subsequence (MSIS) problem */
#include<stdio.h>

/* maxSumIS() returns the maximum sum of increasing subsequence
   in arr[] of size n */
int maxSumIS( int arr[], int n )
{
    int i, j, max = 0;
    int msis[n];

    /* Initialize msis values for all indexes */
    for ( i = 0; i < n; i++ )
        msis[i] = arr[i];

    /* Compute maximum sum values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && msis[i] < msis[j] + arr[i] )
                msis[i] = msis[j] + arr[i];

    /* Pick maximum of all msis values */
    for ( i = 0; i < n; i++ )
        if ( max < msis[i] )
            max = msis[i];

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Sum of maximum sum increasing subsequence is %d\n",
           maxSumIS( arr, n ) );
    return 0;
}
```

Python

```
# Dynamic Programming based Python implementation of Maximum Sum Increasing
# Subsequence (MSIS) problem

# maxSumIS() returns the maximum sum of increasing subsequence in arr[] of
# size n
def maxSumIS(arr, n):
    max = 0
    msis = [0 for x in range(n)]

    # Initialize msis values for all indexes
    for i in range(n):
        msis[i] = arr[i]

    # Compute maximum sum values in bottom up manner
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j] and msis[i] < msis[j] + arr[i]:
                msis[i] = msis[j] + arr[i]

    # Pick maximum of all msis values
    for i in range(n):
        if max < msis[i]:
```

```
max = msis[i]

return max

# Driver program to test above function
arr = [1, 101, 2, 3, 100, 4, 5]
n = len(arr)
print("Sum of maximum sum increasing subsequence is " +
      str(maxSumIS(arr, n)))

# This code is contributed by Bhavya Jain
```

Output:

Sum of maximum sum increasing subsequence is 106

Time Complexity: $O(n^2)$

Source: [Maximum Sum Increasing Subsequence Problem](#)

Longest Increasing Subsequence Size (N log N)

After few months of gap posting an algo. The current post is pending from long time, and many readers (e.g [here](#), [here](#), [here](#) may be few more, I am not keeping track of all) are posting requests for explanation of the below problem.

Given an array of random numbers. Find *longest increasing subsequence* (LIS) in the array. I know many of you might have read recursive and dynamic programming (DP) solutions. There are few requests for [O\(N log N\)](#) algo in the forum posts.

For the time being, forget about recursive and DP solutions. Let us take small samples and extend the solution to large instances. Even though it may look complex at first time, once if we understood the logic, coding is simple.

Consider an input array $A = \{2, 5, 3\}$. I will extend the array during explanation.

By observation we know that the LIS is either $\{2, 3\}$ or $\{2, 5\}$. **Note that I am considering only strictly increasing sequences.**

Let us add two more elements, say $7, 11$ to the array. These elements will extend the existing sequences. Now the increasing sequences are $\{2, 3, 7, 11\}$ and $\{2, 5, 7, 11\}$ for the input array $\{2, 5, 3, 7, 11\}$.

Further, we add one more element, say 8 to the array i.e. input array becomes $\{2, 5, 3, 7, 11, 8\}$. Note that the latest element 8 is greater than smallest element of any active sequence (*will discuss shortly about active sequences*). How can we extend the existing sequences with 8 ? First of all, can 8 be part of LIS? If yes, how? If we want to add 8 , it should come after 7 (by replacing 11).

Since the approach is *offline* (*what we mean by [offline](#)?*), we are not sure whether adding 8 will extend the series or not. Assume there is 9 in the input array, say $\{2, 5, 3, 7, 11, 8, 7, 9\}$. We can replace 11 with 8 , as there is potentially *best candidate* (9) that can extend the new series $\{2, 3, 7, 8\}$ or $\{2, 5, 7, 8\}$.

Our observation is, assume that the end element of largest sequence is E . We can add (replace) current element $A[i]$ to the existing sequence if there is an element $A[j]$ ($j > i$) such that $E < A[i] < A[j]$ or $(E > A[i] < A[j]$ for replace). In the above example, $E = 11$, $A[i] = 8$ and $A[j] = 9$.

In case of our original array $\{2, 5, 3\}$, note that we face same situation when we are adding 3 to increasing sequence $\{2, 5\}$. I just created two increasing sequences to make explanation simple. Instead of two sequences, 3 can replace 5 in the sequence $\{2, 5\}$.

I know it will be confusing. I will clear it shortly!

The question is, when will it be safe to add or replace an element in the existing sequence?

Let us consider another sample $A = \{2, 5, 3\}$. Say, the next element is 1 . How can it extend the current sequences $\{2, 3\}$ or $\{2, 5\}$. Obviously, it cant extend either. Yet, there is a potential that the new smallest element can be start of an LIS. To make it clear, consider the array is $\{2, 5, 3, 1, 2, 3, 4, 5, 6\}$. Making 1 as new sequence will create new sequence which is largest.

The observation is, when we encounter new smallest element in the array, it can be a potential candidate to start new sequence.

From the observations, we need to maintain lists of increasing sequences.

In general, we have set of **active lists** of varying length. We are adding an element $A[i]$ to these lists. We scan the lists (for end elements) in decreasing order of their length. We will verify the end elements of all the lists to find a list whose end element is smaller than $A[i]$ (*floor value*).

Our strategy determined by the following conditions,

1. If $A[i]$ is smallest among all *end candidates of active lists*, we will *start new active list of length 1*.
2. If $A[i]$ is largest among all *end candidates of active lists*, we will *clone the largest active list, and extend it by $A[i]$* .
3. If $A[i]$ is in between, we will *find a list with largest end element that is smaller than $A[i]$. Clone and extend this list by $A[i]$. We will discard all other lists of same length as that of this modified list.*

Note that at any instance during our construction of active lists, the following condition is maintained.

end element of smaller list is smaller than end elements of larger lists.

It will be clear with an example, let us take example from [wiki](#) $\{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$.

$A[0] = 0$. Case 1. There are no active lists, create one.

0.

$A[1] = 8$. Case 2. Clone and extend.

0.

0, 8.

$A[2] = 4$. Case 3. Clone, extend and discard.

0.

0, 4.
~~0, 8.~~ Discarded

A[3] = 12. Case 2. Clone and extend.
0.
0, 4.
0, 4, 12.

A[4] = 2. Case 3. Clone, extend and discard.
0.
0, 2.
~~0, 4.~~ Discarded.
0, 4, 12.

A[5] = 10. Case 3. Clone, extend and discard.
0.
0, 2.
0, 2, 10.
~~0, 4, 12.~~ Discarded.

A[6] = 6. Case 3. Clone, extend and discard.
0.
0, 2.
0, 2, 6.
~~0, 2, 10.~~ Discarded.

A[7] = 14. Case 2. Clone and extend.
0.
0, 2.
0, 2, 6.
0, 2, 6, 14.

A[8] = 1. Case 3. Clone, extend and discard.
0.
0, 1.
~~0, 2.~~ Discarded.
0, 2, 6.
0, 2, 6, 14.

A[9] = 9. Case 3. Clone, extend and discard.
0.
0, 1.
0, 2, 6.
0, 2, 6, 9.
~~0, 2, 6, 14.~~ Discarded.

A[10] = 5. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 5.
~~0, 2, 6.~~ Discarded.
0, 2, 6, 9.

A[11] = 13. Case 2. Clone and extend.
0.
0, 1.
0, 1, 5.
0, 2, 6, 9.
0, 2, 6, 9, 13.

A[12] = 3. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
~~0, 1, 5.~~ Discarded.
0, 2, 6, 9.
0, 2, 6, 9, 13.

A[13] = 11. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 2, 6, 9.
0, 2, 6, 9, 11.
~~0, 2, 6, 9, 13.~~ Discarded.

A[14] = 7. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 1, 3, 7.

```
0, 2, 6, 9. Discarded.  
0, 2, 6, 9, 11.
```

```
A[15] = 15. Case 2. Clone and extend.
```

```
0.  
0, 1.  
0, 1, 3.  
0, 1, 3, 7.  
0, 2, 6, 9, 11.  
0, 2, 6, 9, 11, 15. <-- LIS List
```

It is required to understand above strategy to devise an algorithm. Also, ensure we have maintained the condition, *end element of smaller list is smaller than end elements of larger lists*. Try with few other examples, before reading further. It is important to understand what happening to end elements.

Algorithm:

Querying length of longest is fairly easy. Note that we are dealing with end elements only. We need not to maintain all the lists. We can store the end elements in an array. Discarding operation can be simulated with replacement, and extending a list is analogous to adding more elements to array.

We will use an auxiliary array to keep end elements. The maximum length of this array is that of input. In the worst case the array divided into N lists of size one (*note that it does not lead to worst case complexity*). To discard an element, we will trace ceil value of A[i] in auxiliary array (again observe the end elements in your rough work), and replace ceil value with A[i]. We extend a list by adding element to auxiliary array. We also maintain a counter to keep track of auxiliary array length.

Bonus: You have learned [Patience Sorting](#) technique partially :).

Here is a proverb, *Tell me and I will forget. Show me and I will remember. Involve me and I will understand.* So, pick a suit from deck of cards. Find the longest increasing sub-sequence of cards from the shuffled suit. You will never forget the approach. 😊

Given below is code to find length of LIS,

C++

```
// C++ program to find length of longest increasing subsequence  
// in O(n Log n) time  
#include <iostream>  
#include <string.h>  
#include <stdio.h>  
using namespace std;  
  
#define ARRAY_SIZE(A) sizeof(A)/sizeof(A[0])  
  
// Binary search (note boundaries in the caller)  
// A[] is ceilIndex in the caller  
int CeilIndex(int A[], int l, int r, int key)  
{  
    while (r - l > 1)  
    {  
        int m = l + (r - l)/2;  
        if (A[m]>=key)  
            r = m;  
        else  
            l = m;  
    }  
    return r;  
}  
  
int LongestIncreasingSubsequenceLength(int A[], int size)  
{  
    // Add boundary case, when array size is one  
  
    int *tailTable = new int[size];  
    int len; // always points empty slot  
  
    memset(tailTable, 0, sizeof(tailTable[0])*size);  
  
    tailTable[0] = A[0];  
    len = 1;  
    for (int i = 1; i < size; i++)  
    {  
        if (A[i] < tailTable[0])  
            // new smallest value  
            tailTable[0] = A[i];  
        else if (A[i] > tailTable[len-1])  
            tailTable[len] = A[i];  
        else  
            tailTable[len] = A[i];  
    }  
    return len;  
}
```

```

        else if (A[i] > tailTable[len-1])
            // A[i] wants to extend largest subsequence
            tailTable[len++] = A[i];

        else
            // A[i] wants to be current end candidate of an existing
            // subsequence. It will replace ceil value in tailTable
            tailTable[CeilIndex(tailTable, -1, len-1, A[i])] = A[i];
    }

    delete[] tailTable;
    return len;
}

int main()
{
    int A[] = { 2, 5, 3, 7, 11, 8, 10, 13, 6 };
    int n = ARRAY_SIZE(A);
    printf("Length of Longest Increasing Subsequence is %d\n",
           LongestIncreasingSubsequenceLength(A, n));

    return 0;
}

```

Java

```

// Java program to find length of longest increasing subsequence
// in O(n Log n) time
import java.io.*;
import java.util.*;
import java.lang.Math;

class LIS
{
    // Binary search (note boundaries in the caller)
    // A[] is ceilIndex in the caller
    static int CeilIndex(int A[], int l, int r, int key)
    {
        while (r - l > 1)
        {
            int m = l + (r - l)/2;
            if (A[m]>=key)
                r = m;
            else
                l = m;
        }

        return r;
    }

    static int LongestIncreasingSubsequenceLength(int A[], int size)
    {
        // Add boundary case, when array size is one

        int[] tailTable = new int[size];
        int len; // always points empty slot

        tailTable[0] = A[0];
        len = 1;
        for (int i = 1; i < size; i++)
        {
            if (A[i] < tailTable[0])
                // new smallest value
                tailTable[0] = A[i];

            else if (A[i] > tailTable[len-1])
                // A[i] wants to extend largest subsequence
                tailTable[len++] = A[i];

            else
                // A[i] wants to be current end candidate of an existing
                // subsequence. It will replace ceil value in tailTable
                tailTable[CeilIndex(tailTable, -1, len-1, A[i])] = A[i];
        }

        return len;
    }

    // Driver program to test above function
}

```

```

public static void main(String[] args)
{
    int A[] = { 2, 5, 3, 7, 11, 8, 10, 13, 6 };
    int n = A.length;
    System.out.println("Length of Longest Increasing Subsequence is "+
        LongestIncreasingSubsequenceLength(A, n));
}
/* This code is contributed by Devesh Agrawal*/

```

Length of Longest Increasing Subsequence is 6

Complexity:

The loop runs for N elements. In the worst case (what is worst case input?), we may end up querying ceil value using binary search ($\log i$) for many $A[i]$.

Therefore, $T(n) < O(\log N!) = O(N \log N)$. Analyse to ensure that the upper and lower bounds are also $O(N \log N)$. The complexity is $\Theta(N \log N)$.

Exercises:

1. [Design an algorithm to construct the longest increasing list.](#) Also, model your solution using DAGs.
2. Design an algorithm to construct **all** increasing lists of equal longest size.
3. Is the above algorithm an *online* algorithm?
4. Design an algorithm to construct the longest *decreasing* list..

Find a triplet that sum to a given value

Given an array and a value, find if there is a triplet in array whose sum is equal to the given value. If there is such a triplet present in array, then print the triplet and return true. Else return false. For example, if the given array is {12, 3, 4, 1, 6, 9} and given sum is 24, then there is a triplet (12, 3 and 9) present in array whose sum is 24.

Method 1 (Naive)

A simple method is to generate all possible triplets and compare the sum of every triplet with the given value. The following code implements this simple method using three nested loops.

```
# include <stdio.h>

// returns true if there is triplet with sum equal
// to 'sum' present in A[]. Also, prints the triplet
bool find3Numbers(int A[], int arr_size, int sum)
{
    int l, r;

    // Fix the first element as A[i]
    for (int i = 0; i < arr_size-2; i++)
    {
        // Fix the second element as A[j]
        for (int j = i+1; j < arr_size-1; j++)
        {
            // Now look for the third number
            for (int k = j+1; k < arr_size; k++)
            {
                if (A[i] + A[j] + A[k] == sum)
                {
                    printf("Triplet is %d, %d, %d", A[i], A[j], A[k]);
                    return true;
                }
            }
        }
    }

    // If we reach here, then no triplet was found
    return false;
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int sum = 22;
    int arr_size = sizeof(A)/sizeof(A[0]);

    find3Numbers(A, arr_size, sum);

    getchar();
    return 0;
}
```

Output:

Triplet is 4, 10, 8

Time Complexity: O(n^3)

Method 2 (Use Sorting)

Time complexity of the method 1 is O(n^3). The complexity can be reduced to O(n^2) by sorting the array first, and then using method 1 of [this post](#) in a loop.

- 1) Sort the input array.
- 2) Fix the first element as A[i] where i is from 0 to array size 2. After fixing the first element of triplet, find the other two elements using method 1 of [this post](#).

```
# include <stdio.h>

// A utility function to sort an array using Quicksort
void quickSort(int *, int, int);

// returns true if there is triplet with sum equal
// to 'sum' present in A[]. Also, prints the triplet
bool find3Numbers(int A[], int arr_size, int sum)
```

```

{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

    /* Now fix the first element one by one and find the
       other two elements */
    for (int i = 0; i < arr_size - 2; i++)
    {

        // To find the other two elements, start two index variables
        // from two corners of the array and move them toward each
        // other
        l = i + 1; // index of the first element in the remaining elements
        r = arr_size-1; // index of the last element
        while (l < r)
        {
            if( A[i] + A[l] + A[r] == sum)
            {
                printf("Triplet is %d, %d, %d", A[i], A[l], A[r]);
                return true;
            }
            else if (A[i] + A[l] + A[r] < sum)
                l++;
            else // A[i] + A[l] + A[r] > sum
                r--;
        }
    }

    // If we reach here, then no triplet was found
    return false;
}

/* FOLLOWING 2 FUNCTIONS ARE ONLY FOR SORTING
   PURPOSE */
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a    = *b;
    *b    = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

/* Driver program to test above function */
int main()
{

```

```
int A[] = {1, 4, 45, 6, 10, 8};  
int sum = 22;  
int arr_size = sizeof(A)/sizeof(A[0]);  
  
find3Numbers(A, arr_size, sum);  
  
getchar();  
return 0;  
}
```

Output:

Triplet is 4, 8, 10

Time Complexity: $O(n^2)$

Note that there can be more than one triplet with the given sum. We can easily modify the above methods to print all triplets.

Find the smallest positive number missing from an unsorted array

You are given an unsorted array with both positive and negative elements. You have to find the smallest positive number missing from the array in O(n) time using constant extra space. You can modify the original array.

Examples

Input: {2, 3, 7, 6, 8, -1, -10, 15}
Output: 1

Input: {2, 3, -7, 6, 8, 1, -10, 15}
Output: 4

Input: {1, 1, 0, -1, -2}
Output: 2

Source: [To find the smallest positive no missing from an unsorted array](#)

A **naive method** to solve this problem is to search all positive integers, starting from 1 in the given array. We may have to search at most n+1 numbers in the given array. So this solution takes O(n^2) in worst case.

We can **use sorting** to solve it in lesser time complexity. We can sort the array in O(nLogn) time. Once the array is sorted, then all we need to do is a linear scan of the array. So this approach takes O(nLogn + n) time which is O(nLogn).

We can also **use hashing**. We can build a hash table of all positive elements in the given array. Once the hash table is built. We can look in the hash table for all positive integers, starting from 1. As soon as we find a number which is not there in hash table, we return it. This approach may take O(n) time on average, but it requires O(n) extra space.

A O(n) time and O(1) extra space solution:

The idea is similar to [this post](#). We use array elements as index. To mark presence of an element x, we change the value at the index x to negative. But this approach doesn't work if there are non-positive (-ve and 0) numbers. So we segregate positive from negative numbers as first step and then apply the approach.

Following is the two step algorithm

- 1) Segregate positive numbers from others i.e., move all non-positive numbers to left side. In the following code, segregate() function does this part.
- 2) Now we can ignore non-positive elements and consider only the part of array which contains all positive elements. We traverse the array containing all positive numbers and to mark presence of an element x, we change the sign of value at index x to negative. We traverse the array again and print the first index which has positive value. In the following code, findMissingPositive() function does this part. Note that in findMissingPositive, we have subtracted 1 from the values as indexes start from 0 in C.

```
/* Program to find the smallest positive missing number */
#include <stdio.h>
#include <stdlib.h>

/* Utility to swap to integers */
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

/* Utility function that puts all non-positive (0 and negative) numbers on left
   side of arr[] and return count of such numbers */
int segregate (int arr[], int size)
{
    int j = 0, i;
    for(i = 0; i < size; i++)
    {
        if (arr[i] <= 0)
        {
            swap(&arr[i], &arr[j]);
            j++; // increment count of non-positive integers
        }
    }
    return j;
}

/* Find the smallest positive missing number in an array that contains
   all positive integers */
int findMissingPositive(int arr[], int size)
{
```

```

int i;

// Mark arr[i] as visited by making arr[arr[i] - 1] negative. Note that
// 1 is subtracted because index start from 0 and positive numbers start from 1
for(i = 0; i < size; i++)
{
    if(abs(arr[i]) - 1 < size && arr[abs(arr[i]) - 1] > 0)
        arr[abs(arr[i]) - 1] = -arr[abs(arr[i]) - 1];
}

// Return the first index value at which is positive
for(i = 0; i < size; i++)
    if (arr[i] > 0)
        return i+1; // 1 is added because indexes start from 0

return size+1;
}

/* Find the smallest positive missing number in an array that contains
   both positive and negative integers */
int findMissing(int arr[], int size)
{
    // First separate positive and negative numbers
    int shift = segregate (arr, size);

    // Shift the array and call findMissingPositive for
    // positive part
    return findMissingPositive(arr+shift, size-shift);
}

int main()
{
    int arr[] = {0, 10, 2, -10, -20};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    int missing = findMissing(arr, arr_size);
    printf("The smallest positive missing number is %d ", missing);
    getchar();
    return 0;
}

```

Output:

The smallest positive missing number is 1

Note that this method modifies the original array. We can change the sign of elements in the segregated array to get the same set of elements back. But we still lose the order of elements. If we want to keep the original array as it was, then we can create a copy of the array and run this approach on the temp array.

Find the two numbers with odd occurrences in an unsorted array

Given an unsorted array that contains even number of occurrences for all numbers except two numbers. Find the two numbers which have odd occurrences in O(n) time complexity and O(1) extra space.

Examples:

Input: {12, 23, 34, 12, 12, 23, 12, 45}
Output: 34 and 45

Input: {4, 4, 100, 5000, 4, 4, 4, 4, 100, 100}
Output: 100 and 5000

Input: {10, 20}
Output: 10 and 20

A **naive method** to solve this problem is to run two nested loops. The outer loop picks an element and the inner loop counts the number of occurrences of the picked element. If the count of occurrences is odd then print the number. The time complexity of this method is O(n^2).

We can **use sorting** to get the odd occurring numbers in O(nLogn) time. First sort the numbers using an O(nLogn) sorting algorithm like Merge Sort, Heap Sort.. etc. Once the array is sorted, all we need to do is a linear scan of the array and print the odd occurring number.

We can also **use hashing**. Create an empty hash table which will have elements and their counts. Pick all elements of input array one by one. Look for the picked element in hash table. If the element is found in hash table, increment its count in table. If the element is not found, then enter it in hash table with count as 1. After all elements are entered in hash table, scan the hash table and print elements with odd count. This approach may take O(n) time on average, but it requires O(n) extra space.

A O(n) time and O(1) extra space solution:

The idea is similar to method 2 of [this post](#). Let the two odd occurring numbers be x and y. We **use bitwise XOR** to get x and y. The first step is to do XOR of all elements present in array. XOR of all elements gives us XOR of x and y because of the following properties of XOR operation.

- 1) XOR of any number n with itself gives us 0, i.e., $n \wedge n = 0$
- 2) XOR of any number n with 0 gives us n, i.e., $n \wedge 0 = n$
- 3) XOR is cumulative and associative.

So we have XOR of x and y after the first step. Let the value of XOR be xor2. Every set bit in xor2 indicates that the corresponding bits in x and y have values different from each other. For example, if x = 6 (0110) and y is 15 (1111), then xor2 will be (1001), the two set bits in xor2 indicate that the corresponding bits in x and y are different. In the second step, we pick a set bit of xor2 and divide array elements in two groups. Both x and y will go to different groups. In the following code, the rightmost set bit of xor2 is picked as it is easy to get rightmost set bit of a number. If we do XOR of all those elements of array which have the corresponding bit set (or 1), then we get the first odd number. And if we do XOR of all those elements which have the corresponding bit 0, then we get the other odd occurring number. This step works because of the same properties of XOR. All the occurrences of a number will go in same set. XOR of all occurrences of a number which occur even number of times will result in 0 in its set. And the xor of a set will be one of the odd occurring elements.

```
// Program to find the two odd occurring elements
#include<stdio.h>

/* Prints two numbers that occur odd number of times. The
   function assumes that the array size is at least 2 and
   there are exactly two numbers occurring odd number of times. */
void printTwoOdd(int arr[], int size)
{
    int xor2 = arr[0]; /* Will hold XOR of two odd occurring elements */
    int set_bit_no; /* Will have only single set bit of xor2 */
    int i;
    int n = size - 2;
    int x = 0, y = 0;

    /* Get the xor of all elements in arr[]. The xor will basically
       be xor of two odd occurring elements */
    for(i = 1; i < size; i++)
        xor2 = xor2 ^ arr[i];

    /* Get one set bit in the xor2. We get rightmost set bit
       in the following line as it is easy to get */
    set_bit_no = xor2 & ~(xor2-1);

    /* Now divide elements in two sets:
       1) The elements having the corresponding bit as 1.
       2) The elements having the corresponding bit as 0. */
    for(i = 0; i < size; i++)
    {
        /* XOR of first set is finally going to hold one odd
```

```

        occurring number x */
if(arr[i] & set_bit_no)
    x = x ^ arr[i];

/* XOR of second set is finally going to hold the other
   odd occurring number y */
else
    y = y ^ arr[i];
}

printf("\n The two ODD elements are %d & %d ", x, y);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {4, 2, 4, 5, 2, 3, 3, 1};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printTwoOdd(arr, arr_size);
    getchar();
    return 0;
}

```

Output:

The two ODD elements are 5 & 1

Time Complexity: O(n)

Auxiliary Space: O(1)

The Celebrity Problem

Another classical problem

In a party of N people, only one person is known to everyone. Such a person **maybe present** in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like **does A know B?**. Find the stranger (celebrity) in minimum number of questions.

We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function *HaveAcquaintance(A, B)* which returns *true* if A knows B, *false* otherwise. How can we solve the problem, try yourself first.

We measure the complexity in terms of calls made to *HaveAcquaintance()*.

Graph:

We can model the solution using graphs. Initialize indegree and outdegree of every vertex as 0. If A knows B, draw a directed edge from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair [i, j]. We have $N C_2$ pairs. If celebrity is present in the party, we will have one sink node in the graph with outdegree of zero, and indegree of $N-1$. We can find the sink node in (N) time, but the overall complexity is $O(N^2)$ as we need to construct the graph first.

Recursion:

We can decompose the problem into combination of smaller instances. Say, if we know celebrity of $N-1$ persons, can we extend the solution to N ? We have two possibilities, *Celebrity(N-1)* may know N , or N already knew *Celebrity(N-1)*. In the former case, N will be celebrity if N doesn't know anyone else. In the later case we need to check that *Celebrity(N-1)* doesn't know N .

Solve the problem of smaller instance during divide step. On the way back, we may find a celebrity from the smaller instance. During combine stage, check whether the returned celebrity is known to everyone and he doesn't know anyone. The recurrence of the recursive decomposition is,

$$T(N) = T(N-1) + O(N)$$

$T(N) = O(N^2)$. You may try Writing pseudo code to check your recursion skills.

Using Stack:

The graph construction takes $O(N^2)$ time, it is similar to brute force search. In case of recursion, we reduce the problem instance by not more than one, and also combine step may examine $M-1$ persons (M instance size).

We have following observation based on elimination technique (Refer *Polyas How to Solve It* book).

- If A knows B, then A can't be celebrity. Discard A, and *B may be celebrity*.
- If A doesn't know B, then B can't be celebrity. Discard B, and *A may be celebrity*.
- Repeat above two steps till we left with only one person.
- Ensure the remained person is celebrity. (Why do we need this step?)

We can use stack to verify celebrity.

1. Push all the celebrities into a stack.
2. Pop off top two persons from the stack, discard one person based on return status of *HaveAcquaintance(A, B)*.
3. Push the remained person onto stack.
4. Repeat step 2 and 3 until only one person remains in the stack.
5. Check the remained person in stack doesn't have acquaintance with anyone else.

We will discard N elements utmost (Why?). If the celebrity is present in the party, we will call *HaveAcquaintance()* $3(N-1)$ times. Here is code using stack.

```
#include <iostream>
#include <list>
using namespace std;

// Max # of persons in the party
#define N 8

// Celebrities identified with numbers from 0 through size-1
int size = 4;
// Person with 2 is celebrity
bool MATRIX[N][N] = {{0, 0, 1, 0}, {0, 0, 1, 0}, {0, 0, 0, 0}, {0, 0, 1, 0}};

bool HaveAcquaintance(int a, int b) { return MATRIX[a][b]; }

int CelebrityUsingStack(int size)
{
```

```

// Handle trivial case of size = 2

list<int> stack; // Careful about naming
int i;
int C; // Celebrity

i = 0;
while( i < size )
{
    stack.push_back(i);
    i = i + 1;
}

int A = stack.back();
stack.pop_back();

int B = stack.back();
stack.pop_back();

while( stack.size() != 1 )
{
    if( HaveAcquaintance(A, B) )
    {
        A = stack.back();
        stack.pop_back();
    }
    else
    {
        B = stack.back();
        stack.pop_back();
    }
}

// Potential candidate?
C = stack.back();
stack.pop_back();

// Last candidate was not examined, it leads one excess comparison (optimise)
if( HaveAcquaintance(C, B) )
    C = B;

if( HaveAcquaintance(C, A) )
    C = A;

// I know these are redundant,
// we can simply check i against C
i = 0;
while( i < size )
{
    if( C != i )
        stack.push_back(i);
    i = i + 1;
}

while( !stack.empty() )
{
    i = stack.back();
    stack.pop_back();

    // C must not know i
    if( HaveAcquaintance(C, i) )
        return -1;

    // i must know C
    if( !HaveAcquaintance(i, C) )
        return -1;
}

return C;
}

int main()
{
    int id = CelebrityUsingStack(size);
    id == -1 ? cout << "No celebrity" : cout << "Celebrity ID " << id;
    return 0;
}

```

Output

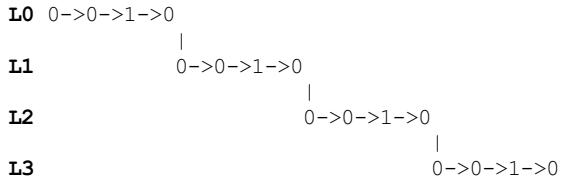
Celebrity ID 2

Complexity O(N). Total comparisons $3(N-1)$. Try the above code for successful MATRIX $\{\{0, 0, 0, 1\}, \{0, 0, 0, 1\}, \{0, 0, 0, 1\}, \{0, 0, 0, 1\}\}$.

A Note:

You may think that why do we need a new graph as we already have access to input matrix. Note that the matrix MATRIX used to help the hypothetical function *HaveAcquaintance(A, B)*, but never accessed via usual notation MATRIX[i, j]. We have access to the input only through the function *HaveAcquaintance(A, B)*. Matrix is just a way to code the solution. We can assume the cost of hypothetical function as O(1).

If still not clear, assume that the function *HaveAcquaintance* accessing information stored in a set of linked lists arranged in levels. List node will have *next* and *nextLevel* pointers. Every level will have N nodes i.e. an N element list, *next* points to next node in the current level list and *nextLevel* pointer in last node of every list will point to head of next level list. For example the linked list representation of above matrix looks like,



The function *HaveAcquaintance(i, j)* will search in the list for j -th node in the i -th level. Our goal is to minimize calls to *HaveAcquaintance* function.

Exercises:

1. Write code to find celebrity. Don't use any data structures like graphs, stack, etc you have access to N and *HaveAcquaintance(int, int)* only.
2. Implement the algorithm using Queues. What is your observation? Compare your solution with [Finding Maximum and Minimum](#) in an array and [Tournament Tree](#). What are minimum number of comparisons do we need (optimal number of calls to *HaveAcquaintance()*)?

Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

Given an array arr[0 n-1] containing n positive integers, a subsequence of arr[] is called Bitonic if it is first increasing, then decreasing. Write a function that takes an array as argument and returns the length of the longest bitonic subsequence.

A sequence, sorted in increasing order is considered Bitonic with the decreasing part as empty. Similarly, decreasing order sequence is considered Bitonic with the increasing part as empty.

Examples:

```
Input arr[] = {1, 11, 2, 10, 4, 5, 2, 1};  
Output: 6 (A Longest Bitonic Subsequence of length 6 is 1, 2, 10, 4, 2, 1)
```

```
Input arr[] = {12, 11, 40, 5, 3, 1}  
Output: 5 (A Longest Bitonic Subsequence of length 5 is 12, 11, 5, 3, 1)
```

```
Input arr[] = {80, 60, 30, 40, 20, 10}  
Output: 5 (A Longest Bitonic Subsequence of length 5 is 80, 60, 30, 20, 10)
```

Source: [Microsoft Interview Question](#)

Solution

This problem is a variation of standard [Longest Increasing Subsequence \(LIS\) problem](#). Let the input array be arr[] of length n. We need to construct two arrays lis[] and lds[] using Dynamic Programming solution of [LIS problem](#). lis[i] stores the length of the Longest Increasing subsequence ending with arr[i]. lds[i] stores the length of the longest Decreasing subsequence starting from arr[i]. Finally, we need to return the max value of lis[i] + lds[i] - 1 where i is from 0 to n-1.

Following is C++ implementation of the above Dynamic Programming solution.

C++

```
/* Dynamic Programming implementation of longest bitonic subsequence problem */  
#include<stdio.h>  
#include<stdlib.h>  
  
/* lbs() returns the length of the Longest Bitonic Subsequence in  
arr[] of size n. The function mainly creates two temporary arrays  
lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.  
lis[i] ==> Longest Increasing subsequence ending with arr[i]  
lds[i] ==> Longest decreasing subsequence starting with arr[i]  
*/  
int lbs( int arr[], int n )  
{  
    int i, j;  
  
    /* Allocate memory for LIS[] and initialize LIS values as 1 for  
    all indexes */  
    int *lis = new int[n];  
    for (i = 0; i < n; i++)  
        lis[i] = 1;  
  
    /* Compute LIS values from left to right */  
    for (i = 1; i < n; i++)  
        for (j = 0; j < i; j++)  
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)  
                lis[i] = lis[j] + 1;  
  
    /* Allocate memory for lds and initialize LDS values for  
    all indexes */  
    int *lds = new int [n];  
    for (i = 0; i < n; i++)  
        lds[i] = 1;  
  
    /* Compute LDS values from right to left */  
    for (i = n-2; i >= 0; i--)  
        for (j = n-1; j > i; j--)  
            if (arr[i] > arr[j] && lds[i] < lds[j] + 1)  
                lds[i] = lds[j] + 1;  
  
    /* Return the maximum value of lis[i] + lds[i] - 1*/  
    int max = lis[0] + lds[0] - 1;  
    for (i = 1; i < n; i++)  
        if (lis[i] + lds[i] - 1 > max)  
            max = lis[i] + lds[i] - 1;  
    return max;
```

```

}

/* Driver program to test above function */
int main()
{
    int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
                13, 3, 11, 7, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LBS is %d\n", lbs( arr, n ) );
    return 0;
}

```

Java

```

/* Dynamic Programming implementation in Java for longest bitonic
   subsequence problem */
import java.util.*;
import java.lang.*;
import java.io.*;

class LBS
{
    /* lbs() returns the length of the Longest Bitonic Subsequence in
       arr[] of size n. The function mainly creates two temporary arrays
       lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

       lis[i] ==> Longest Increasing subsequence ending with arr[i]
       lds[i] ==> Longest decreasing subsequence starting with arr[i]
    */
    static int lbs( int arr[], int n )
    {
        int i, j;

        /* Allocate memory for LIS[] and initialize LIS values as 1 for
           all indexes */
        int[] lis = new int[n];
        for (i = 0; i < n; i++)
            lis[i] = 1;

        /* Compute LIS values from left to right */
        for (i = 1; i < n; i++)
            for (j = 0; j < i; j++)
                if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                    lis[i] = lis[j] + 1;

        /* Allocate memory for lds and initialize LDS values for
           all indexes */
        int[] lds = new int[n];
        for (i = 0; i < n; i++)
            lds[i] = 1;

        /* Compute LDS values from right to left */
        for (i = n-2; i >= 0; i--)
            for (j = n-1; j > i; j--)
                if (arr[i] > arr[j] && lds[i] < lds[j] + 1)
                    lds[i] = lds[j] + 1;

        /* Return the maximum value of lis[i] + lds[i] - 1*/
        int max = lis[0] + lds[0] - 1;
        for (i = 1; i < n; i++)
            if (lis[i] + lds[i] - 1 > max)
                max = lis[i] + lds[i] - 1;

        return max;
    }

    public static void main (String[] args)
    {
        int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
                    13, 3, 11, 7, 15};
        int n = arr.length;
        System.out.println("Length of LBS is "+ lbs( arr, n ) );
    }
}

```

Length of LBS is 7

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n)$

Find a sorted subsequence of size 3 in linear time

Given an array of n integers, find the 3 elements such that $a[i] < a[j] < a[k]$ and $i < j < k$ in $O(n)$ time. If there are multiple such triplets, then print any one of them.

Examples:

Input: arr[] = {12, 11, 10, 5, 6, 2, 30}
Output: 5, 6, 30

Input: arr[] = {1, 2, 3, 4}
Output: 1, 2, 3 OR 1, 2, 4 OR 2, 3, 4

Input: arr[] = {4, 3, 2, 1}
Output: No such triplet

Source: [Amazon Interview Question](#)

Hint: Use Auxiliary Space

Solution:

- 1) Create an auxiliary array smaller[0..n-1]. smaller[i] should store the index of a number which is smaller than arr[i] and is on left side of arr[i]. smaller[i] should contain -1 if there is no such element.
- 2) Create another auxiliary array greater[0..n-1]. greater[i] should store the index of a number which is greater than arr[i] and is on right side of arr[i]. greater[i] should contain -1 if there is no such element.
- 3) Finally traverse both smaller[] and greater[] and find the index i for which both smaller[i] and greater[i] are not -1.

```
#include<stdio.h>

// A function to find a sorted subsequence of size 3
void find3Numbers(int arr[], int n)
{
    int max = n-1; //Index of maximum element from right side
    int min = 0; //Index of minimum element from left side
    int i;

    // Create an array that will store index of a smaller
    // element on left side. If there is no smaller element
    // on left side, then smaller[i] will be -1.
    int *smaller = new int[n];
    smaller[0] = -1; // first entry will always be -1
    for (i = 1; i < n; i++)
    {
        if (arr[i] <= arr[min])
        {
            min = i;
            smaller[i] = -1;
        }
        else
            smaller[i] = min;
    }

    // Create another array that will store index of a
    // greater element on right side. If there is no greater
    // element on right side, then greater[i] will be -1.
    int *greater = new int[n];
    greater[n-1] = -1; // last entry will always be -1
    for (i = n-2; i >= 0; i--)
    {
        if (arr[i] >= arr[max])
        {
            max = i;
            greater[i] = -1;
        }
        else
            greater[i] = max;
    }

    // Now find a number which has both a greater number on
    // right side and smaller number on left side
    for (i = 0; i < n; i++)
    {
        if (smaller[i] != -1 && greater[i] != -1)
        {
            printf("%d %d %d", arr[smaller[i]],
                   arr[i], arr[greater[i]]);
            return;
        }
    }
}
```

```

}

// If we reach number, then there are no such 3 numbers
printf("No such triplet found");

// Free the dynamically allocoed memory to avoid memory leak
delete [] smaller;
delete [] greater;

return;
}

// Driver program to test above function
int main()
{
    int arr[] = {12, 11, 10, 5, 6, 2, 30};
    int n = sizeof(arr)/sizeof(arr[0]);
    find3Numbers(arr, n);
    return 0;
}

```

Output:

5 6 30

Time Complexity: O(n)

Auxiliary Space: O(n)

Source: [How to find 3 numbers in increasing order and increasing indices in an array in linear time](#)

Exercise:

1. Find a subsequence of size 3 such that $\text{arr}[i] < \text{arr}[j] > \text{arr}[k]$.
2. Find a sorted subsequence of size 4 in linear time

Largest subarray with equal number of 0s and 1s

Given an array containing only 0s and 1s, find the largest subarray which contain equal no of 0s and 1s. Expected time complexity is O(n).

Examples:

Input: arr[] = {1, 0, 1, 1, 1, 0, 0}
Output: 1 to 6 (Starting and Ending indexes of output subarray)

Input: arr[] = {1, 1, 1, 1}
Output: No such subarray

Input: arr[] = {0, 0, 1, 1, 0}
Output: 0 to 3 Or 1 to 4

Source: [Largest subarray with equal number of 0s and 1s](#)

Method 1 (Simple)

A simple method is to use two nested loops. The outer loop picks a starting point i. The inner loop considers all subarrays starting from i. If size of a subarray is greater than maximum size so far, then update the maximum size.

In the below code, 0s are considered as -1 and sum of all values from i to j is calculated. If sum becomes 0, then size of this subarray is compared with largest size so far.

```
// A simple program to find the largest subarray with equal number of 0s and 1s
#include <stdio.h>

// This function Prints the starting and ending indexes of the largest subarray
// with equal number of 0s and 1s. Also returns the size of such subarray.
int findSubArray(int arr[], int n)
{
    int sum = 0;
    int maxsize = -1, startindex;

    // Pick a starting point as i
    for (int i = 0; i < n-1; i++)
    {
        sum = (arr[i] == 0)? -1 : 1;

        // Consider all subarrays starting from i
        for (int j = i+1; j < n; j++)
        {
            (arr[j] == 0)? sum += -1: sum += 1;

            // If this is a 0 sum subarray, then compare it with
            // maximum size subarray calculated so far
            if(sum == 0 && maxsize < j-i+1)
            {
                maxsize = j - i + 1;
                startindex = i;
            }
        }
    }
    if ( maxsize == -1 )
        printf("No such subarray");
    else
        printf("%d to %d", startindex, startindex+maxsize-1);

    return maxsize;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 0, 0, 1, 0, 1, 1};
    int size = sizeof(arr)/sizeof(arr[0]);

    findSubArray(arr, size);
    return 0;
}
```

Output:

0 to 5

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Method 2 (Tricky)

Following is a solution that uses O(n) extra space and solves the problem in O(n) time complexity.

Let input array be arr[] of size n and maxsize be the size of output subarray.

- 1) Consider all 0 values as -1. The problem now reduces to find out the maximum length subarray with sum = 0.
- 2) Create a temporary array sumleft[] of size n. Store the sum of all elements from arr[0] to arr[i] in sumleft[i]. This can be done in O(n) time.
- 3) There are two cases, the output subarray may start from 0th index or may start from some other index. We will return the max of the values obtained by two cases.
- 4) To find the maximum length subarray starting from 0th index, scan the sumleft[] and find the maximum i where sumleft[i] = 0.
- 5) Now, we need to find the subarray where subarray sum is 0 and start index is not 0. This problem is equivalent to finding two indexes i & j in sumleft[] such that sumleft[i] = sumleft[j] and j-i is maximum. To solve this, we can create a hash table with size = max-min+1 where min is the minimum value in the sumleft[] and max is the maximum value in the sumleft[]. The idea is to hash the leftmost occurrences of all different values in sumleft[]. The size of hash is chosen as max-min+1 because there can be these many different possible values in sumleft[]. Initialize all values in hash as -1
- 6) To fill and use hash[], traverse sumleft[] from 0 to n-1. If a value is not present in hash[], then store its index in hash. If the value is present, then calculate the difference of current index of sumleft[] and previously stored value in hash[]. If this difference is more than maxsize, then update the maxsize.
- 7) To handle corner cases (all 1s and all 0s), we initialize maxsize as -1. If the maxsize remains -1, then print there is no such subarray.

```
// A O(n) program to find the largest subarray with equal number of 0s and 1s
#include <stdio.h>
#include <stdlib.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return a>b? a: b; }

// This function Prints the starting and ending indexes of the largest subarray
// with equal number of 0s and 1s. Also returns the size of such subarray.
int findSubArray(int arr[], int n)
{
    int maxsize = -1, startIndex; // variables to store result values

    // Create an auxiliary array sumleft[]. sumleft[i] will be sum of array
    // elements from arr[0] to arr[i]
    int sumleft[n];
    int min, max; // For min and max values in sumleft[]
    int i;

    // Fill sumleft array and get min and max values in it.
    // Consider 0 values in arr[] as -1
    sumleft[0] = ((arr[0] == 0)? -1: 1);
    min = arr[0]; max = arr[0];
    for (i=1; i<n; i++)
    {
        sumleft[i] = sumleft[i-1] + ((arr[i] == 0)? -1: 1);
        if (sumleft[i] < min)
            min = sumleft[i];
        if (sumleft[i] > max)
            max = sumleft[i];
    }

    // Now calculate the max value of j - i such that sumleft[i] = sumleft[j].
    // The idea is to create a hash table to store indexes of all visited values.
    // If you see a value again, that it is a case of sumleft[i] = sumleft[j]. Check
    // if this j-i is more than maxsize.
    // The optimum size of hash will be max-min+1 as these many different values
    // of sumleft[i] are possible. Since we use optimum size, we need to shift
    // all values in sumleft[] by min before using them as an index in hash[].
    int hash[max-min+1];

    // Initialize hash table
    for (i=0; i<max-min+1; i++)
        hash[i] = -1;

    for (i=0; i<n; i++)
    {
        // Case 1: when the subarray starts from index 0
        if (sumleft[i] == 0)
        {
            maxsize = i+1;
            startIndex = 0;
        }

        // Case 2: fill hash table value. If already filled, then use it
        if (hash[sumleft[i]-min] == -1)
            hash[sumleft[i]-min] = i;
    }
}
```

```

    else
    {
        if ( (i - hash[sumleft[i]-min]) > maxsize )
        {
            maxsize = i - hash[sumleft[i]-min];
            startindex = hash[sumleft[i]-min] + 1;
        }
    }
    if ( maxsize == -1 )
        printf("No such subarray");
    else
        printf("%d to %d", startindex, startindex+maxsize-1);

    return maxsize;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 0, 0, 1, 0, 1, 1};
    int size = sizeof(arr)/sizeof(arr[0]);

    findSubArray(arr, size);
    return 0;
}

```

Output:

0 to 5

Time Complexity: O(n)

Auxiliary Space: O(n)

Thanks to [Aashish Barnwal](#) for suggesting this solution.

Dynamic Programming | Set 18 (Partition problem)

Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Examples

```
arr[] = {1, 5, 11, 5}
Output: true
The array can be partitioned as {1, 5, 5} and {11}
```

```
arr[] = {1, 5, 3}
Output: false
The array cannot be partitioned into equal sum sets.
```

Following are the two main steps to solve this problem:

- 1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
- 2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.

The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Recursive Solution

Following is the recursive property of the second step mentioned above.

Let `isSubsetSum(arr, n, sum/2)` be the function that returns true if there is a subset of `arr[0..n-1]` with sum equal to `sum/2`

The `isSubsetSum` problem can be divided into two subproblems

- a) `isSubsetSum()` without considering last element
(reducing n to n-1)
- b) `isSubsetSum` considering the last element
(reducing sum/2 by arr[n-1] and n to n-1)

If any of the above the above subproblems return true, then return true.

```
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||
                                isSubsetSum (arr, n-1, sum/2 - arr[n-1])
```

C/C++

```
// A recursive C program for partition problem
#include <stdio.h>
```

```
// A utility function that returns true if there is
// a subset of arr[] with sun equal to given sum
```

```
bool isSubsetSum (int arr[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;
```

```
    // If last element is greater than sum, then
    // ignore it
    if (arr[n-1] > sum)
        return isSubsetSum (arr, n-1, sum);
```

```
    /* else, check if sum can be obtained by any of
       the following
       (a) including the last element
       (b) excluding the last element
    */
    return isSubsetSum (arr, n-1, sum) ||
           isSubsetSum (arr, n-1, sum-arr[n-1]);
```

```
}
```

```
// Returns true if arr[] can be partitioned in two
// subsets of equal sum, otherwise false
```

```
bool findPartition (int arr[], int n)
{
```

```
    // Calculate sum of the elements in array
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];
```

```
    // If sum is odd, there cannot be two subsets
    // with equal sum
    if (sum%2 != 0)
        return false;
```

```

// Find if there is subset with sum equal to
// half of total sum
return isSubsetSum (arr, n, sum/2);
}

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 5, 9, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (findPartition(arr, n) == true)
        printf("Can be divided into two subsets "
               "of equal sum");
    else
        printf("Can not be divided into two subsets"
               " of equal sum");
    return 0;
}

```

Java

```

// A recursive Java solution for partition problem
import java.io.*;

class Partition
{
    // A utility function that returns true if there is a
    // subset of arr[] with sun equal to given sum
    static boolean isSubsetSum (int arr[], int n, int sum)
    {
        // Base Cases
        if (sum == 0)
            return true;
        if (n == 0 && sum != 0)
            return false;

        // If last element is greater than sum, then ignore it
        if (arr[n-1] > sum)
            return isSubsetSum (arr, n-1, sum);

        /* else, check if sum can be obtained by any of
           the following
           (a) including the last element
           (b) excluding the last element
        */
        return isSubsetSum (arr, n-1, sum) ||
               isSubsetSum (arr, n-1, sum-arr[n-1]);
    }

    // Returns true if arr[] can be partitioned in two
    // subsets of equal sum, otherwise false
    static boolean findPartition (int arr[], int n)
    {
        // Calculate sum of the elements in array
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += arr[i];

        // If sum is odd, there cannot be two subsets
        // with equal sum
        if (sum%2 != 0)
            return false;

        // Find if there is subset with sum equal to half
        // of total sum
        return isSubsetSum (arr, n, sum/2);
    }

    /*Driver function to check for above function*/
    public static void main (String[] args)
    {

        int arr[] = {3, 1, 5, 9, 12};
        int n = arr.length;
        if (findPartition(arr, n) == true)
            System.out.println("Can be divided into two "+
                               "subsets of equal sum");
        else
            System.out.println("Can not be divided into " +

```

```

        "two subsets of equal sum");
    }
}

/* This code is contributed by Devesh Agrawal */

```

Can be divided into two subsets of equal sum

Time Complexity: O(2^n) In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

Dynamic Programming Solution

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array part[][] of size $(\text{sum}/2) \times (n+1)$. And we can construct the solution in bottom up manner such that every filled entry has following property

```

part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]} has sum
            equal to i, otherwise false

```

C/C++

```

// A Dynamic Programming based C program to partition problem
#include <stdio.h>

// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    int sum = 0;
    int i, j;

    // Calculate sum of all elements
    for (i = 0; i < n; i++)
        sum += arr[i];

    if (sum%2 != 0)
        return false;

    bool part[sum/2+1][n+1];

    // Initialize top row as true
    for (i = 0; i <= n; i++)
        part[0][i] = true;

    // Initialize leftmost column, except part[0][0], as 0
    for (i = 1; i <= sum/2; i++)
        part[i][0] = false;

    // Fill the partition table in bottom up manner
    for (i = 1; i <= sum/2; i++)
    {
        for (j = 1; j <= n; j++)
        {
            part[i][j] = part[i][j-1];
            if (i >= arr[j-1])
                part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
        }
    }

    /* // uncomment this part to print table
    for (i = 0; i <= sum/2; i++)
    {
        for (j = 0; j <= n; j++)
            printf ("%4d", part[i][j]);
        printf("\n");
    } */

    return part[sum/2][n];
}

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 1, 2, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (findPartiion(arr, n) == true)
        printf("Can be divided into two subsets of equal sum");
    else

```

```

    printf("Can not be divided into two subsets of equal sum");
getchar();
return 0;
}

```

Java

```

// A dynamic programming based Java program for partition problem
import java.io.*;

class Partition {

    // Returns true if arr[] can be partitioned in two subsets of
    // equal sum, otherwise false
    static boolean findPartition (int arr[], int n)
    {
        int sum = 0;
        int i, j;

        // Caculate sun of all elements
        for (i = 0; i < n; i++)
            sum += arr[i];

        if (sum%2 != 0)
            return false;

        boolean part[][]=new boolean[sum/2+1][n+1];

        // initialize top row as true
        for (i = 0; i <= n; i++)
            part[0][i] = true;

        // initialize leftmost column, except part[0][0], as 0
        for (i = 1; i <= sum/2; i++)
            part[i][0] = false;

        // Fill the partition table in bottom up manner
        for (i = 1; i <= sum/2; i++)
        {
            for (j = 1; j <= n; j++)
            {
                part[i][j] = part[i][j-1];
                if (i >= arr[j-1])
                    part[i][j] = part[i][j] ||
                                  part[i - arr[j-1]][j-1];
            }
        }

        /* // uncomment this part to print table
        for (i = 0; i <= sum/2; i++)
        {
            for (j = 0; j <= n; j++)
                printf ("%4d", part[i][j]);
            printf("\n");
        } */

        return part[sum/2][n];
    }

    /*Driver function to check for above function*/
    public static void main (String[] args)
    {
        int arr[] = {3, 1, 1, 2, 2,1};
        int n = arr.length;
        if (findPartition(arr, n) == true)
            System.out.println("Can be divided into two "
                               "subsets of equal sum");
        else
            System.out.println("Can not be divided into"
                               " two subsets of equal sum");

    }
}
/* This code is contributed by Devesh Agrawal */

```

Can be divided into two subsets of equal sum

Following diagram shows the values in partition table. The diagram is taken from the [wiki page of partition problem](#).

The entry part[i][j] indicates whether there is a subset of {arr[0], arr[1], .. arr[j-1]} that sums to i

| | {} | {3} | {3,1} | {3,1,1} | {3,1,1,2} | {3,1,1,2,2} | {3,1,1,2,2,1} |
|---|-------|-------|-------|---------|-----------|-------------|---------------|
| 0 | True | True | True | True | True | True | True |
| 1 | False | False | True | True | True | True | True |
| 2 | False | False | False | True | True | True | True |
| 3 | False | True | True | True | True | True | True |
| 4 | False | False | True | True | True | True | True |
| 5 | False | False | False | True | True | True | True |

Dynamic Programming table for

arr[] = {3, 1, 1, 2, 2, 1}

Time Complexity: O(sum*n)

Auxiliary Space: O(sum*n)

Please note that this solution will not be feasible for arrays with big sum.

References:

http://en.wikipedia.org/wiki/Partition_problem

Maximum Product Subarray

Given an array that contains both positive and negative integers, find the product of the maximum product subarray. Expected Time complexity is O(n) and only O(1) extra space can be used.

Examples:

Input: arr[] = {6, -3, -10, 0, 2}
Output: 180 // The subarray is {6, -3, -10}

Input: arr[] = {-1, -3, -10, 0, 60}
Output: 60 // The subarray is {60}

Input: arr[] = {-2, -3, 0, -2, -40}
Output: 80 // The subarray is {-2, -40}

The following solution assumes that the given input array always has a positive output. The solution works for all cases mentioned above. It doesn't work for arrays like {0, 0, -20, 0}, {0, 0, 0}.. etc. The solution can be easily modified to handle this case.

It is similar to [Largest Sum Contiguous Subarray](#) problem. The only thing to note here is, maximum product can also be obtained by minimum (negative) product ending with the previous element multiplied by this element. For example, in array {12, 2, -3, -5, -6, -2}, when we are at element -2, the maximum product is multiplication of minimum product ending with -6 and -2.

```
#include <stdio.h>

// Utility functions to get minimum of two integers
int min (int x, int y) {return x < y? x : y; }

// Utility functions to get maximum of two integers
int max (int x, int y) {return x > y? x : y; }

/* Returns the product of max product subarray. Assumes that the
   given array always has a subarray with product more than 1 */
int maxSubarrayProduct(int arr[], int n)
{
    // max positive product ending at the current position
    int max_ending_here = 1;

    // min negative product ending at the current position
    int min_ending_here = 1;

    // Initialize overall max product
    int max_so_far = 1;

    /* Traverse through the array. Following values are maintained after the ith iteration:
       max_ending_here is always 1 or some positive product ending with arr[i]
       min_ending_here is always 1 or some negative product ending with arr[i] */
    for (int i = 0; i < n; i++)
    {
        /* If this element is positive, update max_ending_here. Update
           min_ending_here only if min_ending_here is negative */
        if (arr[i] > 0)
        {
            max_ending_here = max_ending_here*arr[i];
            min_ending_here = min (min_ending_here * arr[i], 1);
        }

        /* If this element is 0, then the maximum product cannot
           end here, make both max_ending_here and min_ending_here 0
           Assumption: Output is always greater than or equal to 1. */
        else if (arr[i] == 0)
        {
            max_ending_here = 1;
            min_ending_here = 1;
        }

        /* If element is negative. This is tricky
           max_ending_here can either be 1 or positive. min_ending_here can either be 1
           or negative.
           next min_ending_here will always be prev. max_ending_here * arr[i]
           next max_ending_here will be 1 if prev min_ending_here is 1, otherwise
           next max_ending_here will be prev min_ending_here * arr[i] */
        else
        {
            int temp = max_ending_here;
            max_ending_here = max (min_ending_here * arr[i], 1);
            min_ending_here = temp * arr[i];
        }
    }
}
```

```
// update max_so_far, if needed
if (max_so_far < max_ending_here)
    max_so_far = max_ending_here;
}

return max_so_far;
```

// Driver Program to test above function

```
int main()
{
    int arr[] = {1, -2, -3, 0, 7, -8, -2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Sub array product is %d", maxSubarrayProduct(arr, n));
    return 0;
}
```

Output:

```
Maximum Sub array product is 112
```

Time Complexity: O(n)

Auxiliary Space: O(1)

Find a pair with the given difference

Given an unsorted array and a number n, find if there exists a pair of elements in the array whose difference is n.

Examples:

Input: arr[] = {5, 20, 3, 2, 50, 80}, n = 78
Output: Pair Found: (2, 80)

Input: arr[] = {90, 70, 20, 80, 50}, n = 45
Output: No Such Pair

Source: [find pair](#)

The simplest method is to run two loops, the outer loop picks the first element (smaller element) and the inner loop looks for the element picked by outer loop plus n. Time complexity of this method is $O(n^2)$.

We can use sorting and Binary Search to improve time complexity to $O(n \log n)$. The first step is to sort the array in ascending order. Once the array is sorted, traverse the array from left to right, and for each element $\text{arr}[i]$, binary search for $\text{arr}[i] + n$ in $\text{arr}[i+1..n-1]$. If the element is found, return the pair.

Both first and second steps take $O(n \log n)$. So overall complexity is $O(n \log n)$.

The second step of the above algorithm can be improved to $O(n)$. The first step remain same. The idea for second step is take two index variables i and j, initialize them as 0 and 1 respectively. Now run a linear loop. If $\text{arr}[j] - \text{arr}[i]$ is smaller than n, we need to look for greater $\text{arr}[j]$, so increment j. If $\text{arr}[j] - \text{arr}[i]$ is greater than n, we need to look for greater $\text{arr}[i]$, so increment i. Thanks to [Aashish Barnwal](#) for suggesting this approach.

The following code is only for the second step of the algorithm, it assumes that the array is already sorted.

```
#include <stdio.h>

// The function assumes that the array is sorted
bool findPair(int arr[], int size, int n)
{
    // Initialize positions of two elements
    int i = 0;
    int j = 1;

    // Search for a pair
    while (i<size && j<size)
    {
        if (i != j && arr[j]-arr[i] == n)
        {
            printf("Pair Found: (%d, %d)", arr[i], arr[j]);
            return true;
        }
        else if (arr[j]-arr[i] < n)
            j++;
        else
            i++;
    }

    printf("No such pair");
    return false;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 8, 30, 40, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    int n = 60;
    findPair(arr, size, n);
    return 0;
}
```

Output:

Pair Found: (40, 100)

Hashing can also be used to solve this problem. Create an empty hash table HT. Traverse the array, use array elements as hash keys and enter them in HT. Traverse the array again look for value $n + \text{arr}[i]$ in HT.

Replace every element with the greatest element on right side

Given an array of integers, replace every element with the next greatest element (greatest element on the right side) in the array. Since there is no element next to the last element, replace it with -1. For example, if the array is {16, 17, 4, 3, 5, 2}, then it should be modified to {17, 5, 5, 5, 2, -1}.

The question is very similar to [this post](#) and solutions are also similar.

A **naive method** is to run two loops. The outer loop will one by one pick array elements from left to right. The inner loop will find the greatest element present after the picked element. Finally the outer loop will replace the picked element with the greatest element found by inner loop. The time complexity of this method will be $O(n^2)$.

A **tricky method** is to replace all elements using one traversal of the array. The idea is to start from the rightmost element, move to the left side one by one, and keep track of the maximum element. Replace every element with the maximum element.

```
#include <stdio.h>

/* Function to replace every element with the
   next greatest element */
void nextGreatest(int arr[], int size)
{
    // Initialize the next greatest element
    int max_from_right = arr[size-1];

    // The next greatest element for the rightmost element
    // is always -1
    arr[size-1] = -1;

    // Replace all other elements with the next greatest
    for(int i = size-2; i >= 0; i--)
    {
        // Store the current element (needed later for updating
        // the next greatest element)
        int temp = arr[i];

        // Replace current element with the next greatest
        arr[i] = max_from_right;

        // Update the greatest element, if needed
        if(max_from_right < temp)
            max_from_right = temp;
    }
}

/* A utility Function that prints an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver program to test above function */
int main()
{
    int arr[] = {16, 17, 4, 3, 5, 2};
    int size = sizeof(arr)/sizeof(arr[0]);
    nextGreatest (arr, size);
    printf ("The modified array is: \n");
    printArray (arr, size);
    return (0);
}
```

Output:

```
The modified array is:
17 5 5 5 2 -1
```

Time Complexity: $O(n)$ where n is the number of elements in array.

Dynamic Programming | Set 20 (Maximum Length Chain of Pairs)

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if $b < c$. Chain of pairs can be formed in this fashion. Find the longest chain which can be formed from a given set of pairs. Source: [Amazon Interview | Set 2](#)

For example, if the given pairs are $\{\{5, 24\}, \{39, 60\}, \{15, 28\}, \{27, 40\}, \{50, 90\}\}$, then the longest chain that can be formed is of length 3, and the chain is $\{\{5, 24\}, \{27, 40\}, \{50, 90\}\}$

This problem is a variation of standard [Longest Increasing Subsequence](#) problem. Following is a simple two step process.

- 1) Sort given pairs in increasing order of first (or smaller) element.
- 2) Now run a modified LIS process where we compare the second element of already finalized LIS with the first element of new LIS being constructed.

The following code is a slight modification of method 2 of [this post](#).

```
#include<stdio.h>
#include<stdlib.h>

// Structure for a pair
struct pair
{
    int a;
    int b;
};

// This function assumes that arr[] is sorted in increasing order
// according the first (or smaller) values in pairs.
int maxChainLength( struct pair arr[], int n)
{
    int i, j, max = 0;
    int *mcl = (int*) malloc ( sizeof( int ) * n );

    /* Initialize MCL (max chain length) values for all indexes */
    for ( i = 0; i < n; i++ )
        mcl[i] = 1;

    /* Compute optimized chain length values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i].a > arr[j].b && mcl[i] < mcl[j] + 1)
                mcl[i] = mcl[j] + 1;

    // mcl[i] now stores the maximum chain length ending with pair i

    /* Pick maximum of all MCL values */
    for ( i = 0; i < n; i++ )
        if ( max < mcl[i] )
            max = mcl[i];

    /* Free memory to avoid memory leak */
    free( mcl );
}

return max;
}

/* Driver program to test above function */
int main()
{
    struct pair arr[] = { {5, 24}, {15, 25},
                         {27, 40}, {50, 60} };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of maximum size chain is %d\n",
           maxChainLength( arr, n ));
    return 0;
}
```

Output:

```
Length of maximum size chain is 3
```

Time Complexity: $O(n^2)$ where n is the number of pairs.

The given problem is also a variation of [Activity Selection problem](#) and can be solved in $(n \log n)$ time. To solve it as a activity selection problem, consider the first element of a pair as start time in activity selection problem, and the second element of pair as end time. Thanks to Palash for suggesting this approach.

Find four elements that sum to a given value | Set 1 (n^3 solution)

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X. For example, if the given array is {10, 2, 3, 4, 5, 9, 7, 8} and X=23, then your function should print 3 5 7 8? ($3 + 5 + 7 + 8 = 23$).

Sources: [Find Specific Sum](#) and [Amazon Interview Question](#)

A **Naive Solution** is to generate all possible quadruples and compare the sum of every quadruple with X. The following code implements this simple method using four nested loops

```
#include <stdio.h>

/* A naive solution to print all combination of 4 elements in A[]
   with sum equal to X */
void findFourElements(int A[], int n, int X)
{
    // Fix the first element and find other three
    for (int i = 0; i < n-3; i++)
    {
        // Fix the second element and find other two
        for (int j = i+1; j < n-2; j++)
        {
            // Fix the third element and find the fourth
            for (int k = j+1; k < n-1; k++)
            {
                // find the fourth
                for (int l = k+1; l < n; l++)
                    if (A[i] + A[j] + A[k] + A[l] == X)
                        printf("%d, %d, %d, %d", A[i], A[j], A[k], A[l]);
            }
        }
    }
}

// Driver program to test above funtion
int main()
{
    int A[] = {10, 20, 30, 40, 1, 2};
    int n = sizeof(A) / sizeof(A[0]);
    int X = 91;
    findFourElements (A, n, X);
    return 0;
}
```

Output:

```
20, 30, 40, 1
```

Time Complexity: $O(n^4)$

The time complexity can be improved to $O(n^3)$ with the **use of sorting** as a preprocessing step, and then using method 1 of [this](#) post to reduce a loop.

Following are the detailed steps.

- 1) Sort the input array.
- 2) Fix the first element as $A[i]$ where i is from 0 to $n-3$. After fixing the first element of quadruple, fix the second element as $A[j]$ where j varies from $i+1$ to $n-2$. Find remaining two elements in $O(n)$ time, using the method 1 of [this](#) post

Following is C implementation of $O(n^3)$ solution.

```
# include <stdio.h>
# include <stdlib.h>

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare (const void *a, const void * b)
{ return ( *(int *)a - *(int *)b ); }

/* A sorting based solution to print all combination of 4 elements in A[]
   with sum equal to X */
void find4Numbers(int A[], int n, int X)
{
    int l, r;

    // Sort the array in increasing order, using library
    // function for quick sort
    qsort (A, n, sizeof(A[0]), compare);
```

```

/* Now fix the first 2 elements one by one and find
   the other two elements */
for (int i = 0; i < n - 3; i++)
{
    for (int j = i+1; j < n - 2; j++)
    {
        // Initialize two variables as indexes of the first and last
        // elements in the remaining elements
        l = j + 1;
        r = n-1;

        // To find the remaining two elements, move the index
        // variables (l & r) toward each other.
        while (l < r)
        {
            if( A[i] + A[j] + A[l] + A[r] == X)
            {
                printf("%d, %d, %d, %d", A[i], A[j],
                       A[l], A[r]);
                l++; r--;
            }
            else if (A[i] + A[j] + A[l] + A[r] < X)
                l++;
            else // A[i] + A[j] + A[l] + A[r] > X
                r--;
        } // end of while
    } // end of inner for loop
} // end of outer for loop
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 12};
    int X = 21;
    int n = sizeof(A)/sizeof(A[0]);
    find4Numbers(A, n, X);
    return 0;
}

```

Output:

1, 4, 6, 10

Time Complexity: $O(n^3)$

This problem can also be solved in $O(n^2 \log n)$ complexity. We will soon be publishing the $O(n^2 \log n)$ solution as a separate post.

Find four elements that sum to a given value | Set 2 (O(n^2Logn) Solution)

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X.

For example, if the given array is {10, 2, 3, 4, 5, 9, 7, 8} and X = 23, then your function should print 3 5 7 8? ($3 + 5 + 7 + 8 = 23$).

Sources: [Find Specific Sum](#) and [Amazon Interview Question](#)

We have discussed a $O(n^3)$ algorithm in [the previous post](#) on this topic. The problem can be solved in $O(n^2\log n)$ time with the help of auxiliary space.

Thanks to [itsnimish](#) for suggesting this method. Following is the detailed process.

Let the input array be A[].

1) Create an auxiliary array aux[] and store sum of all possible pairs in aux[]. The size of aux[] will be $n*(n-1)/2$ where n is the size of A[].

2) Sort the auxiliary array aux[].

3) Now the problem reduces to find two elements in aux[] with sum equal to X. We can use method 1 of [this post](#) to find the two elements efficiently. There is following important point to note though. An element of aux[] represents a pair from A[]. While picking two elements from aux[], we must check whether the two elements have an element of A[] in common. For example, if first element sum of A[1] and A[2], and second element is sum of A[2] and A[4], then these two elements of aux[] dont represent four distinct elements of input array A[].

Following is C implementation of this method.

```
#include <stdio.h>
#include <stdlib.h>

// The following structure is needed to store pair sums in aux[]
struct pairSum
{
    int first; // index (int A[]) of first element in pair
    int sec; // index of second element in pair
    int sum; // sum of the pair
};

// Following function is needed for library function qsort()
int compare (const void *a, const void * b)
{
    return ( (*(pairSum *)a).sum - (*(pairSum*)b).sum );
}

// Function to check if two given pairs have any common element or not
bool noCommon(struct pairSum a, struct pairSum b)
{
    if (a.first == b.first || a.first == b.sec ||
        a.sec == b.first || a.sec == b.sec)
        return false;
    return true;
}

// The function finds four elements with given sum X
void findFourElements (int arr[], int n, int X)
{
    int i, j;

    // Create an auxiliary array to store all pair sums
    int size = (n*(n-1))/2;
    struct pairSum aux[size];

    /* Generate all possible pairs from A[] and store sums
       of all possible pairs in aux[] */
    int k = 0;
    for (i = 0; i < n-1; i++)
    {
        for (j = i+1; j < n; j++)
        {
            aux[k].sum = arr[i] + arr[j];
            aux[k].first = i;
            aux[k].sec = j;
            k++;
        }
    }

    // Sort the aux[] array using library function for sorting
    qsort (aux, size, sizeof(aux[0]), compare);
```

```

// Now start two index variables from two corners of array
// and move them toward each other.
i = 0;
j = size-1;
while (i < size && j >=0 )
{
    if ((aux[i].sum + aux[j].sum == X) && noCommon(aux[i], aux[j]))
    {
        printf ("%d, %d, %d, %d\n", arr[aux[i].first], arr[aux[i].sec],
               arr[aux[j].first], arr[aux[j].sec]);
        return;
    }
    else if (aux[i].sum + aux[j].sum < X)
        i++;
    else
        j--;
}
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 20, 30, 40, 1, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    int X = 91;
    findFourElements (arr, n, X);
    return 0;
}

```

Output:

20, 1, 30, 40

Please note that the above code prints only one quadruple. If we remove the return statement and add statements `i++; j;`, then it prints same quadruple five times. The code can modified to print all quadruples only once. It has been kept this way to keep it simple.

Time complexity: The step 1 takes $O(n^2)$ time. The second step is sorting an array of size $O(n^2)$. Sorting can be done in $O(n^2 \log n)$ time using merge sort or heap sort or any other $O(n \log n)$ algorithm. The third step takes $O(n^2)$ time. So overall complexity is $O(n^2 \log n)$.

Auxiliary Space: $O(n^2)$. The big size of auxiliary array can be a concern in this method.

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in $O(n \log k)$ time. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: [Nearly sorted algorithm](#)

We can use **Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int A[], int size)
{
    int i, key, j;
    for (i = 1; i < size; i++)
    {
        key = A[i];
        j = i-1;

        /* Move elements of A[0..i-1], that are greater than key,
           to one position ahead of their current position.
           This loop will run at most k times */
        while (j >= 0 && A[j] > key)
        {
            A[j+1] = A[j];
            j = j-1;
        }
        A[j+1] = key;
    }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall complexity will be $O(nk)$

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.

- 1) Create a Min Heap of size $k+1$ with first $k+1$ elements. This will take $O(k)$ time (See [this GFact](#))
- 2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Log k time. So overall complexity will be $O(k) + O((n-k)*\log k)$

```
#include<iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// Given an array of size n, where every element is k away from its target
// position, sorts the array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i<=k && i<n; i++) // i < n condition is needed when k > n
```

```

        harr[i] = arr[i];
MinHeap hp(harr, k+1);

// i is index for remaining elements in arr[] and ti
// is target index of for cuurent minimum element in
// Min Heapm 'hp'.
for(int i = k+1, ti = 0; ti < n; i++, ti++)
{
    // If there are remaining elements, then place
    // root of heap at target index and add arr[i]
    // to Min Heap
    if (i < n)
        arr[ti] = hp.replaceMin(arr[i]);

    // Otherwise place root at its target index and
    // reduce heap size
    else
        arr[ti] = hp.extractMin();
}
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

```

```

    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = {2, 6, 3, 12, 56, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array\n";
    printArray (arr, n);

    return 0;
}

```

Output:

```

Following is sorted array
2 3 6 8 12 56

```

The Min Heap based method takes $O(n \log k)$ time and uses $O(k)$ auxiliary space.

We can also **use a Balanced Binary Search Tree** instead of Heap to store $K+1$ elements. The [insert](#) and [delete](#) operations on Balanced BST also take $O(\log k)$ time. So Balanced BST based method will also take $O(n \log k)$ time, but the Heap based method seems to be more efficient as the **minimum element** will always be at root. Also, Heap doesn't need extra space for left and right pointers.

Maximum circular subarray sum

Given n numbers (both +ve and -ve), arranged in a circle, find the maximum sum of consecutive number.

Examples:

Input: $a[] = \{8, -8, 9, -9, 10, -11, 12\}$
Output: 22 ($12 + 8 - 8 + 9 - 9 + 10$)

Input: $a[] = \{10, -3, -4, 7, 6, 5, -4, -1\}$
Output: 23 ($7 + 6 + 5 - 4 - 1 + 10$)

Input: $a[] = \{-1, 40, -14, 7, 6, 5, -4, -1\}$
Output: 52 ($7 + 6 + 5 - 4 - 1 - 1 + 40$)

There can be two cases for the maximum sum:

Case 1: The elements that contribute to the maximum sum are arranged such that no wrapping is there. Examples: $\{-10, 2, -1, 5\}$, $\{-2, 4, -1, 4, -1\}$. In this case, [Kadane's algorithm](#) will produce the result.

Case 2: The elements which contribute to the maximum sum are arranged such that wrapping is there. Examples: $\{10, -12, 11\}$, $\{12, -5, 4, -8, 11\}$. In this case, we change wrapping to non-wrapping. Let us see how. Wrapping of contributing elements implies non wrapping of non contributing elements, so find out the sum of non contributing elements and subtract this sum from the total sum. To find out the sum of non contributing, invert sign of each element and then run Kadanes algorithm.

Our array is like a ring and we have to eliminate the maximum continuous negative that implies maximum continuous positive in the inverted arrays.

Finally we compare the sum obtained by both cases, and return the maximum of the two sums.

Thanks to [ashishdew0](#) for suggesting this solution. Following is C implementation of the above method.

```
// Program for maximum contiguous circular sum problem
#include<stdio.h>

// Standard Kadane's algorithm to find maximum subarray sum
int kadane (int a[], int n)
{
    // Case 1: get the maximum sum using standard kadane's algorithm
    int max_kadane = kadane(a, n);

    // Case 2: Now find the maximum sum that includes corner elements.
    int max_wrap = 0, i;
    for(i=0; i<n; i++)
    {
        max_wrap += a[i]; // Calculate array-sum
        a[i] = -a[i]; // invert the array (change sign)
    }

    // max sum with corner elements will be:
    // array-sum - (-max subarray sum of inverted array)
    max_wrap = max_wrap + kadane(a, n);

    // The maximum circular sum will be maximum of two sums
    return (max_wrap > max_kadane)? max_wrap: max_kadane;
}

// Standard Kadane's algorithm to find maximum subarray sum
// See http://www.geeksforgeeks.org/archives/576 for details
int kadane (int a[], int n)
{
    int max_so_far = 0, max_ending_here = 0;
    int i;
    for(i = 0; i < n; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if(max_ending_here < 0)
            max_ending_here = 0;
        if(max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

/* Driver program to test maxCircularSum() */
int main()
```

```
{  
    int a[] = {11, 10, -20, 5, -3, -5, 8, -13, 10};  
    int n = sizeof(a)/sizeof(a[0]);  
    printf("Maximum circular sum is %d\n", maxCircularSum(a, n));  
    return 0;  
}
```

Output:

Maximum circular sum is 31

Time Complexity: O(n) where n is the number of elements in input array.

Note that the above algorithm doesn't work if all numbers are negative e.g., {-1, -2, -3}. It returns 0 in this case. This case can be handled by adding a pre-check to see if all the numbers are negative before running the above algorithm.

Find the row with maximum number of 1s

Given a boolean 2D array, where each row is sorted. Find the row with the maximum number of 1s.

Example

```
Input matrix
0 1 1 1
0 0 1 1
1 1 1 1 // this row has maximum 1s
0 0 0 0
```

Output: 2

A **simple method** is to do a row wise traversal of the matrix, count the number of 1s in each row and compare the count with max. Finally, return the index of row with maximum 1s. The time complexity of this method is $O(m*n)$ where m is number of rows and n is number of columns in matrix.

We can do better. Since each row is sorted, we can **use Binary Search** to count of 1s in each row. We find the index of first instance of 1 in each row. The count of 1s will be equal to total number of columns minus the index of first 1.

See the following code for implementation of the above approach.

```
#include <stdio.h>
#define R 4
#define C 4

/* A function to find the index of first index of 1 in a boolean array arr[] */
int first(bool arr[], int low, int high)
{
    if(high >= low)
    {
        // get the middle index
        int mid = low + (high - low)/2;

        // check if the element at middle index is first 1
        if ((mid == 0 || arr[mid-1] == 0) && arr[mid] == 1)
            return mid;

        // if the element is 0, recur for right side
        else if (arr[mid] == 0)
            return first(arr, (mid + 1), high);

        else // If element is not first 1, recur for left side
            return first(arr, low, (mid -1));
    }
    return -1;
}

// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    int max_row_index = 0, max = -1; // Initialize max values

    // Traverse for each row and count number of 1s by finding the index
    // of first 1
    int i, index;
    for (i = 0; i < R; i++)
    {
        index = first (mat[i], 0, C-1);
        if (index != -1 && C-index > max)
        {
            max = C - index;
            max_row_index = i;
        }
    }

    return max_row_index;
}

/* Driver program to test above functions */
int main()
{
    bool mat[R][C] = { {0, 0, 0, 1},
                      {0, 1, 1, 1},
                      {1, 1, 1, 1},
                      {0, 0, 0, 0}
                    };

    printf("Index of row with maximum 1s is %d \n", rowWithMax1s(mat));
```

```
    return 0;
}
```

Output:

```
Index of row with maximum 1s is 2
```

Time Complexity: O(mLogn) where m is number of rows and n is number of columns in matrix.

The above solution **can be optimized further**. Instead of doing binary search in every row, we first check whether the row has more 1s than max so far. If the row has more 1s, then only count 1s in the row. Also, to count 1s in a row, we dont do binary search in complete row, we do search in before the index of last max.

Following is an optimized version of the above solution.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    int i, index;

    // Initialize max using values from first row.
    int max_row_index = 0;
    int max = C - first(mat[0], 0, C-1);

    // Traverse for each row and count number of 1s by finding the index
    // of first 1
    for (i = 1; i < R; i++)
    {
        // Count 1s in this row only if this row has more 1s than
        // max so far
        if (mat[i][C-max-1] == 1)
        {
            // Note the optimization here also
            index = first (mat[i], 0, C-max);

            if (index != -1 && C-index > max)
            {
                max = C - index;
                max_row_index = i;
            }
        }
    }
    return max_row_index;
}
```

The worst case time complexity of the above optimized version is also O(mLogn), the will solution work better on average. Thanks to [Naveen Kumar Singh](#) for suggesting the above solution.

Sources: [this](#) and [this](#)

The worst case of the above solution occurs for a matrix like following

```
0 0 0 0 1
0 0 0 ..0 1 1
0 0 1 1 1
.0 1 1 1 1
```

Following method works in O(m+n) time complexity in worst case.

Step1: Get the index of first (or leftmost) 1 in the first row.

Step2: Do following for every row after the first row

IF the element on left of previous leftmost 1 is 0, ignore this row.

ELSE Move left until a 0 is found. Update the leftmost index to this index and max_row_index to be the current row.

The time complexity is O(m+n) because we can possibly go as far left as we came ahead in the first step.

Following is C++ implementation of this method.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    // Initialize first row as row with max 1s
    int max_row_index = 0;

    // The function first() returns index of first 1 in row 0.
```

```
// Use this index to initialize the index of leftmost 1 seen so far
int j = first(mat[0], 0, C-1) - 1;
if (j == -1) // if 1 is not present in first row
    j = C - 1;

for (int i = 1; i < R; i++)
{
    // Move left until a 0 is found
    while (j >= 0 && mat[i][j] == 1)
    {
        j = j-1; // Update the index of leftmost 1 seen so far
        max_row_index = i; // Update max_row_index
    }
}
return max_row_index;
}
```

Thanks to Tylor, Ankan and Palash for their inputs.

Median of two sorted arrays of different sizes

This is an extension of [median of two sorted arrays of equal size](#) problem. Here we handle arrays of unequal size also.

The approach discussed in this post is similar to method 2 of equal size post. The basic idea is same, we find the median of two arrays and compare the medians to discard almost half of the elements in both arrays. Since the number of elements may differ here, there are many base cases that need to be handled separately. Before we proceed to complete solution, let us first talk about all base cases.

Let the two arrays be A[N] and B[M]. In the following explanation, it is assumed that N is smaller than or equal to M.

Base cases:

The smaller array has only one element

Case 1: N = 1, M = 1.

Case 2: N = 1, M is odd

Case 3: N = 1, M is even

The smaller array has only two elements

Case 4: N = 2, M = 2

Case 5: N = 2, M is odd

Case 6: N = 2, M is even

Case 1: There is only one element in both arrays, so output the average of A[0] and B[0].

Case 2: N = 1, M is odd

Let B[5] = {5, 10, 12, 15, 20}

First find the middle element of B[], which is 12 for above array. There are following 4 sub-cases.

2.1 If A[0] is smaller than 10, the median is average of 10 and 12.

2.2 If A[0] lies between 10 and 12, the median is average of A[0] and 12.

2.3 If A[0] lies between 12 and 15, the median is average of 12 and A[0].

2.4 If A[0] is greater than 15, the median is average of 12 and 15.

In all the sub-cases, we find that 12 is fixed. So, we need to find the median of B[M / 2], B[M / 2 + 1], A[0] and take its average with B[M / 2].

Case 3: N = 1, M is even

Let B[4] = {5, 10, 12, 15}

First find the middle items in B[], which are 10 and 12 in above example. There are following 3 sub-cases.

3.1 If A[0] is smaller than 10, the median is 10.

3.2 If A[0] lies between 10 and 12, the median is A[0].

3.3 If A[0] is greater than 12, the median is 12.

So, in this case, find the median of three elements B[M / 2], B[M / 2] and A[0].

Case 4: N = 2, M = 2

There are four elements in total. So we find the median of 4 elements.

Case 5: N = 2, M is odd

Let B[5] = {5, 10, 12, 15, 20}

The median is given by median of following three elements: B[M/2], max(A[0], B[M/2 1]), min(A[1], B[M/2 + 1]).

Case 6: N = 2, M is even

Let B[4] = {5, 10, 12, 15}

The median is given by median of following four elements: B[M/2], B[M/2 1], max(A[0], B[M/2 2]), min(A[1], B[M/2 + 1])

Remaining Cases:

Once we have handled the above base cases, following is the remaining process.

1) Find the middle item of A[] and middle item of B[].

.1.1) If the middle item of A[] is greater than middle item of B[], ignore the last half of A[], let length of ignored part is idx. Also, cut down B[] by idx from the start.

.1.2) else, ignore the first half of A[], let length of ignored part is idx. Also, cut down B[] by idx from the last.

Following is C implementation of the above approach.

```
// A C program to find median of two sorted arrays of unequal size
#include <stdio.h>
#include <stdlib.h>

// A utility function to find maximum of two integers
int max( int a, int b )
{ return a > b ? a : b; }

// A utility function to find minimum of two integers
```

```

int min( int a, int b )
{ return a < b ? a : b; }

// A utility function to find median of two integers
float MO2( int a, int b )
{ return ( a + b ) / 2.0; }

// A utility function to find median of three integers
float MO3( int a, int b, int c )
{
    return a + b + c - max( a, max( b, c ) )
           - min( a, min( b, c ) );
}

// A utility function to find median of four integers
float MO4( int a, int b, int c, int d )
{
    int Max = max( a, max( b, max( c, d ) ) );
    int Min = min( a, min( b, min( c, d ) ) );
    return ( a + b + c + d - Max - Min ) / 2.0;
}

// This function assumes that N is smaller than or equal to M
float findMedianUtil( int A[], int N, int B[], int M )
{
    // If the smaller array has only one element
    if( N == 1 )
    {
        // Case 1: If the larger array also has one element, simply call MO2()
        if( M == 1 )
            return MO2( A[0], B[0] );

        // Case 2: If the larger array has odd number of elements, then consider
        // the middle 3 elements of larger array and the only element of
        // smaller array. Take few examples like following
        // A = {9}, B[] = {5, 8, 10, 20, 30} and
        // A[] = {1}, B[] = {5, 8, 10, 20, 30}
        if( M & 1 )
            return MO2( B[M/2], MO3(A[0], B[M/2 - 1], B[M/2 + 1]) );

        // Case 3: If the larger array has even number of element, then median
        // will be one of the following 3 elements
        // ... The middle two elements of larger array
        // ... The only element of smaller array
        return MO3( B[M/2], B[M/2 - 1], A[0] );
    }

    // If the smaller array has two elements
    else if( N == 2 )
    {
        // Case 4: If the larger array also has two elements, simply call MO4()
        if( M == 2 )
            return MO4( A[0], A[1], B[0], B[1] );

        // Case 5: If the larger array has odd number of elements, then median
        // will be one of the following 3 elements
        // 1. Middle element of larger array
        // 2. Max of first element of smaller array and element just
        //     before the middle in bigger array
        // 3. Min of second element of smaller array and element just
        //     after the middle in bigger array
        if( M & 1 )
            return MO3( B[M/2],
                        max( A[0], B[M/2 - 1] ),
                        min( A[1], B[M/2 + 1] )
                      );

        // Case 6: If the larger array has even number of elements, then
        // median will be one of the following 4 elements
        // 1) & 2) The middle two elements of larger array
        // 3) Max of first element of smaller array and element
        //     just before the first middle element in bigger array
        // 4. Min of second element of smaller array and element
        //     just after the second middle in bigger array
        return MO4( B[M/2],
                    B[M/2 - 1],
                    max( A[0], B[M/2 - 2] ),
                    min( A[1], B[M/2 + 1] )
                  );
    }
}

```

```

int idxA = ( N - 1 ) / 2;
int idxB = ( M - 1 ) / 2;

/* if A[idxA] <= B[idxB], then median must exist in
   A[idxA....] and B[....idxB] */
if( A[idxA] <= B[idxB] )
    return findMedianUtil( A + idxA, N / 2 + 1, B, M - idxA );

/* if A[idxA] > B[idxB], then median must exist in
   A[...idxA] and B[idxB....] */
return findMedianUtil( A, N / 2 + 1, B + idxA, M - idxA );
}

// A wrapper function around findMedianUtil(). This function makes
// sure that smaller array is passed as first argument to findMedianUtil
float findMedian( int A[], int N, int B[], int M )
{
    if ( N > M )
        return findMedianUtil( B, M, A, N );

    return findMedianUtil( A, N, B, M );
}

// Driver program to test above functions
int main()
{
    int A[] = {900};
    int B[] = {5, 8, 10, 20};

    int N = sizeof(A) / sizeof(A[0]);
    int M = sizeof(B) / sizeof(B[0]);

    printf( "%f", findMedian( A, N, B, M ) );
    return 0;
}

```

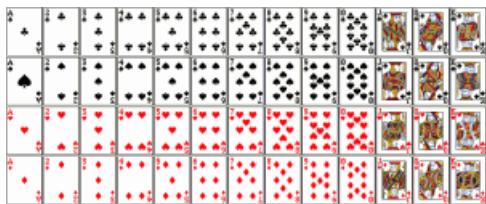
Output:

10

Time Complexity: O(LogM + LogN)

Shuffle a given array

Given an array, write a program to generate a random permutation of array elements. This question is also asked as shuffle a deck of cards or randomize a given array.



Let the given array be $arr[]$. A simple solution is to create an auxiliary array $temp[]$ which is initially a copy of $arr[]$. Randomly select an element from $temp[]$, copy the randomly selected element to $arr[0]$ and remove the selected element from $temp[]$. Repeat the same process n times and keep copying elements to $arr[1], arr[2], \dots$. The time complexity of this solution will be $O(n^2)$.

[FisherYates shuffle Algorithm](#) works in $O(n)$ time complexity. The assumption here is, we are given a function `rand()` that generates random number in $O(1)$ time.

The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to $n-2$ (size reduced by 1), and repeat the process till we hit the first element.

Following is the detailed algorithm

```
To shuffle an array a of n elements (indices 0..n-1):
for i from n - 1 downto 1 do
    j = random integer with 0 <= j <= i
    exchange a[j] and a[i]
```

Following is C++ implementation of this algorithm

```
// C Program to shuffle a given array

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// A utility function to swap two integers
void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A utility function to print an array
void printArray (int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// A function to generate a random permutation of arr[]
void randomize ( int arr[], int n )
{
    // Use a different seed value so that we don't get same
    // result each time we run this program
    srand ( time(NULL) );

    // Start from the last element and swap one by one. We don't
    // need to run for the first element that's why i > 0
    for (int i = n-1; i > 0; i--)
    {
        // Pick a random index from 0 to i
        int j = rand() % (i+1);

        // Swap arr[i] with the element at random index
        swap(&arr[i], &arr[j]);
    }
}

// Driver program to test above function.
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n = sizeof(arr)/ sizeof(arr[0]);
```

```
randomize (arr, n);  
printArray(arr, n);  
  
    return 0;  
}
```

Output:

```
7 8 4 6 3 1 2 5
```

The above function assumes that rand() generates a random number.

Time Complexity: O(n), assuming that the function rand() takes O(1) time.

How does this work?

The probability that i th element (including the last one) goes to last position is $1/n$, because we randomly pick an element in first iteration.

The probability that i th element goes to second last position can be proved to be $1/n$ by dividing it in two cases.

Case 1: $i = n-1$ (index of last element):

The probability of last element going to second last position is = (probability that last element doesn't stay at its original position) \times (probability that the index picked in previous step is picked again so that the last element is swapped)

So the probability = $((n-1)/n) \times (1/(n-1)) = 1/n$

Case 2: $0 < i < n-1$ (index of non-last):

The probability of i th element going to second position = (probability that i th element is not picked in previous iteration) \times (probability that i th element is picked in this iteration)

So the probability = $((n-1)/n) \times (1/(n-1)) = 1/n$

We can easily generalize above proof for any other position.

Count the number of possible triangles

Given an unsorted array of positive integers. Find the number of triangles that can be formed with three different array elements as three sides of triangles. For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).

For example, if the input array is {4, 6, 3, 7}, the output should be 3. There are three triangles possible {3, 4, 6}, {4, 6, 7} and {3, 6, 7}. Note that {3, 4, 7} is not a possible triangle.

As another example, consider the array {10, 21, 22, 100, 101, 200, 300}. There can be 6 possible triangles: {10, 21, 22}, {21, 100, 101}, {22, 100, 101}, {10, 100, 101}, {100, 101, 200} and {101, 200, 300}

Method 1 (Brute force)

The brute force method is to run three loops and keep track of the number of triangles possible so far. The three loops select three different values from array, the innermost loop checks for the triangle property (the sum of any two sides must be greater than the value of third side).

Time Complexity: $O(N^3)$ where N is the size of input array.

Method 2 (Tricky and Efficient)

Let a, b and c be three sides. The below condition must hold for a triangle (Sum of two sides is greater than the third side)

- i) $a + b > c$
- ii) $b + c > a$
- iii) $a + c > b$

Following are steps to count triangle.

1. Sort the array in non-decreasing order.

2. Initialize two pointers i and j to first and second elements respectively, and initialize count of triangles as 0.

3. Fix i and j and find the rightmost index k (or largest arr[k]) such that $arr[i] + arr[j] > arr[k]$. The number of triangles that can be formed with $arr[i]$ and $arr[j]$ as two sides is $k - j$. Add $k - j$ to count of triangles.

Let us consider $arr[i]$ as a, $arr[j]$ as b and all elements between $arr[j+1]$ and $arr[k]$ as c. The above mentioned conditions (ii) and (iii) are satisfied because $arr[i] < arr[j] < arr[k]$. And we check for condition (i) when we pick 'k'. 4. Increment j to fix the second element again.

Note that in step 3, we can use the previous value of k. The reason is simple, if we know that the value of $arr[i] + arr[j-1]$ is greater than $arr[k]$, then we can say $arr[i] + arr[j]$ will also be greater than $arr[k]$, because the array is sorted in increasing order.

5. If j has reached end, then increment i. Initialize j as $i + 1$? , k as $i + 2$? and repeat the steps 3 and 4.

Following is implementation of the above approach.

```
// Program to count number of triangles that can be formed from given array
#include <stdio.h>
#include <stdlib.h>

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int comp(const void* a, const void* b)
{ return *(int*)a > *(int*)b ; }

// Function to count all possible triangles with arr[] elements
int findNumberOfTriangles(int arr[], int n)
{
    // Sort the array elements in non-decreasing order
    qsort(arr, n, sizeof( arr[0] ), comp);

    // Initialize count of triangles
    int count = 0;

    // Fix the first element. We need to run till n-3 as the other two elements are
    // selected from arr[i+1...n-1]
    for (int i = 0; i < n-2; ++i)
    {
        // Initialize index of the rightmost third element
        int k = i+2;

        // Fix the second element
        for (int j = i+1; j < n; ++j)
        {
            // Find the rightmost element which is smaller than the sum
            // of two fixed elements
            // The important thing to note here is, we use the previous
            // value of k. If value of arr[i] + arr[j-1] was greater than arr[k],
            // then arr[i] + arr[j] must be greater than k, because the
            // array is sorted.
        }
    }
}
```

```

        while (k < n && arr[i] + arr[j] > arr[k])
            ++k;

        // Total number of possible triangles that can be formed
        // with the two fixed elements is k - j - 1. The two fixed
        // elements are arr[i] and arr[j]. All elements between arr[j+1]
        // to arr[k-1] can form a triangle with arr[i] and arr[j].
        // One is subtracted from k because k is incremented one extra
        // in above while loop.
        // k will always be greater than j. If j becomes equal to k, then
        // above loop will increment k, because arr[k] + arr[i] is always
        // greater than arr[k]
        count += k - j - 1;
    }
}

return count;
}

// Driver program to test above functionarr[j+1]
int main()
{
    int arr[] = {10, 21, 22, 100, 101, 200, 300};
    int size = sizeof( arr ) / sizeof( arr[0] );

    printf("Total number of triangles possible is %d ",
           findNumberOfTriangles( arr, size ) );

    return 0;
}

```

Output:

Total number of triangles possible is 6

Time Complexity: $O(n^2)$. The time complexity looks more because of 3 nested loops. If we take a closer look at the algorithm, we observe that k is initialized only once in the outermost loop. The innermost loop executes at most $O(n)$ time for every iteration of outer most loop, because k starts from $i+2$ and goes upto n for all values of j . Therefore, the time complexity is $O(n^2)$.

Source: <http://stackoverflow.com/questions/8110538/total-number-of-possible-triangles-from-n-numbers>

Iterative Quick Sort

Following is a typical recursive implementation of [Quick Sort](#) that uses last element as pivot.

```
/* A typical recursive implementation of quick sort */

/* This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

The above implementation can be optimized in many ways

- 1) The above implementation uses last index as pivot. This causes worst-case behavior on already sorted arrays, which is a commonly occurring case. The problem can be solved by choosing either a random index for the pivot, or choosing the middle index of the partition or choosing the median of the first, middle and last element of the partition for the pivot. (See [this](#) for details)
- 2) To reduce the recursion depth, recur first for the smaller half of the array, and use a tail call to recurse into the other.
- 3) Insertion sort works better for small subarrays. Insertion sort can be used for invocations on such small arrays (i.e. where the length is less than a threshold t determined experimentally). For example, [this](#) library implementation of qsort uses insertion sort below size 7.

Despite above optimizations, the function remains recursive and uses [function call stack](#) to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution.

The above function can be easily converted to iterative version with the help of an auxiliary stack. Following is an iterative implementation of the above recursive code.

```
// An iterative implementation of quick sort
#include <stdio.h>

// A utility function to swap two elements
void swap ( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function is same in both iterative and recursive*/
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
}
```

```

        swap (&arr[i], &arr[j]);
    }
}
swap (&arr[i + 1], &arr[h]);
return (i + 1);
}

/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */
void quickSortIterative (int arr[], int l, int h)
{
    // Create an auxiliary stack
    int stack[ h - l + 1 ];

    // initialize top of stack
    int top = -1;

    // push initial values of l and h to stack
    stack[ ++top ] = l;
    stack[ ++top ] = h;

    // Keep popping from stack while is not empty
    while ( top >= 0 )
    {
        // Pop h and l
        h = stack[ top-- ];
        l = stack[ top-- ];

        // Set pivot element at its correct position in sorted array
        int p = partition( arr, l, h );

        // If there are elements on left side of pivot, then push left
        // side to stack
        if ( p-1 > l )
        {
            stack[ ++top ] = l;
            stack[ ++top ] = p - 1;
        }

        // If there are elements on right side of pivot, then push right
        // side to stack
        if ( p+1 < h )
        {
            stack[ ++top ] = p + 1;
            stack[ ++top ] = h;
        }
    }
}

// A utility function to print contents of arr
void printArr( int arr[], int n )
{
    int i;
    for ( i = 0; i < n; ++i )
        printf( "%d ", arr[i] );
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};
    int n = sizeof( arr ) / sizeof( *arr );
    quickSortIterative( arr, 0, n - 1 );
    printArr( arr, n );
    return 0;
}

```

Output:

1 2 2 3 3 3 4 5

The above mentioned optimizations for recursive quick sort can also be applied to iterative version.

- 1) Partition process is same in both recursive and iterative. The same techniques to choose optimal pivot can also be applied to iterative version.
- 2) To reduce the stack size, first push the indexes of smaller half.
- 3) Use insertion sort when the size reduces below a experimentally calculated threshold.

References:

<http://en.wikipedia.org/wiki/Quicksort>

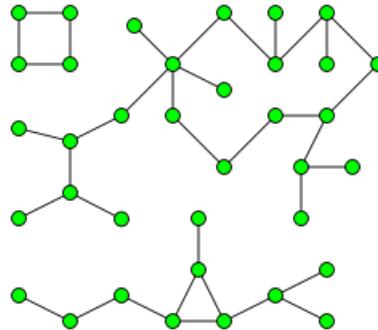
Find the number of islands

Given a boolean 2D matrix, find the number of islands.

This is an variation of the standard problem Counting number of connected components in a undirected graph.

Before we go to the problem, let us understand what is a connected component. A [connected component](#) of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.

For example, the graph shown below has three connected components.



A graph where all vertices are connected with each other, has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},  
{0, 1, 0, 0, 1},  
{1, 0, 0, 1, 1},  
{0, 0, 0, 0, 0},  
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbors. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursive call for 8 neighbors only. We keep track of the visited 1s so that they are not visited again.

C/C++

```
// Program to count islands in boolean 2D matrix  
#include <stdio.h>  
#include <string.h>  
#include <stdbool.h>  
  
#define ROW 5  
#define COL 5  
  
// A function to check if a given cell (row, col) can be included in DFS  
int isSafe(int M[][COL], int row, int col, bool visited[] [COL])  
{  
    // row number is in range, column number is in range and value is 1  
    // and not yet visited  
    return (row >= 0) && (row < ROW) &&  
        (col >= 0) && (col < COL) &&  
        (M[row][col] && !visited[row][col]);  
}  
  
// A utility function to do DFS for a 2D boolean matrix. It only considers  
// the 8 neighbours as adjacent vertices  
void DFS(int M[][] [COL], int row, int col, bool visited[] [COL])  
{  
    // These arrays are used to get row and column numbers of 8 neighbours  
    // of a given cell  
    static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};  
    static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};  
  
    // Mark this cell as visited  
    visited[row][col] = true;  
  
    // Recur for all connected neighbours  
    for (int k = 0; k < 8; ++k)  
    {  
        int r = row + rowNbr[k];  
        int c = col + colNbr[k];  
        if (isSafe(M, r, c, visited))  
            DFS(M, r, c, visited);  
    }  
}
```

```

        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
    }

// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[][])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW][COL];
    memset(visited, 0, sizeof(visited));

    // Initialize count as 0 and traverse through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j] && !visited[i][j]) // If a cell with value 1 is not
            {                                // visited yet, then new island found
                DFS(M, i, j, visited);      // Visit all cells in this island.
                ++count;                  // and increment island count
            }

    return count;
}

// Driver program to test above function
int main()
{
    int M[][][COL]= { {1, 1, 0, 0, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 0, 1, 1},
                      {0, 0, 0, 0, 0},
                      {1, 0, 1, 0, 1}
    };

    printf("Number of islands is: %d\n", countIslands(M));
    return 0;
}

```

Java

```

// Java program to count islands in boolean 2D matrix
import java.util.*;
import java.lang.*;
import java.io.*;

class Islands
{
    //No of rows and columns
    static final int ROW = 5, COL = 5;

    // A function to check if a given cell (row, col) can
    // be included in DFS
    boolean isSafe(int M[][], int row, int col,
                  boolean visited[][])
    {
        // row number is in range, column number is in range
        // and value is 1 and not yet visited
        return (row >= 0) && (row < ROW) &&
               (col >= 0) && (col < COL) &&
               (M[row][col]==1 && !visited[row][col]);
    }

    // A utility function to do DFS for a 2D boolean matrix.
    // It only considers the 8 neighbors as adjacent vertices
    void DFS(int M[][], int row, int col, boolean visited[][])
    {
        // These arrays are used to get row and column numbers
        // of 8 neighbors of a given cell
        int rowNbr[] = new int[] {-1, -1, -1, 0, 0, 1, 1, 1};
        int colNbr[] = new int[] {-1, 0, 1, -1, 1, -1, 0, 1};

        // Mark this cell as visited
        visited[row][col] = true;

        // Recur for all connected neighbours
        for (int k = 0; k < 8; ++k)

```

```

        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
    }

// The main function that returns count of islands in a given
// boolean 2D matrix
int countIslands(int M[][])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    boolean visited[][] = new boolean[ROW][COL];

    // Initialize count as 0 and traverse through the all cells
    // of given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j]==1 && !visited[i][j]) // If a cell with
            {                                // value 1 is not
                // visited yet, then new island found, Visit all
                // cells in this island and increment island count
                DFS(M, i, j, visited);
                ++count;
            }
    return count;
}

// Driver method
public static void main (String[] args) throws java.lang.Exception
{
    int M[][]= new int[][] {{1, 1, 0, 0, 0},
                           {0, 1, 0, 0, 1},
                           {1, 0, 0, 1, 1},
                           {0, 0, 0, 0, 0},
                           {1, 0, 1, 0, 1}
                           };
    Islands I = new Islands();
    System.out.println("Number of islands is: "+ I.countIslands(M));
}
} //Contributed by Aakash Hasija

```

Output:

Number of islands is: 5

Time complexity: O(ROW x COL)

Reference:

http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29

Construction of Longest Monotonically Increasing Subsequence (N log N)

In my previous post, I have explained about longest [monotonically increasing sub-sequence](#)(LIS) problem in detail. However, the post only covered code related to querying size of LIS, but not the construction of LIS. I left it as an exercise. If you have solved, cheers. If not, you are not alone, here is code.

If you have not read my previous post, read [here](#). Note that the below code prints LIS in reverse order. We can modify print order using a stack (explicit or system stack). I am leaving explanation as an exercise (easy).

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;

// Binary search
int GetCeilIndex(int A[], int T[], int l, int r, int key) {
    int m;

    while( r - l > 1 ) {
        m = l + (r - l)/2;
        if( A[T[m]] >= key )
            r = m;
        else
            l = m;
    }

    return r;
}

int LongestIncreasingSubsequence(int A[], int size) {
    // Add boundary case, when array size is zero
    // Depend on smart pointers

    int *tailIndices = new int[size];
    int *prevIndices = new int[size];
    int len;

    memset(tailIndices, 0, sizeof(tailIndices[0])*size);
    memset(prevIndices, 0xFF, sizeof(prevIndices[0])*size);

    tailIndices[0] = 0;
    prevIndices[0] = -1;
    len = 1; // it will always point to empty location
    for( int i = 1; i < size; i++ ) {
        if( A[i] < A[tailIndices[0]] ) {
            // new smallest value
            tailIndices[0] = i;
        } else if( A[i] > A[tailIndices[len-1]] ) {
            // A[i] wants to extend largest subsequence
            prevIndices[i] = tailIndices[len-1];
            tailIndices[len++] = i;
        } else {
            // A[i] wants to be a potential candidate of future subsequence
            // It will replace ceil value in tailIndices
            int pos = GetCeilIndex(A, tailIndices, -1, len-1, A[i]);

            prevIndices[i] = tailIndices[pos-1];
            tailIndices[pos] = i;
        }
    }
    cout << "LIS of given input" << endl;
    for( int i = tailIndices[len-1]; i >= 0; i = prevIndices[i] )
        cout << A[i] << " ";
    cout << endl;

    delete[] tailIndices;
    delete[] prevIndices;

    return len;
}

int main() {
    int A[] = { 2, 5, 3, 7, 11, 8, 10, 13, 6 };
    int size = sizeof(A)/sizeof(A[0]);

    printf("LIS size %d\n", LongestIncreasingSubsequence(A, size));

    return 0;
}
```

Exercises:

1. You know [Kadane's](#) algorithm to find [maximum sum sub-array](#). Modify Kadane's algorithm to trace starting and ending location of maximum sum sub-array.
2. Modify [Kadane's](#) algorithm to find maximum sum sub-array in a circular array. Refer GFG forum for many comments on the question.
3. Given two integers A and B as input. Find number of Fibonacci numbers existing in between these two numbers (including A and B). For example, A = 3 and B = 18, there are 4 Fibonacci numbers in between {3,5, 8, 13}. Do it in O(log K) time, where K is max(A, B). What is your observation?

Find the first circular tour that visits all petrol pumps

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be start = 1? (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).

We can **use a Queue** to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeuing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

```
// C program to find circular tour for a truck
#include <stdio.h>

// A petrol pump has petrol and distance to next petrol pump
struct petrolPump
{
    int petrol;
    int distance;
};

// The function returns starting point if there is a possible solution,
// otherwise returns -1
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end = 1;

    int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
     And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If current amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
            start = (start + 1)%n;

            // If 0 is being considered as start again, then there is no
            // possible solution
            if (start == 0)
                return -1;
        }

        // Add a petrol pump to current tour
        curr_petrol += arr[end].petrol - arr[end].distance;

        end = (end + 1)%n;
    }

    // Return starting point
    return start;
}

// Driver program to test above functions
int main()
{
    struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
int start = printTour(arr, n);

(start == -1)? printf("No solution"): printf("Start = %d", start);

return 0;
}
```

Output:

```
start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is $O(n)$.

Auxiliary Space: $O(1)$

Arrange given numbers to form the biggest number

Given an array of numbers, arrange them in a way that yields the largest value. For example, if the given numbers are {54, 546, 548, 60}, the arrangement 6054854654 gives the largest value. And if the given numbers are {1, 34, 3, 98, 9, 76, 45, 4}, then the arrangement 998764543431 gives the largest value.

A simple solution that comes to our mind is to sort all numbers in descending order, but simply sorting doesn't work. For example, 548 is greater than 60, but in output 60 comes before 548. As a second example, 98 is greater than 9, but 9 comes before 98 in output.

So how do we go about it? The idea is to use any comparison based sorting algorithm. In the used sorting algorithm, instead of using the default comparison, write a comparison function `myCompare()` and use it to sort numbers. Given two numbers X and Y, how should `myCompare()` decide which number to put first we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y). If XY is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

Following is C++ implementation of the above approach. To keep the code simple, numbers are considered as strings, and `vector` is used instead of normal array.

```
// Given an array of numbers, program to arrange the numbers to form the
// largest number
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

// A comparison function which is used by sort() in printLargest()
int myCompare(string X, string Y)
{
    // first append Y at the end of X
    string XY = X.append(Y);

    // then append X at the end of Y
    string YX = Y.append(X);

    // Now see which of the two formed numbers is greater
    return XY.compare(YX) > 0 ? 1 : 0;
}

// The main function that prints the arrangement with the largest value.
// The function accepts a vector of strings
void printLargest(vector<string> arr)
{
    // Sort the numbers using library sort function. The function uses
    // our comparison function myCompare() to compare two strings.
    // See http://www.cplusplus.com/reference/algorithm/sort/ for details
    sort(arr.begin(), arr.end(), myCompare);

    for (int i=0; i < arr.size(); i++)
        cout << arr[i];
}

// driver program to test above functions
int main()
{
    vector<string> arr;

    // output should be 6054854654
    arr.push_back("54");
    arr.push_back("546");
    arr.push_back("548");
    arr.push_back("60");
    printLargest(arr);

    // output should be 77776
    /*arr.push_back("7");
    arr.push_back("776");
    arr.push_back("7");
    arr.push_back("7");*/

    // output should be 998764543431
    /*arr.push_back("1");
    arr.push_back("34");
    arr.push_back("3");
    arr.push_back("98");
    arr.push_back("9");
    arr.push_back("76");
```

```
arr.push_back("45");
arr.push_back("4");
*/
return 0;
}
```

Output:

6054854654

Pancake sorting

Given an unsorted array, sort the given array. You are allowed to do only following operation on array.

```
flip(arr, i): Reverse array from 0 to i
```

Unlike a traditional sorting algorithm, which attempts to sort with the fewest comparisons possible, the goal is to sort the sequence in as few reversals as possible.

The idea is to do something similar to [Selection Sort](#). We one by one place maximum element at the end and reduce the size of current array by one.

Following are the detailed steps. Let given array be arr[] and size of array be n.

1) Start from current size equal to n and reduce current size by one while its greater than 1. Let the current size be curr_size. Do following for every curr_size

- a) Find index of the maximum element in arr[0..curr_size-1]. Let the index be mi
- b) Call flip(arr, mi)
- c) Call flip(arr, curr_size-1)

See following video for visualization of the above algorithm.

<http://www.youtube.com/embed/kk-DDgoXfk>

C

```
/* A C++ program for Pancake Sorting */
#include <stdlib.h>
#include <stdio.h>

/* Reverses arr[0..i] */
void flip(int arr[], int i)
{
    int temp, start = 0;
    while (start < i)
    {
        temp = arr[start];
        arr[start] = arr[i];
        arr[i] = temp;
        start++;
        i--;
    }
}

/* Returns index of the maximum element in arr[0..n-1] */
int findMax(int arr[], int n)
{
    int mi, i;
    for (mi = 0, i = 0; i < n; ++i)
        if (arr[i] > arr[mi])
            mi = i;
    return mi;
}

// The main function that sorts given array using flip
// operations
int pancakeSort(int *arr, int n)
{
    // Start from the complete array and one by one reduce
    // current size by one
    for (int curr_size = n; curr_size > 1; --curr_size)
    {
        // Find index of the maximum element in
        // arr[0..curr_size-1]
        int mi = findMax(arr, curr_size);

        // Move the maximum element to end of current array
        // if it's not already at the end
        if (mi != curr_size-1)
        {
            // To move at the end, first move maximum number
            // to beginning
            flip(arr, mi);

            // Now move the maximum number to end by reversing
            // current array
            flip(arr, curr_size-1);
        }
    }
}
```

```

        }
    }

/* A utility function to print an array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
    int arr[] = {23, 10, 20, 11, 12, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    pancakeSort(arr, n);

    puts("Sorted Array ");
    printArray(arr, n);

    return 0;
}

```

Java

```

/* Java program to sort array using pancake sort */
import java.io.*;

class PancakeSort {

    /* Reverses arr[0..i] */
    static void flip(int arr[], int i)
    {
        int temp, start = 0;
        while (start < i)
        {
            temp = arr[start];
            arr[start] = arr[i];
            arr[i] = temp;
            start++;
            i--;
        }
    }

    /* Returns index of the maximum element in arr[0..n-1] */
    static int findMax(int arr[], int n)
    {
        int mi, i;
        for (mi = 0, i = 0; i < n; ++i)
            if (arr[i] > arr[mi])
                mi = i;
        return mi;
    }

    // The main function that sorts given array using flip
    // operations
    static int pancakeSort(int arr[], int n)
    {
        // Start from the complete array and one by one
        // reduce current size by one
        for (int curr_size = n; curr_size > 1; --curr_size)
        {
            // Find index of the maximum element in
            // arr[0..curr_size-1]
            int mi = findMax(arr, curr_size);

            // Move the maximum element to end of current
            // array if it's not already at the end
            if (mi != curr_size-1)
            {
                // To move at the end, first move maximum
                // number to beginning
                flip(arr, mi);

                // Now move the maximum number to end by
                // reversing current array
                flip(arr, curr_size-1);
            }
        }
    }
}

```

```

    }
    return 0;
}

/* Utility function to print array arr[] */
static void printArray(int arr[], int arr_size)
{
    for (int i = 0; i < arr_size; i++)
        System.out.print(arr[i] + " ");
    System.out.println("");
}

/*Driver function to check for above functions*/
public static void main (String[] args) {
    int arr[] = {23, 10, 20, 11, 12, 6, 7};
    int n = arr.length;
    pancakeSort(arr, n);
    System.out.println("Sorted Array:  ");
    printArray(arr, n);
}
/*This code is contributed by Devesh Agrawal*/

```

Sorted Array
6 7 10 11 12 20 23

Total O(n) flip operations are performed in above code. The overall time complexity is O(n^2).

References:

http://en.wikipedia.org/wiki/Pancake_sorting

A Pancake Sorting Problem

We have discussed [Pancake Sorting](#) in the previous post. Following is a problem based on Pancake Sorting. Given an unsorted array, sort the given array. You are allowed to do only following operation on array.

```
flip(arr, i): Reverse array from 0 to i
```

Imagine a hypothetical machine where flip(i) always takes O(1) time. Write an efficient program for sorting a given array in O(nLogn) time on the given machine. If we apply [the same algorithm](#) here, the time taken will be $O(n^2)$ because the algorithm calls findMax() in a loop and findMax() takes $O(n)$ time even on this hypothetical machine.

We can use insertion sort that uses binary search. The idea is to run a loop from second element to last element (from $i = 1$ to $n-1$), and one by one insert arr[i] in arr[0..i-1] (like [standard insertion sort algorithm](#)). When we insert an element arr[i], we can use binary search to find position of arr[i] in $O(\log n)$ time. Once we have the position, we can use some flip operations to put arr[i] at its new place. Following are abstract steps.

```
// Standard Insertion Sort Loop that starts from second element
for (i=1; i < n; i++) ----> O(n)
{
    int key = arr[i];

    // Find index of ceiling of arr[i] in arr[0..i-1] using binary search
    j = ceilSearch(arr, key, 0, i-1); ----> O(logn) (See this)
    // Apply some flip operations to put arr[i] at correct place
}
```

Since flip operation takes $O(1)$ on given hypothetical machine, total running time of above algorithm is $O(n \log n)$. Thanks to [Kumar](#) for suggesting above problem and algorithm.

Let us see how does the above algorithm work. [ceilSearch\(\)](#) actually returns the index of the smallest element which is greater than arr[i] in arr[0..i-1]. If there is no such element, it returns -1. Let the returned value be j. If j is -1, then we don't need to do anything as arr[i] is already the greatest element among arr[0..i]. Otherwise we need to put arr[i] just before arr[j].

So how to apply flip operations to put arr[i] just before arr[j] using values of i and j. Let us take an example to understand this. Let i be 6 and current array be {12, 15, 18, 30, 35, 40, **20**, 6, 90, 80}. To put 20 at its new place, the array should be changed to {12, 15, 18, **20**, 30, 35, 40, 6, 90, 80}. We apply following steps to put 20 at its new place.

- 1) Find j using ceilSearch (In the above example j is 3).
- 2) flip(arr, j-1) (array becomes {18, 15, 12, 30, 35, 40, **20**, 6, 90, 80})
- 3) flip(arr, i-1); (array becomes {40, 35, 30, 12, 15, 18, **20**, 6, 90, 80})
- 4) flip(arr, i); (array becomes {**20**, 18, 15, 12, 30, 35, 40, 6, 90, 80})
- 5) flip(arr, j); (array becomes {12, 15, 18, **20**, 30, 35, 40, 6, 90, 80})

Following is C implementation of the above algorithm

```
#include <stdlib.h>
#include <stdio.h>

/* A Binary Search based function to get index of ceiling of x in
   arr[low..high] */
int ceilSearch(int arr[], int low, int high, int x)
{
    int mid;

    /* If x is smaller than or equal to the first element,
       then return the first element */
    if(x <= arr[low])
        return low;

    /* If x is greater than the last element, then return -1 */
    if(x > arr[high])
        return -1;

    /* get the index of middle element of arr[low..high]*/
    mid = (low + high)/2; /* low + (high - low)/2 */

    /* If x is same as middle element, then return mid */
    if(arr[mid] == x)
        return mid;

    /* If x is greater than arr[mid], then either arr[mid + 1]
       is ceiling of x, or ceiling lies in arr[mid+1..high] */
    if(arr[mid] < x)
    {
        if(mid + 1 <= high && x <= arr[mid+1])
            return mid + 1;
    }
}
```

```

        else
            return ceilSearch(arr, mid+1, high, x);
    }

/* If x is smaller than arr[mid], then either arr[mid]
   is ceiling of x or ceiling lies in arr[mid-1...high] */
if (mid - 1 >= low && x > arr[mid-1])
    return mid;
else
    return ceilSearch(arr, low, mid - 1, x);
}

/* Reverses arr[0..i] */
void flip(int arr[], int i)
{
    int temp, start = 0;
    while (start < i)
    {
        temp = arr[start];
        arr[start] = arr[i];
        arr[i] = temp;
        start++;
        i--;
    }
}

/* Function to sort an array using insertion sort, binary search and flip */
void insertionSort(int arr[], int size)
{
    int i, j;

    // Start from the second element and one by one insert arr[i]
    // in already sorted arr[0..i-1]
    for(i = 1; i < size; i++)
    {
        // Find the smallest element in arr[0..i-1] which is also greater than
        // or equal to arr[i]
        int j = ceilSearch(arr, 0, i-1, arr[i]);

        // Check if there was no element greater than arr[i]
        if (j != -1)
        {
            // Put arr[i] before arr[j] using following four flip operations
            flip(arr, j-1);
            flip(arr, i-1);
            flip(arr, i);
            flip(arr, j);
        }
    }
}

/* A utility function to print an array of size n */
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d ", arr[i]);
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = {18, 40, 35, 12, 30, 35, 20, 6, 90, 80};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertionSort(arr, n);
    printArray(arr, n);
    return 0;
}

```

Output:

6 12 18 20 30 35 35 40 80 90

Tug of War

Given a set of n integers, divide the set in two subsets of $n/2$ sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly $n/2$ and if n is odd, then size of one subset must be $(n-1)/2$ and size of other subset must be $(n+1)/2$.

For example, let given set be $\{3, 4, 5, -3, 100, 1, 89, 54, 23, 20\}$, the size of set is 10. Output for this set should be $\{4, 100, 1, 23, 20\}$ and $\{3, 5, -3, 89, 54\}$. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).

Let us consider another example where n is odd. Let given set be $\{23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4\}$. The output subsets should be $\{45, -34, 12, 98, -1\}$ and $\{23, 0, -99, 4, 189, 4\}$. The sums of elements in two subsets are 120 and 121 respectively.

The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for every element. When the size of current set becomes $n/2$, we check whether this solution is better than the best solution available so far. If it is, then we update the best solution.

Following is C++ implementation for Tug of War problem. It prints the required arrays.

```
#include <iostream>
#include <stdlib.h>
#include <limits.h>
using namespace std;

// function that tries every possible solution by calling itself recursively
void TOWUtil(int* arr, int n, bool* curr_elements, int no_of_selected_elements,
             bool* soln, int* min_diff, int sum, int curr_sum, int curr_position)
{
    // checks whether the it is going out of bound
    if (curr_position == n)
        return;

    // checks that the numbers of elements left are not less than the
    // number of elements required to form the solution
    if ((n/2 - no_of_selected_elements) > (n - curr_position))
        return;

    // consider the cases when current element is not included in the solution
    TOWUtil(arr, n, curr_elements, no_of_selected_elements,
            soln, min_diff, sum, curr_sum, curr_position+1);

    // add the current element to the solution
    no_of_selected_elements++;
    curr_sum = curr_sum + arr[curr_position];
    curr_elements[curr_position] = true;

    // checks if a solution is formed
    if (no_of_selected_elements == n/2)
    {
        // checks if the solution formed is better than the best solution so far
        if (abs(sum/2 - curr_sum) < *min_diff)
        {
            *min_diff = abs(sum/2 - curr_sum);
            for (int i = 0; i<n; i++)
                soln[i] = curr_elements[i];
        }
    }
    else
    {
        // consider the cases where current element is included in the solution
        TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln,
                min_diff, sum, curr_sum, curr_position+1);
    }

    // removes current element before returning to the caller of this function
    curr_elements[curr_position] = false;
}

// main function that generate an arr
void tugOfWar(int *arr, int n)
{
    // the boolean array that contains the inclusion and exclusion of an element
    // in current set. The number excluded automatically form the other set
    bool* curr_elements = new bool[n];

    // The inclusion/exclusion array for final solution
    bool* soln = new bool[n];
```

```

int min_diff = INT_MAX;

int sum = 0;
for (int i=0; i<n; i++)
{
    sum += arr[i];
    curr_elements[i] = soln[i] = false;
}

// Find the solution using recursive function TOWUtil()
TOWUtil(arr, n, curr_elements, 0, soln, &min_diff, sum, 0, 0);

// Print the solution
cout << "The first subset is: ";
for (int i=0; i<n; i++)
{
    if (soln[i] == true)
        cout << arr[i] << " ";
}
cout << "\nThe second subset is: ";
for (int i=0; i<n; i++)
{
    if (soln[i] == false)
        cout << arr[i] << " ";
}
}

// Driver program to test above functions
int main()
{
    int arr[] = {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    tugOfWar(arr, n);
    return 0;
}

```

Output:

The first subset is: 45 -34 12 98 -1
The second subset is: 23 0 -99 4 189 4

Divide and Conquer | Set 3 (Maximum Subarray Sum)

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is $\{-2, -5, 6, -2, -3, 1, 5, -6\}$, then the maximum subarray sum is 7 (see highlighted elements).

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum. The time complexity of the Naive method is $O(n^2)$.

Using **Divide and Conquer** approach, we can find the maximum subarray sum in $O(n \log n)$ time. Following is the Divide and Conquer algorithm.

- 1) Divide the given array in two halves
- 2) Return the maximum of following three
 - a) Maximum subarray sum in left half (Make a recursive call)
 - b) Maximum subarray sum in right half (Make a recursive call)
 - c) Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.

```
// A Divide and Conquer based program for maximum subarray sum problem
#include <stdio.h>
#include <limits.h>

// A utility function to find maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// A utility function to find maximum of three integers
int max(int a, int b, int c) { return max(max(a, b), c); }

// Find the maximum possible sum in arr[] such that arr[m] is part of it
int maxCrossingSum(int arr[], int l, int m, int h)
{
    // Include elements on left of mid.
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--)
    {
        sum = sum + arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }

    // Include elements on right of mid
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m+1; i <= h; i++)
    {
        sum = sum + arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }

    // Return sum of elements on left and right of mid
    return left_sum + right_sum;
}

// Returns sum of maximum sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
{
    // Base Case: Only one element
    if (l == h)
        return arr[l];

    // Find middle point
    int m = (l + h)/2;

    /* Return maximum of following three possible cases
     * a) Maximum subarray sum in left half
     * b) Maximum subarray sum in right half
     * c) Maximum subarray sum such that the subarray crosses the midpoint */
    return max(maxSubArraySum(arr, l, m),
               maxSubArraySum(arr, m+1, h),
               maxCrossingSum(arr, l, m, h));
}
```

```

}

/*Driver program to test maxSubArraySum*/
int main()
{
    int arr[] = {2, 3, 4, 5, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    int max_sum = maxSubArraySum(arr, 0, n-1);
    printf("Maximum contiguous sum is %d\n", max_sum);
    getchar();
    return 0;
}

```

Time Complexity: `maxSubArraySum()` is a recursive method and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + ?(n)$$

The above recurrence is similar to [Merge Sort](#) and can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $?(\log n)$.

[**The Kadanes Algorithm**](#) for this problem takes $O(n)$ time. Therefore the Kadanes algorithm is better than the Divide and Conquer approach, but this problem can be considered as a good example to show power of Divide and Conquer. The above simple approach where we divide the array in two halves, reduces the time complexity from $O(n^2)$ to $O(n\log n)$.

References:

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

Counting Sort

[Counting sort](#) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

| | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Count: | 0 | 2 | 2 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

2) Modify the count array such that each element at each index stores the sum of previous counts.

| | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Count: | 0 | 2 | 4 | 4 | 5 | 6 | 6 | 7 | 7 | 7 |

The modified count array indicates the position of each object in the output sequence.

3) Output each object from the input sequence followed by decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

Following is C implementation of counting sort.

```
// C Program for counting sort
#include <stdio.h>
#include <string.h>
#define RANGE 255

// The main function that sort the given string str in alphabetical order
void countSort(char *str)
{
    // The output character array that will have sorted str
    char output[strlen(str)];

    // Create a count array to store count of individual characters and
    // initialize count array as 0
    int count[RANGE + 1], i;
    memset(count, 0, sizeof(count));

    // Store count of each character
    for(i = 0; str[i]; ++i)
        ++count[str[i]];

    // Change count[i] so that count[i] now contains actual position of
    // this character in output array
    for (i = 1; i <= RANGE; ++i)
        count[i] += count[i-1];

    // Build the output character array
    for (i = 0; str[i]; ++i)
    {
        output[count[str[i]]-1] = str[i];
        --count[str[i]];
    }

    // Copy the output array to str, so that str now
    // contains sorted characters
    for (i = 0; str[i]; ++i)
        str[i] = output[i];
}

// Driver program to test above function
int main()
{
    char str[] = "geeksforgeeks";//"applepp";

    countSort(str);

    printf("Sorted string is %s\n", str);
    return 0;
}
```

Output:

```
Sorted character array is eeeefggkkorss
```

Time Complexity: $O(n+k)$ where n is the number of elements in input array and k is the range of input.

Auxiliary Space: $O(n+k)$

Points to be noted:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. Its running time complexity is $O(n)$ with space proportional to the range of data.
3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in $O(1)$.
5. Counting sort can be extended to work for negative inputs also.

Exercise:

1. Modify above code to sort the input data in the range from M to N.
2. Modify above code to sort negative input data.
3. Is counting sort stable and online?
4. Thoughts on parallelizing the counting sort algorithm.

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- [Selection Sort](#)
- [Bubble Sort](#)
- [Insertion Sort](#)
- [Merge Sort](#)
- [Heap Sort](#)
- [QuickSort](#)
- [Radix Sort](#)
- [Bucket Sort](#)
- [ShellSort](#)

Merge Overlapping Intervals

Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity.

For example, let the given set of intervals be $\{\{1,3\}, \{2,4\}, \{5,7\}, \{6,8\}\}$. The intervals $\{1,3\}$ and $\{2,4\}$ overlap with each other, so they should be merged and become $\{1, 4\}$. Similarly $\{5, 7\}$ and $\{6, 8\}$ should be merged and become $\{5, 8\}$.

Write a function which produces the set of merged intervals for the given set of intervals.

A **simple approach** is to start from the first interval and compare it with all other intervals for overlapping, if it overlaps with any other interval, then remove the other interval from list and merge the other into the first interval. Repeat the same steps for remaining intervals after first. This approach cannot be implemented in better than $O(n^2)$ time.

An **efficient approach** is to first sort the intervals according to starting time. Once we have the sorted intervals, we can combine all intervals in a linear traversal. The idea is, in sorted array of intervals, if $\text{interval}[i]$ doesn't overlap with $\text{interval}[i-1]$, then $\text{interval}[i+1]$ cannot overlap with $\text{interval}[i-1]$ because starting time of $\text{interval}[i+1]$ must be greater than or equal to $\text{interval}[i]$. Following is the detailed step by step algorithm

1. Sort the intervals based on increasing order of starting time.
2. Push the first interval on to a stack.
3. For each interval do the following
 - a. If the current interval does not overlap with the stack top, push it.
 - b. If the current interval overlaps with stack top and ending time of current interval is more than that of stack top, update stack top with the ending time of current interval.
4. At the end stack contains the merged intervals.

Below is a C++ implementation of the above approach.

```
// A C++ program for merging overlapping intervals
#include<bits/stdc++.h>
using namespace std;

// An interval has start time and end time
struct Interval
{
    int start, end;
};

// Compares two intervals according to their starting time.
// This is needed for sorting the intervals using library
// function std::sort(). See http://goo.gl/iGspV
bool compareInterval(Interval i1, Interval i2)
{
    return (i1.start < i2.start);
}

// The main function that takes a set of intervals, merges
// overlapping intervals and prints the result
void mergeIntervals(Interval arr[], int n)
{
    // Test if the given set has at least one interval
    if (n <= 0)
        return;

    // Create an empty stack of intervals
    stack<Interval> s;

    // sort the intervals in increasing order of start time
    sort(arr, arr+n, compareInterval);

    // push the first interval to stack
    s.push(arr[0]);

    // Start from the next interval and merge if necessary
    for (int i = 1 ; i < n; i++)
    {
        // get interval from stack top
        Interval top = s.top();

        // if current interval is not overlapping with stack top,
        // push it to the stack
        if (top.end < arr[i].start)
            s.push(arr[i]);

        // Otherwise update the ending time of top if ending of current
    }
}
```

```

        // interval is more
        else if (top.end < arr[i].end)
        {
            top.end = arr[i].end;
            s.pop();
            s.push(top);
        }
    }

    // Print contents of stack
    cout << "\n The Merged Intervals are: ";
    while (!s.empty())
    {
        Interval t = s.top();
        cout << "[" << t.start << "," << t.end << "] ";
        s.pop();
    }
    return;
}

// Driver program
int main()
{
    Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };
    int n = sizeof(arr)/sizeof(arr[0]);
    mergeIntervals(arr, n);
    return 0;
}

```

Output:

The Merged Intervals are: [1,9]

Time complexity of the method is $O(n \log n)$ which is for sorting. Once the array of intervals is sorted, merging takes linear time.

A $O(n \log n)$ and $O(1)$ Extra Space Solution

The above solution requires $O(n)$ extra space for stack. We can avoid use of extra space by doing merge operations in-place. Below are detailed steps.

- 1) Sort all intervals in decreasing order of start time.
- 2) Traverse sorted intervals starting from first interval, do following for every interval.
 - a) If current interval is not first interval and it overlaps with previous interval, then merge it with previous interval.
 - b) Else add current interval to output list of intervals.

Note that if intervals are sorted by decreasing order of start times, we can quickly check if intervals overlap or not by comparing start time of previous interval with end time of current interval.

Below is C++ implementation of above algorithm

```

// C++ program to merge overlapping Intervals in
// O(n Log n) time and O(1) extra space.
#include<bits/stdc++.h>
using namespace std;

// An Interval
struct Interval
{
    int s, e;
};

// Function used in sort
bool mycomp(Interval a, Interval b)
{   return a.s > b.s; }

void mergeIntervals(Interval arr[], int n)
{
    // Sort Intervals in decreasing order of
    // start time
    sort(arr, arr+n, mycomp);

    int index = 0; // Stores index of last element
                   // in output array (modified arr[])
}

// Traverse all input Intervals
for (int i=0; i<n; i++)
{

```

```

// If this is not first Interval and overlaps
// with the previous one
if (index != 0 && arr[index-1].s <= arr[i].e)
{
    // Merge previous and current Intervals
    arr[index-1].e = max(arr[index-1].e, arr[i].e);
    arr[index-1].s = min(arr[index-1].s, arr[i].s);
}
else // Doesn't overlap with previous, add to
{
    // solution
    arr[index] = arr[i];
    index++;
}
}

// Now arr[0..index-1] stores the merged Intervals
cout << "\n The Merged Intervals are: ";
for (int i = 0; i < index; i++)
    cout << "[" << arr[i].s << ", " << arr[i].e << "] ";
}

// Driver program
int main()
{
    Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };
    int n = sizeof(arr)/sizeof(arr[0]);
    mergeIntervals(arr, n);
    return 0;
}

```

Output:

The Merged Intervals are: [1,9]

Thanks to [Gaurav Ahirwar](#) for suggesting this method.

Find the maximum repeating number in O(n) time and O(1) extra space

Given an array of size n , the array contains numbers in range from 0 to $k-1$ where k is a positive integer and $k \leq n$. Find the maximum repeating number in this array. For example, let k be 10 the given array be $arr[] = \{1, 2, 2, 2, 0, 2, 0, 2, 3, 8, 0, 9, 2, 3\}$, the maximum repeating number would be 2. Expected time complexity is $O(n)$ and extra space allowed is $O(1)$. Modifications to array are allowed.

The **naive approach** is to run two loops, the outer loop picks an element one by one, the inner loop counts number of occurrences of the picked element. Finally return the element with maximum count. Time complexity of this approach is $O(n^2)$.

A **better approach** is to create a count array of size k and initialize all elements of $count[]$ as 0. Iterate through all elements of input array, and for every element $arr[i]$, increment $count[arr[i]]$. Finally, iterate through $count[]$ and return the index with maximum value. This approach takes $O(n)$ time, but requires $O(k)$ space.

Following is the **$O(n)$ time and $O(1)$ extra space** approach. Let us understand the approach with a simple example where $arr[] = \{2, 3, 3, 5, 3, 4, 1, 7\}$, $k = 8$, $n = 8$ (number of elements in $arr[]$).

- 1) Iterate though input array $arr[]$, for every element $arr[i]$, increment $arr[arr[i]\%k]$ by k ($arr[]$ becomes $\{2, 11, 11, 29, 11, 12, 1, 15\}$)
- 2) Find the maximum value in the modified array (maximum value is 29). Index of the maximum value is the maximum repeating element (index of 29 is 3).
- 3) If we want to get the original array back, we can iterate through the array one more time and do $arr[i] = arr[i] \% k$ where i varies from 0 to $n-1$.

How does the above algorithm work? Since we use $arr[i]\%k$ as index and add value k at the index $arr[i]\%k$, the index which is equal to maximum repeating element will have the maximum value in the end. Note that k is added maximum number of times at the index equal to maximum repeating element and all array elements are smaller than k .

Following is C++ implementation of the above algorithm

C++

```
// C++ program to find the maximum repeating number

#include<iostream>
using namespace std;

// Returns maximum repeating element in arr[0..n-1].
// The array elements are in range from 0 to k-1
int maxRepeating(int* arr, int n, int k)
{
    // Iterate though input array, for every element
    // arr[i], increment arr[arr[i]\%k] by k
    for (int i = 0; i < n; i++)
        arr[arr[i]\%k] += k;

    // Find index of the maximum repeating element
    int max = arr[0], result = 0;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
            result = i;
        }
    }

    /* Uncomment this code to get the original array back
    for (int i = 0; i < n; i++)
        arr[i] = arr[i]\%k; */

    // Return index of the maximum element
    return result;
}

// Driver program to test above function
int main()
{
    int arr[] = {2, 3, 3, 5, 3, 4, 1, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 8;

    cout << "The maximum repeating number is " <<
    maxRepeating(arr, n, k) << endl;
```

```
    return 0;  
}
```

Java

```
// Java program to find the maximum repeating number  
import java.io.*;  
  
class MaxRepeating {  
  
    // Returns maximum repeating element in arr[0..n-1].  
    // The array elements are in range from 0 to k-1  
    static int maxRepeating(int arr[], int n, int k)  
    {  
        // Iterate though input array, for every element  
        // arr[i], increment arr[arr[i]%k] by k  
        for (int i = 0; i < n; i++)  
            arr[(arr[i]%k)] += k;  
  
        // Find index of the maximum repeating element  
        int max = arr[0], result = 0;  
        for (int i = 1; i < n; i++)  
        {  
            if (arr[i] > max)  
            {  
                max = arr[i];  
                result = i;  
            }  
        }  
  
        /* Uncomment this code to get the original array back  
        for (int i = 0; i < n; i++)  
            arr[i] = arr[i]%k; */  
  
        // Return index of the maximum element  
        return result;  
    }  
  
    /*Driver function to check for above function*/  
    public static void main (String[] args)  
    {  
  
        int arr[] = {2, 3, 3, 5, 3, 4, 1, 7};  
        int n = arr.length;  
        int k=8;  
        System.out.println("Maximum repeating element is: " +  
                           maxRepeating(arr,n,k));  
    }  
}  
/* This code is contributed by Devesh Agrawal */
```

The maximum repeating number is 3

Exercise:

The above solution prints only one repeating element and doesn't work if we want to print all maximum repeating elements. For example, if the input array is {2, 3, 2, 3}, the above solution will print only 3. What if we need to print both of 2 and 3 as both of them occur maximum number of times. Write a O(n) time and O(1) extra space function that prints all maximum repeating elements. (Hint: We can use maximum quotient $arr[i]/n$ instead of maximum value in step 2).

Note that the above solutions may cause overflow if adding k repeatedly makes the value more than INT_MAX.

Stock Buy Sell to Maximize Profit

The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days. For example, if the given array is {100, 180, 260, 310, 40, 535, 695}, the maximum profit can be earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

If we are allowed to buy and sell only once, then we can use following algorithm. [Maximum difference between two elements](#). Here we are allowed to buy and sell multiple times. Following is algorithm for this problem.

1. Find the local minima and store it as starting index. If not exists, return.
2. Find the local maxima. and store it as ending index. If we reach the end, set the end as ending index.
3. Update the solution (Increment count of buy sell pairs)
4. Repeat the above steps if end is not reached.

```
// Program to find best buying and selling days
#include <stdio.h>

// solution structure
struct Interval
{
    int buy;
    int sell;
};

// This function finds the buy sell schedule for maximum profit
void stockBuySell(int price[], int n)
{
    // Prices must be given for at least two days
    if (n == 1)
        return;

    int count = 0; // count of solution pairs

    // solution vector
    Interval sol[n/2 + 1];

    // Traverse through given price array
    int i = 0;
    while (i < n-1)
    {
        // Find Local Minima. Note that the limit is (n-2) as we are
        // comparing present element to the next element.
        while ((i < n-1) && (price[i+1] <= price[i]))
            i++;

        // If we reached the end, break as no further solution possible
        if (i == n-1)
            break;

        // Store the index of minima
        sol[count].buy = i++;

        // Find Local Maxima. Note that the limit is (n-1) as we are
        // comparing to previous element
        while ((i < n) && (price[i] >= price[i-1]))
            i++;

        // Store the index of maxima
        sol[count].sell = i-1;

        // Increment count of buy/sell pairs
        count++;
    }

    // print solution
    if (count == 0)
        printf("There is no day when buying the stock will make profit\n");
    else
    {
        for (int i = 0; i < count; i++)
            printf("Buy on day: %d\t Sell on day: %d\n", sol[i].buy, sol[i].sell);
    }
}

return;
}

// Driver program to test above functions
int main()
{
```

```
// stock prices on consecutive days
int price[] = {100, 180, 260, 310, 40, 535, 695};
int n = sizeof(price)/sizeof(price[0]);

// fucntion call
stockBuySell(price, n);

return 0;
}
```

Output:

```
Buy on day : 0    Sell on day: 3
Buy on day : 4    Sell on day: 6
```

Time Complexity: The outer loop runs till i becomes n-1. The inner two loops increment value of i in every iteration. So overall time complexity is O(n)

Rearrange positive and negative numbers in O(n) time and O(1) extra space

An array contains both positive and negative numbers in random order. Rearrange the array elements so that positive and negative numbers are placed alternatively. Number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear in the end of the array.

For example, if the input array is [-1, 2, -3, 4, 5, 6, -7, 8, 9], then the output should be [9, -7, 8, -3, 5, -1, 2, 4, 6]

The solution is to first separate positive and negative numbers using partition process of QuickSort. In the partition process, consider 0 as value of pivot element so that all negative numbers are placed before positive numbers. Once negative and positive numbers are separated, we start from the first negative number and first positive number, and swap every alternate negative number with next positive number.

C

```
// A C++ program to put positive numbers at even indexes (0,
// 2, 4,...) and negative numbers at odd indexes (1, 3, 5, ...)
#include <stdio.h>

// prototype for swap
void swap(int *a, int *b);

// The main function that rearranges elements of given array.
// It puts positive elements at even indexes (0, 2, ..) and
// negative numbers at odd indexes (1, 3, ..).
void rearrange(int arr[], int n)
{
    // The following few lines are similar to partition process
    // of QuickSort. The idea is to consider 0 as pivot and
    // divide the array around it.
    int i = -1;
    for (int j = 0; j < n; j++)
    {
        if (arr[j] < 0)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Now all positive numbers are at end and negative numbers
    // at the beginning of array. Initialize indexes for starting
    // point of positive and negative numbers to be swapped
    int pos = i+1, neg = 0;

    // Increment the negative index by 2 and positive index by 1,
    // i.e., swap every alternate negative number with next
    // positive number
    while (pos < n && neg < pos && arr[neg] < 0)
    {
        swap(&arr[neg], &arr[pos]);
        pos++;
        neg += 2;
    }
}

// A utility function to swap two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A utility function to print an array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%4d ", arr[i]);
}

// Driver program to test above functions
int main()
{
    int arr[] = {-1, 2, -3, 4, 5, 6, -7, 8, 9};
    int n = sizeof(arr)/sizeof(arr[0]);
    rearrange(arr, n);
    printArray(arr, n);
    return 0;
}
```

```
}
```

Java

```
// A JAVA program to put positive numbers at even indexes
// (0, 2, 4,...) and negative numbers at odd indexes (1, 3,
// 5, ..)
import java.io.*;

class Alternate {

    // The main function that rearranges elements of given
    // array. It puts positive elements at even indexes (0,
    // 2, ...) and negative numbers at odd indexes (1, 3, ...).
    static void rearrange(int arr[], int n)
    {
        // The following few lines are similar to partition
        // process of QuickSort. The idea is to consider 0
        // as pivot and divide the array around it.
        int i = -1, temp = 0;
        for (int j = 0; j < n; j++)
        {
            if (arr[j] < 0)
            {
                i++;
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Now all positive numbers are at end and negative numbers at
        // the beginning of array. Initialize indexes for starting point
        // of positive and negative numbers to be swapped
        int pos = i+1, neg = 0;

        // Increment the negative index by 2 and positive index by 1, i.e.,
        // swap every alternate negative number with next positive number
        while (pos < n && neg < pos && arr[neg] < 0)
        {
            temp = arr[neg];
            arr[neg] = arr[pos];
            arr[pos] = temp;
            pos++;
            neg += 2;
        }
    }

    // A utility function to print an array
    static void printArray(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + "   ");
    }

    /*Driver function to check for above functions*/
    public static void main (String[] args)
    {
        int arr[] = {-1, 2, -3, 4, 5, 6, -7, 8, 9};
        int n = arr.length;
        rearrange(arr,n);
        System.out.println("Array after rearranging: ");
        printArray(arr,n);
    }
}
/*This code is contributed by Devesh Agrawal*/
```

```
4   -3   5   -1   6   -7   2   8   9
```

Time Complexity: O(n) where n is number of elements in given array.

Auxiliary Space: O(1)

Note that the partition process changes relative order of elements.

If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Sort elements by frequency | Set 2

Given an array of integers, sort the array according to frequency of elements. For example, if the input array is {2, 3, 2, 4, 5, 12, 2, 3, 3, 3, 12}, then modify the array to {3, 3, 3, 3, 2, 2, 2, 12, 12, 4, 5}.

In the [previous post](#), we have discussed all methods for sorting according to frequency. In this post, method 2 is discussed in detail and C++ implementation for the same is provided.

Following is detailed algorithm.

- 1) Create a BST and while creating BST maintain the count i,e frequency of each coming element in same BST. This step may take $O(n\log n)$ time if a self balancing BST is used.
- 2) Do Inorder traversal of BST and store every element and count of each element in an auxiliary array. Let us call the auxiliary array as count[]. Note that every element of this array is element and frequency pair. This step takes $O(n)$ time.
- 3) Sort count[] according to frequency of the elements. This step takes $O(n\log n)$ time if a $O(n\log n)$ sorting algorithm is used.
- 4) Traverse through the sorted array count[]. For each element x, print it freq times where freq is frequency of x. This step takes $O(n)$ time.

Overall time complexity of the algorithm can be minimum $O(n\log n)$ if we use a $O(n\log n)$ sorting algorithm and use a self balancing BST with $O(\log n)$ insert operation.

Following is C++ implementation of the above algorithm.

```
// Implementation of above algorithm in C++.
#include <iostream>
#include <stdlib.h>
using namespace std;

/* A BST node has data, freq, left and right pointers */
struct BSTNode
{
    struct BSTNode *left;
    int data;
    int freq;
    struct BSTNode *right;
};

// A structure to store data and its frequency
struct dataFreq
{
    int data;
    int freq;
};

/* Function for qsort() implementation. Compare frequencies to
   sort the array according to decreasing order of frequency */
int compare(const void *a, const void *b)
{
    return ( (*(const dataFreq*)b).freq - (*(const dataFreq*)a).freq );
}

/* Helper function that allocates a new node with the given data,
   frequency as 1 and NULL left and right pointers.*/
BSTNode* newNode(int data)
{
    struct BSTNode* node = new BSTNode;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->freq = 1;
    return (node);
}

// A utility function to insert a given key to BST. If element
// is already present, then increases frequency
BSTNode *insert(BSTNode *root, int data)
{
    if (root == NULL)
        return newNode(data);
    if (data == root->data) // If already present
        root->freq += 1;
    else if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
}

// Function to copy elements and their frequencies to count[].
void store(BSTNode *root, dataFreq count[], int *index)
```

```

{
    // Base Case
    if (root == NULL) return;

    // Recur for left subtree
    store(root->left, count, index);

    // Store item from root and increment index
    count[(*index)].freq = root->freq;
    count[(*index)].data = root->data;
    (*index)++;

    // Recur for right subtree
    store(root->right, count, index);
}

// The main function that takes an input array as an argument
// and sorts the array items according to frequency
void sortByFrequency(int arr[], int n)
{
    // Create an empty BST and insert all array items in BST
    struct BSTNode *root = NULL;
    for (int i = 0; i < n; ++i)
        root = insert(root, arr[i]);

    // Create an auxiliary array 'count[]' to store data and
    // frequency pairs. The maximum size of this array would
    // be n when all elements are different
    dataFreq count[n];
    int index = 0;
    store(root, count, &index);

    // Sort the count[] array according to frequency (or count)
    qsort(count, index, sizeof(count[0]), compare);

    // Finally, traverse the sorted count[] array and copy the
    // i'th item 'freq' times to original array 'arr[]'
    int j = 0;
    for (int i = 0; i < index; i++)
    {
        for (int freq = count[i].freq; freq > 0; freq--)
            arr[j++] = count[i].data;
    }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {2, 3, 2, 4, 5, 12, 2, 3, 3, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortByFrequency(arr, n);
    printArray(arr, n);
    return 0;
}

```

Output:

3 3 3 2 2 2 12 12 5 4

Exercise:

The above implementation doesn't guarantee original order of elements with same frequency (for example, 4 comes before 5 in input, but 4 comes after 5 in output). Extend the implementation to maintain original order. For example, if two elements have same frequency then print the one which came 1st in input array.

Find a peak element

Given an array of integers. Find a peak element in it. An array element is peak if it is NOT smaller than its neighbors. For corner elements, we need to consider only one neighbor. For example, for input array {5, 10, 20, 15}, 20 is the only peak element. For input array {10, 20, 15, 2, 23, 90, 67}, there are two peak elements: 20 and 90. Note that we need to return any one peak element.

Following corner cases give better idea about the problem.

- 1) If input array is sorted in strictly increasing order, the last element is always a peak element. For example, 50 is peak element in {10, 20, 30, 40, 50}.
- 2) If input array is sorted in strictly decreasing order, the first element is always a peak element. 100 is the peak element in {100, 80, 60, 50, 20}.
- 3) If all elements of input array are same, every element is a peak element.

It is clear from above examples that there is always a peak element in input array in any input array.

A **simple solution** is to do a linear scan of array and as soon as we find a peak element, we return it. The worst case time complexity of this method would be O(n).

Can we find a peak element in worst time complexity better than O(n)?

We can use [Divide and Conquer](#) to find a peak in O(Logn) time. The idea is Binary Search based, we compare middle element with its neighbors. If middle element is greater than both of its neighbors, then we return it. If the middle element is smaller than the its left neighbor, then there is always a peak in left half (Why? take few examples). If the middle element is smaller than the its right neighbor, then there is always a peak in right half (due to same reason as left half). Following are C and Java implementations of this approach.

C/C++

```
// A C++ program to find a peak element element using divide and conquer
#include <stdio.h>

// A binary search based function that returns index of a peak element
int findPeakUtil(int arr[], int low, int high, int n)
{
    // Find index of middle element
    int mid = low + (high - low)/2; /* (low + high)/2 */

    // Compare middle element with its neighbours (if neighbours exist)
    if ((mid == 0 || arr[mid-1] <= arr[mid]) &&
        (mid == n-1 || arr[mid+1] <= arr[mid]))
        return mid;

    // If middle element is not peak and its left neighbour is greater
    // than it, then left half must have a peak element
    else if (mid > 0 && arr[mid-1] > arr[mid])
        return findPeakUtil(arr, low, (mid - 1), n);

    // If middle element is not peak and its right neighbour is greater
    // than it, then right half must have a peak element
    else return findPeakUtil(arr, (mid + 1), high, n);
}

// A wrapper over recursive function findPeakUtil()
int findPeak(int arr[], int n)
{
    return findPeakUtil(arr, 0, n-1, n);
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 3, 20, 4, 1, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Index of a peak point is %d", findPeak(arr, n));
    return 0;
}
```

Java

```
// A Java program to find a peak element element using divide and conquer
import java.util.*;
import java.lang.*;
import java.io.*;

class PeakElement
{
    // A binary search based function that returns index of a peak
```

```

// element
static int findPeakUtil(int arr[], int low, int high, int n)
{
    // Find index of middle element
    int mid = low + (high - low)/2; /* (low + high)/2 */

    // Compare middle element with its neighbours (if neighbours
    // exist)
    if ((mid == 0 || arr[mid-1] <= arr[mid]) && (mid == n-1 ||
        arr[mid+1] <= arr[mid]))
        return mid;

    // If middle element is not peak and its left neighbor is
    // greater than it, then left half must have a peak element
    else if (mid > 0 && arr[mid-1] > arr[mid])
        return findPeakUtil(arr, low, (mid -1), n);

    // If middle element is not peak and its right neighbor
    // is greater than it, then right half must have a peak
    // element
    else return findPeakUtil(arr, (mid + 1), high, n);
}

// A wrapper over recursive function findPeakUtil()
static int findPeak(int arr[], int n)
{
    return findPeakUtil(arr, 0, n-1, n);
}

// Driver method
public static void main (String[] args)
{
    int arr[] = {1, 3, 20, 4, 1, 0};
    int n = arr.length;
    System.out.println("Index of a peak point is " +
                       findPeak(arr, n));
}
}

```

Index of a peak point is 2

Time Complexity: O(Logn) where n is number of elements in input array.

Exercise:

Consider the following modified definition of peak element. An array element is peak if it is greater than its neighbors. Note that an array may not contain a peak element with this modified definition.

References:

<http://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf>
<http://www.youtube.com/watch?v=HtSuA80QTy0>

Print all possible combinations of r elements in a given array of size n

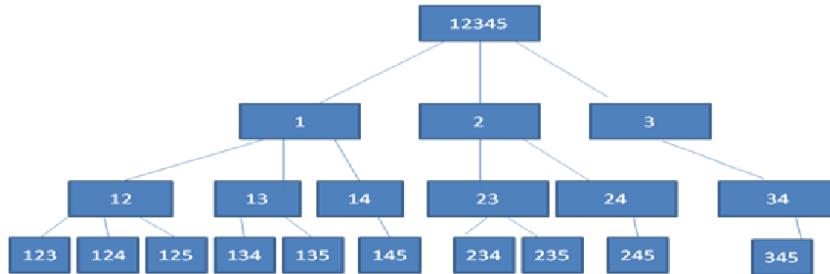
Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Following are two methods to do this.

Method 1 (Fix Elements and Recur)

We create a temporary array data[] which stores all outputs one by one. The idea is to start from first index (index = 0) in data[], one by one fix elements at this index and recur for remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at index 0 in data[], then recur for remaining indexes, then we fix 2 at index 0 and recur. Finally, we fix 3 and recur for remaining indexes. When number of elements in data[] becomes equal to r (size of a combination), we print data[].

Following diagram shows recursion tree for same input.



Following is C++ implementation of above approach.

C

```
// Program to print all combination of size r in an array of size n
#include <stdio.h>
void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temporary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/* arr[] ----> Input Array
   data[] ----> Temporary array to store current combination
   start & end ---> Starting and Ending indexes in arr[]
   index ----> Current index in data[]
   r ---> Size of a combination to be printed */
void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r)
{
    // Current combination is ready to be printed, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    // replace index with all possible elements. The condition
    // "end-i+1 >= r-index" makes sure that including one element
    // at index will make a combination with remaining elements
    // at remaining positions
    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
    }
}

// Driver program to test above functions
int main()
```

```

{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
}

```

Java

```

// Java program to print all combination of size r in an array of size n
import java.io.*;

class Permutation {

    /* arr[] ----> Input Array
    data[] ----> Temporary array to store current combination
    start & end ----> Starting and Ending indexes in arr[]
    index ----> Current index in data[]
    r ----> Size of a combination to be printed */
    static void combinationUtil(int arr[], int data[], int start,
                                int end, int index, int r)
    {
        // Current combination is ready to be printed, print it
        if (index == r)
        {
            for (int j=0; j<r; j++)
                System.out.print(data[j]+" ");
            System.out.println("");
            return;
        }

        // replace index with all possible elements. The condition
        // "end-i+1 >= r-index" makes sure that including one element
        // at index will make a combination with remaining elements
        // at remaining positions
        for (int i=start; i<=end && end-i+1 >= r-index; i++)
        {
            data[index] = arr[i];
            combinationUtil(arr, data, i+1, end, index+1, r);
        }
    }

    // The main function that prints all combinations of size r
    // in arr[] of size n. This function mainly uses combinationUtil()
    static void printCombination(int arr[], int n, int r)
    {
        // A temporary array to store all combination one by one
        int data[]={};

        // Print all combination using temporary array 'data[]'
        combinationUtil(arr, data, 0, n-1, 0, r);
    }

    /*Driver function to check for above function*/
    public static void main (String[] args) {
        int arr[] = {1, 2, 3, 4, 5};
        int r = 3;
        int n = arr.length;
        printCombination(arr, n, r);
    }
}

/* This code is contributed by Devesh Agrawal */

```

```

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

```

How to handle duplicates?

Note that the above method doesn't handle duplicates. For example, if input array is {1, 2, 1} and r is 2, then the program prints {1, 2} and {2, 1}

as two different combinations. We can avoid duplicates by adding following two additional things to above code.

- 1) Add code to sort the array before calling combinationUtil() in printCombination()
- 2) Add following lines at the end of for loop in combinationUtil()

```
// Since the elements are sorted, all occurrences of an element
// must be together
while (arr[i] == arr[i+1])
    i++;
```

See [this](#) for an implementation that handles duplicates.

Method 2 (Include and Exclude every element)

Like the above method, We create a temporary array data[]. The idea here is similar to [Subset Sum Problem](#). We one by one consider every element of input array, and recur for two cases:

- 1) The element is included in current combination (We put the element in data[] and increment next available index in data[])
- 2) The element is excluded in current combination (We do not put the element and do not change index)

When number of elements in data[] become equal to r (size of a combination), we print it.

This method is mainly based on [Pascals Identity](#), i.e. $n_c_r = n-1_c_r + n-1_c_{r-1}$

Following is C++ implementation of method 2.

C

```
// Program to print all combination of size r in an array of size n
#include<stdio.h>
void combinationUtil(int arr[], int n, int r, int index, int data[], int i);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temporary array 'data[]'
    combinationUtil(arr, n, r, 0, data, 0);
}

/* arr[] ----> Input Array
   n      ----> Size of input array
   r      ----> Size of a combination to be printed
   index  ----> Current index in data[]
   data[] ----> Temporary array to store current combination
   i      ----> index of current element in arr[]      */
void combinationUtil(int arr[], int n, int r, int index, int data[], int i)
{
    // Current combination is ready, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    // When no more elements are there to put in data[]
    if (i >= n)
        return;

    // current is included, put next at next location
    data[index] = arr[i];
    combinationUtil(arr, n, r, index+1, data, i+1);

    // current is excluded, replace it with next (Note that
    // i+1 is passed, but index is not changed)
    combinationUtil(arr, n, r, index, data, i+1);
}

// Driver program to test above functions
int main()
{
```

```

int arr[] = {1, 2, 3, 4, 5};
int r = 3;
int n = sizeof(arr)/sizeof(arr[0]);
printCombination(arr, n, r);
return 0;
}

```

Java

```

// Java program to print all combination of size r in an array of size n
import java.io.*;

class Permutation {

    /* arr[] ----> Input Array
    data[] ----> Temporary array to store current combination
    start & end ----> Starting and Ending indexes in arr[]
    index ----> Current index in data[]
    r ----> Size of a combination to be printed */
    static void combinationUtil(int arr[], int n, int r, int index,
                                int data[], int i)
    {
        // Current combination is ready to be printed, print it
        if (index == r)
        {
            for (int j=0; j<r; j++)
                System.out.print(data[j]+" ");
            System.out.println("");
            return;
        }

        // When no more elements are there to put in data[]
        if (i >= n)
            return;

        // current is included, put next at next location
        data[index] = arr[i];
        combinationUtil(arr, n, r, index+1, data, i+1);

        // current is excluded, replace it with next (Note that
        // i+1 is passed, but index is not changed)
        combinationUtil(arr, n, r, index, data, i+1);
    }

    // The main function that prints all combinations of size r
    // in arr[] of size n. This function mainly uses combinationUtil()
    static void printCombination(int arr[], int n, int r)
    {
        // A temporary array to store all combination one by one
        int data[]={new int[r]};

        // Print all combination using temporary array 'data[]'
        combinationUtil(arr, n, r, 0, data, 0);
    }

    /*Driver function to check for above function*/
    public static void main (String[] args) {
        int arr[] = {1, 2, 3, 4, 5};
        int r = 3;
        int n = arr.length;
        printCombination(arr, n, r);
    }
}
/* This code is contributed by Devesh Agrawal */

```

```

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

```

How to handle duplicates in method 2?

Like method 1, we can follow two things to handle duplicates.

- 1) Add code to sort the array before calling combinationUtil() in printCombination()
- 2) Add following lines between two recursive calls of combinationUtil() in combinationUtil()

```
// Since the elements are sorted, all occurrences of an element  
// must be together  
while (arr[i] == arr[i+1])  
    i++;
```

See [this](#) for an implementation that handles duplicates.

Given an array of size n and a number k, find all elements that appear more than n/k times

Given an array of size n, find all elements in array that appear more than n/k times. For example, if the input arrays is {3, 1, 2, 2, 1, 2, 3, 3} and k is 4, then the output should be [2, 3]. Note that size of array is 8 (or $n = 8$), so we need to find all elements that appear more than 2 (or $8/4$) times. There are two elements that appear more than two times, 2 and 3.

A **simple method** is to pick all elements one by one. For every picked element, count its occurrences by traversing the array, if count becomes more than n/k , then print the element. Time Complexity of this method would be $O(n^2)$.

A better solution is to **use sorting**. First, sort all elements using a $O(n \log n)$ algorithm. Once the array is sorted, we can find all required elements in a linear scan of array. So overall time complexity of this method is $O(n \log n) + O(n)$ which is $O(n \log n)$.

Following is an interesting **$O(nk)$ solution:**

We can solve the above problem in $O(nk)$ time using $O(k-1)$ extra space. Note that there can never be more than $k-1$ elements in output (Why?). There are mainly three steps in this algorithm.

- 1) Create a temporary array of size $(k-1)$ to store elements and their counts (The output elements are going to be among these $k-1$ elements). Following is structure of temporary array elements.

```
struct eleCount {
    int element;
    int count;
};

struct eleCount temp[];
```

This step takes $O(k)$ time.

- 2) Traverse through the input array and update $\text{temp}[]$ (add/remove an element or increase/decrease count) for every traversed element. The array $\text{temp}[]$ stores potential $(k-1)$ candidates at every step. This step takes $O(nk)$ time.

- 3) Iterate through final $(k-1)$ potential candidates (stored in $\text{temp}[]$). or every element, check if it actually has count more than n/k . This step takes $O(nk)$ time.

The main step is step 2, how to maintain $(k-1)$ potential candidates at every point? The steps used in step 2 are like famous game: [Tetris](#). We treat each number as a piece in Tetris, which falls down in our temporary array $\text{temp}[]$. Our task is to try to keep the same number stacked on the same column (count in temporary array is incremented).

Consider $k = 4$, $n = 9$
Given array: 3 1 2 2 2 1 4 3 3

```
i = 0
      3
temp[] has one element, 3 with count 1

i = 1
      3 1
temp[] has two elements, 3 and 1 with
counts 1 and 1 respectively

i = 2
      3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 1 respectively.

i = 3
      - - 2
      3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 2 respectively.

i = 4
      - - 2
      - - 2
      3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 3 respectively.

i = 5
      - - 2
      - 1 2
      3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 2 and 3 respectively.
```

Now the question arises, what to do when `temp[]` is full and we see a new element we remove the bottom row from stacks of elements, i.e., we decrease count of every element by 1 in `temp[]`. We ignore the current element.

```
i = 6
    - - 2
    - 1 2
temp[] has two elements, 1 and 2 with
counts as 1 and 2 respectively.

i = 7
    - 2
    3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 2 respectively.

i = 8
    3 - 2
    3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 2, 1 and 2 respectively.
```

Finally, we have at most $k-1$ numbers in `temp[]`. The elements in `temp` are {3, 1, 2}. Note that the counts in `temp[]` are useless now, the counts were needed only in step 2. Now we need to check whether the actual counts of elements in `temp[]` are more than n/k ($9/4$) or not. The elements 3 and 2 have counts more than $9/4$. So we print 3 and 2.

Note that the algorithm doesn't miss any output element. There can be two possibilities, many occurrences are together or spread across the array. If occurrences are together, then count will be high and won't become 0. If occurrences are spread, then the element would come again in `temp[]`. Following is C++ implementation of above algorithm.

```
// A C++ program to print elements with count more than n/k
#include<iostream>
using namespace std;

// A structure to store an element and its current count
struct eleCount
{
    int e; // Element
    int c; // Count
};

// Prints elements with more than n/k occurrences in arr[] of
// size n. If there are no such elements, then it prints nothing.
void moreThanNdK(int arr[], int n, int k)
{
    // k must be greater than 1 to get some output
    if (k < 2)
        return;

    /* Step 1: Create a temporary array (contains element
       and count) of size k-1. Initialize count of all
       elements as 0 */
    struct eleCount temp[k-1];
    for (int i=0; i<k-1; i++)
        temp[i].c = 0;

    /* Step 2: Process all elements of input array */
    for (int i = 0; i < n; i++)
    {
        int j;

        /* If arr[i] is already present in
           the element count array, then increment its count */
        for (j=0; j<k-1; j++)
        {
            if (temp[j].e == arr[i])
            {
                temp[j].c += 1;
                break;
            }
        }

        /* If arr[i] is not present in temp[] */
        if (j == k-1)
        {
            int l;

            /* If there is position available in temp[], then place
               arr[i] in the first available position and set count as 1*/
            for (l=0; l<k-1; l++)
```

```

    {
        if (temp[l].c == 0)
        {
            temp[l].e = arr[i];
            temp[l].c = 1;
            break;
        }
    }

    /* If all the position in the temp[] are filled, then
       decrease count of every element by 1 */
    if (l == k-1)
        for (l=0; l<k; l++)
            temp[l].c -= 1;
    }
}

/*Step 3: Check actual counts of potential candidates in temp[]*/
for (int i=0; i<k-1; i++)
{
    // Calculate actual count of elements
    int ac = 0; // actual count
    for (int j=0; j<n; j++)
        if (arr[j] == temp[i].e)
            ac++;

    // If actual count is more than n/k, then print it
    if (ac > n/k)
        cout << "Number:" << temp[i].e
            << " Count:" << ac << endl;
}
}

/* Driver program to test above function */
int main()
{
    cout << "First Test\n";
    int arr1[] = {4, 5, 6, 7, 8, 4, 4};
    int size = sizeof(arr1)/sizeof(arr1[0]);
    int k = 3;
    moreThanNdK(arr1, size, k);

    cout << "\nSecond Test\n";
    int arr2[] = {4, 2, 2, 7};
    size = sizeof(arr2)/sizeof(arr2[0]);
    k = 3;
    moreThanNdK(arr2, size, k);

    cout << "\nThird Test\n";
    int arr3[] = {2, 7, 2};
    size = sizeof(arr3)/sizeof(arr3[0]);
    k = 2;
    moreThanNdK(arr3, size, k);

    cout << "\nFourth Test\n";
    int arr4[] = {2, 3, 3, 2};
    size = sizeof(arr4)/sizeof(arr4[0]);
    k = 3;
    moreThanNdK(arr4, size, k);

    return 0;
}

```

Output:

```

First Test
Number:4 Count:3

Second Test
Number:2 Count:2

Third Test
Number:2 Count:2

Fourth Test
Number:2 Count:2
Number:3 Count:2

```

Time Complexity: O(nk)
Auxiliary Space: O(k)

Generally asked variations of this problem are, find all elements that appear $n/3$ times or $n/4$ times in $O(n)$ time complexity and $O(1)$ extra space.

Hashing can also be an efficient solution. With a good hash function, we can solve the above problem in $O(n)$ time on average. Extra space required hashing would be higher than $O(k)$. Also, hashing cannot be used to solve above variations with $O(1)$ extra space.

Exercise:

The above problem can be solved in $O(n \log k)$ time with the help of more appropriate data structures than array for auxiliary storage of $k-1$ elements. Suggest a $O(n \log k)$ approach.

Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time)

Given a function int f(unsigned int x) which takes a **non-negative integer** x as input and returns an **integer** as output. The function is monotonically increasing with respect to value of x, i.e., the value of f(x+1) is greater than f(x) for every input x. Find the value n where f() becomes positive for the first time. Since f() is monotonically increasing, values of f(n+1), f(n+2), must be positive and values of f(n-2), f(n-3), .. must be negative.

Find n in O(logn) time, you may assume that f(x) can be evaluated in O(1) time for any input x.

A **simple solution** is to start from i equals to 0 and one by one calculate value of f(i) for 1, 2, 3, 4 .. etc until we find a positive f(i). This works, but takes O(n) time.

Can we apply Binary Search to find n in O(Logn) time? We cant directly apply Binary Search as we dont have an upper limit or high index. The idea is to do repeated doubling until we find a positive value, i.e., check values of f() for following values until f(i) becomes positive.

```
f(0)
f(1)
f(2)
f(4)
f(8)
f(16)
f(32)
...
...
f(high)
```

Let 'high' be the value of i when f() becomes positive for first time.

Can we apply Binary Search to find n after finding high? We can apply Binary Search now, we can use high/2? as low and high as high indexes in binary search. The result n must lie between high/2? and high.

Number of steps for finding high is O(Logn). So we can find high in O(Logn) time. What about time taken by Binary Search between high/2 and high? The value of high must be less than 2^n . The number of elements between high/2 and high must be O(n). Therefore, time complexity of Binary Search is O(Logn) and overall time complexity is $2 \cdot O(Logn)$ which is O(Logn).

```
#include <stdio.h>
int binarySearch(int low, int high); // prototype

// Let's take an example function as f(x) = x^2 - 10*x - 20
// Note that f(x) can be any monotonically increasing function
int f(int x) { return (x*x - 10*x - 20); }

// Returns the value x where above function f() becomes positive
// first time.
int findFirstPositive()
{
    // When first value itself is positive
    if (f(0) > 0)
        return 0;

    // Find 'high' for binary search by repeated doubling
    int i = 1;
    while (f(i) <= 0)
        i = i*2;

    // Call binary search
    return binarySearch(i/2, i);
}

// Searches first positive value of f(i) where low <= i <= high
int binarySearch(int low, int high)
{
    if (high >= low)
    {
        int mid = low + (high - low)/2; /* mid = (low + high)/2 */

        // If f(mid) is greater than 0 and one of the following two
        // conditions is true:
        // a) mid is equal to low
        // b) f(mid-1) is negative
        if (f(mid) > 0 && (mid == low || f(mid-1) <= 0))
            return mid;

        // If f(mid) is smaller than or equal to 0
        if (f(mid) <= 0)
            return binarySearch((mid + 1), high);
        else // f(mid) > 0
    }
}
```

```
        return binarySearch(low, (mid -1));
    }

/* Return -1 if there is no positive value in given range */
return -1;
}

/* Driver program to check above functions */
int main()
{
    printf("The value n where f() becomes positive first is %d",
           findFirstPositive());
    return 0;
}
```

Output:

The value n where f() becomes positive first is 12

Find the Increasing subsequence of length three with maximum product

Given a sequence of non-negative integers, find the subsequence of length 3 having maximum product with the numbers of the subsequence being in ascending order.

Examples:

```
Input:  
arr[] = {6, 7, 8, 1, 2, 3, 9, 10}  
Output:  
8 9 10
```

```
Input:  
arr[] = {1, 5, 10, 8, 9}  
Output: 5 8 9
```

Since we want to find the maximum product, we need to find following two things for every element in the given sequence:

LSL: The largest smaller element on left of given element

LGR: The largest greater element on right of given element.

Once we find LSL and LGR for an element, we can find the product of element with its LSL and LGR (if they both exist). We calculate this product for every element and return maximum of all products.

A **simple method** is to use nested loops. The outer loop traverses every element in sequence. Inside the outer loop, run two inner loops (one after other) to find LSL and LGR of current element. Time complexity of this method is $O(n^2)$.

We can do this **in $O(n\log n)$ time**. For simplicity, let us first create two arrays LSL[] and LGR[] of size n each where n is number of elements in input array arr[]. The main task is to fill two arrays LSL[] and LGR[]. Once we have these two arrays filled, all we need to find maximum product $LSL[i]*arr[i]*LGR[i]$ where $0 < i < n-1$ (Note that LSL[i] doesn't exist for $i = 0$ and LGR[i] doesn't exist for $i = n-1$). We can **fill LSL[]** in $O(n\log n)$ time. The idea is to use a Balanced Binary Search Tree like AVL. We start with empty AVL tree, insert the leftmost element in it. Then we traverse the input array starting from the second element to second last element. For every element currently being traversed, we find the floor of it in AVL tree. If floor exists, we store the floor in LSL[], otherwise we store NIL. After storing the floor, we insert the current element in the AVL tree.

We can **fill LGR[]** in $O(n)$ time. The idea is similar to [this](#) post. We traverse from right side and keep track of the largest element. If the largest element is greater than current element, we store it in LGR[], otherwise we store NIL.

Finally, we run a $O(n)$ loop and **return maximum of $LSL[i]*arr[i]*LGR[i]$**

Overall complexity of this approach is $O(n\log n) + O(n) + O(n)$ which is $O(n\log n)$. Auxiliary space required is $O(n)$. Note that we can avoid space required for LSL, we can find and use LSL values in final loop.

Find the minimum element in a sorted and rotated array

A sorted array is rotated at some unknown point, find the minimum element in it.

Following solution assumes that all elements are distinct.

Examples

Input: {5, 6, 1, 2, 3, 4}
Output: 1

Input: {1, 2, 3, 4}
Output: 1

Input: {2, 1}
Output: 1

A simple solution is to traverse the complete array and find minimum. This solution requires $\Theta(n)$ time.

We can do it in $O(\log n)$ using Binary Search. If we take a closer look at above examples, we can easily figure out following pattern: The minimum element is the only element whose previous element is greater than it. If there is no such element, then there is no rotation and first element is the minimum element. Therefore, we do binary search for an element which is smaller than the previous element. We strongly recommend you to try it yourself before seeing the following are C and Java implementations.

C/C++

```
// C program to find minimum element in a sorted and rotated array
#include <stdio.h>

int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)    return arr[0];

    // If there is only one element left
    if (high == low)   return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {3, 4, 5, 1, 2}
    if (mid < high && arr[mid+1] < arr[mid])
        return arr[mid+1];

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
        return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
        return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] = {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));

    int arr2[] = {1, 2, 3, 4};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));

    int arr3[] = {1};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

    int arr4[] = {1, 2};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

    int arr5[] = {2, 1};
    int n5 = sizeof(arr5)/sizeof(arr5[0]);
    printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));
}
```

```

int arr6[] = {5, 6, 7, 1, 2, 3, 4};
int n6 = sizeof(arr6)/sizeof(arr6[0]);
printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

int arr7[] = {1, 2, 3, 4, 5, 6, 7};
int n7 = sizeof(arr7)/sizeof(arr7[0]);
printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

int arr8[] = {2, 3, 4, 5, 6, 7, 8, 1};
int n8 = sizeof(arr8)/sizeof(arr8[0]);
printf("The minimum element is %d\n", findMin(arr8, 0, n8-1));

int arr9[] = {3, 4, 5, 1, 2};
int n9 = sizeof(arr9)/sizeof(arr9[0]);
printf("The minimum element is %d\n", findMin(arr9, 0, n9-1));

return 0;
}

```

Java

```

// Java program to find minimum element in a sorted and rotated array
import java.util.*;
import java.lang.*;
import java.io.*;

class Minimum
{
    static int findMin(int arr[], int low, int high)
    {
        // This condition is needed to handle the case when array
        // is not rotated at all
        if (high < low) return arr[0];

        // If there is only one element left
        if (high == low) return arr[low];

        // Find mid
        int mid = low + (high - low)/2; /*(low + high)/2;*/

        // Check if element (mid+1) is minimum element. Consider
        // the cases like {3, 4, 5, 1, 2}
        if (mid < high && arr[mid+1] < arr[mid])
            return arr[mid+1];

        // Check if mid itself is minimum element
        if (mid > low && arr[mid] < arr[mid - 1])
            return arr[mid];

        // Decide whether we need to go to left half or right half
        if (arr[high] > arr[mid])
            return findMin(arr, low, mid-1);
        return findMin(arr, mid+1, high);
    }

    // Driver Program
    public static void main (String[] args)
    {
        int arr1[] = {5, 6, 1, 2, 3, 4};
        int n1 = arr1.length;
        System.out.println("The minimum element is "+ findMin(arr1, 0, n1-1));

        int arr2[] = {1, 2, 3, 4};
        int n2 = arr2.length;
        System.out.println("The minimum element is "+ findMin(arr2, 0, n2-1));

        int arr3[] = {1};
        int n3 = arr3.length;
        System.out.println("The minimum element is "+ findMin(arr3, 0, n3-1));

        int arr4[] = {1, 2};
        int n4 = arr4.length;
        System.out.println("The minimum element is "+ findMin(arr4, 0, n4-1));

        int arr5[] = {2, 1};
        int n5 = arr5.length;
        System.out.println("The minimum element is "+ findMin(arr5, 0, n5-1));

        int arr6[] = {5, 6, 7, 1, 2, 3, 4};
    }
}

```

```

int n6 = arr6.length;
System.out.println("The minimum element is "+ findMin(arr6, 0, n1-1));

int arr7[] = {1, 2, 3, 4, 5, 6, 7};
int n7 = arr7.length;
System.out.println("The minimum element is "+ findMin(arr7, 0, n7-1));

int arr8[] = {2, 3, 4, 5, 6, 7, 8, 1};
int n8 = arr8.length;
System.out.println("The minimum element is "+ findMin(arr8, 0, n8-1));

int arr9[] = {3, 4, 5, 1, 2};
int n9 = arr9.length;
System.out.println("The minimum element is "+ findMin(arr9, 0, n9-1));
}
}

```

The minimum element is 1
The minimum element is 1

How to handle duplicates?

It turned out that duplicates cant be handled in O(Logn) time in all cases. Thanks to [Amit Jain](#) for inputs. The special cases that cause problems are like {2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} and {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2}. It doesnt look possible to go to left half or right half by doing constant number of comparisons at the middle. Following is an implementation that handles duplicates. It may become O(n) in worst case though.

C

```

// C program to find minimum element in a sorted and rotated array
#include <stdio.h>

int min(int x, int y) { return (x < y)? x :y; }

// The function that handles duplicates. It can be O(n) in worst case.
int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)   return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {1, 1, 0, 1}
    if (mid < high && arr[mid+1] < arr[mid])
        return arr[mid+1];

    // This case causes O(n) time
    if (arr[low] == arr[mid] && arr[high] == arr[mid])
        return min(findMin(arr, low, mid-1), findMin(arr, mid+1, high));

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
        return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
        return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] = {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));
}
```

```

int arr2[] = {1, 1, 0, 1};
int n2 = sizeof(arr2)/sizeof(arr2[0]);
printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));

int arr3[] = {1, 1, 2, 2, 3};
int n3 = sizeof(arr3)/sizeof(arr3[0]);
printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

int arr4[] = {3, 3, 3, 4, 4, 4, 5, 3, 3};
int n4 = sizeof(arr4)/sizeof(arr4[0]);
printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

int arr5[] = {2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2};
int n5 = sizeof(arr5)/sizeof(arr5[0]);
printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));

int arr6[] = {2, 2, 2, 2, 2, 2, 2, 2, 1, 1};
int n6 = sizeof(arr6)/sizeof(arr6[0]);
printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

int arr7[] = {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2};
int n7 = sizeof(arr7)/sizeof(arr7[0]);
printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

return 0;
}

```

Java

```

// Java program to find minimum element in a sorted and rotated array
import java.util.*;
import java.lang.*;
import java.io.*;

class Minimum
{
    static int min(int x, int y)
    {   return (x < y)? x :y; }

    // The function that handles duplicates. It can be O(n) in
    // worst case.
    static int findMin(int arr[], int low, int high)
    {
        // This condition is needed to handle the case when array is not
        // rotated at all
        if (high < low)  return arr[0];

        // If there is only one element left
        if (high == low) return arr[low];

        // Find mid
        int mid = low + (high - low)/2; /*(low + high)/2;*/

        // Check if element (mid+1) is minimum element. Consider
        // the cases like {1, 1, 0, 1}
        if (mid < high && arr[mid+1] < arr[mid])
            return arr[mid+1];

        // This case causes O(n) time
        if (arr[low] == arr[mid] && arr[high] == arr[mid])
            return min(findMin(arr, low, mid-1),
                       findMin(arr, mid+1, high));

        // Check if mid itself is minimum element
        if (mid > low && arr[mid] < arr[mid - 1])
            return arr[mid];

        // Decide whether we need to go to left half or right half
        if (arr[high] > arr[mid])
            return findMin(arr, low, mid-1);
        return findMin(arr, mid+1, high);
    }

    public static void main (String[] args)
    {
        int arr1[] = {5, 6, 1, 2, 3, 4};
        int n1 = arr1.length;
        System.out.println("The minimum element is "+ findMin(arr1, 0, n1-1));
    }
}

```

```
int arr2[] = {1, 1, 0, 1};
int n2 = arr2.length;
System.out.println("The minimum element is "+ findMin(arr2, 0, n2-1));

int arr3[] = {1, 1, 2, 2, 3};
int n3 = arr3.length;
System.out.println("The minimum element is "+ findMin(arr3, 0, n3-1));

int arr4[] = {3, 3, 3, 4, 4, 4, 5, 3, 3};
int n4 = arr4.length;
System.out.println("The minimum element is "+ findMin(arr4, 0, n4-1));

int arr5[] = {2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2};
int n5 = arr5.length;
System.out.println("The minimum element is "+ findMin(arr5, 0, n5-1));

int arr6[] = {2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1};
int n6 = arr6.length;
System.out.println("The minimum element is "+ findMin(arr6, 0, n6-1));

int arr7[] = {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2};
int n7 = arr7.length;
System.out.println("The minimum element is "+ findMin(arr7, 0, n7-1));
}
}
```

The minimum element is 1
The minimum element is 0
The minimum element is 1
The minimum element is 3
The minimum element is 0
The minimum element is 1
The minimum element is 0

Stable Marriage Problem

Given N men and N women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are stable (Source [Wiki](#)).

Consider the following example.

Let there be two men m_1 and m_2 and two women w_1 and w_2 .

Let m_1 's list of preferences be $\{w_1, w_2\}$

Let m_2 's list of preferences be $\{w_1, w_2\}$

Let w_1 's list of preferences be $\{m_1, m_2\}$

Let w_2 's list of preferences be $\{m_1, m_2\}$

The matching $\{ \{m_1, w_2\}, \{w_1, m_2\} \}$ is not stable because m_1 and w_1 would prefer each other over their assigned partners. The matching $\{m_1, w_1\}$ and $\{m_2, w_2\}$ is stable because there are no two people of opposite sex that would prefer each other over their assigned partners.

It is always possible to form stable marriages from lists of preferences (See references for proof). Following is GaleShapley algorithm to find a stable matching:

The idea is to iterate through all free men while there is any free man available. Every free man goes to all women in his preference list according to the order. For every woman he goes to, he checks if the woman is free, if yes, they both become engaged. If the woman is not free, then the woman chooses either says no to him or dumps her current engagement according to her preference list. So an engagement done once can be broken if a woman gets better option.

Following is complete algorithm from [Wiki](#)

```
Initialize all men and women to free
while there exist a free man m who still has a woman w to propose to
{
    w = m's highest ranked such woman to whom he has not yet proposed
    if w is free
        (m, w) become engaged
    else some pair (m', w) already exists
        if w prefers m to m'
            (m, w) become engaged
            m' becomes free
        else
            (m', w) remain engaged
}
```

Input & Output: Input is a 2D matrix of size $(2*N)*N$ where N is number of women or men. Rows from 0 to N-1 represent preference lists of men and rows from N to $2*N - 1$ represent preference lists of women. So men are numbered from 0 to N-1 and women are numbered from N to $2*N - 1$. The output is list of married pairs.

Following is C++ implementation of the above algorithm.

```
// C++ program for stable marriage problem
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;

// Number of Men or Women
#define N 4

// This function returns true if woman 'w' prefers man 'm1' over man 'm'
bool wPrefersM1OverM(int prefer[2*N][N], int w, int m, int m1)
{
    // Check if w prefers m over her current engagement m1
    for (int i = 0; i < N; i++)
    {
        // If m1 comes before m in list of w, then w prefers her
        // current engagement, don't do anything
        if (prefer[w][i] == m1)
            return true;

        // If m comes before m1 in w's list, then free her current
        // engagement and engage her with m
        if (prefer[w][i] == m)
            return false;
    }
}

// Prints stable matching for N boys and N girls. Boys are numbered as 0 to
// N-1. Girls are numbered as N to 2N-1.
void stableMarriage(int prefer[2*N][N])
```



```
        return 0;  
    }
```

Output:

| Woman | Man |
|-------|-----|
| 4 | 2 |
| 5 | 1 |
| 6 | 3 |
| 7 | 0 |

References:

<http://www.csee.wvu.edu/~ksmani/courses/fa01/random/lecnotes/lecture5.pdf>

<http://www.youtube.com/watch?v=5RSMLgy06Ew#t=11m4s>

Merge k sorted arrays | Set 1

Given k sorted arrays of size n each, merge them and print the sorted output.

Example:

Input:

```
k = 3, n = 4
arr[][] = {{1, 3, 5, 7},
           {2, 4, 6, 8},
           {0, 9, 10, 11}};
```

Output: 0 1 2 3 4 5 6 7 8 9 10 11

A simple solution is to create an output array of size $n*k$ and one by one copy all arrays to it. Finally, sort the output array using any $O(n \log n)$ sorting algorithm. This approach takes $O(nk \log n)$ time.

We can merge arrays in $O(nk \log k)$ time using Min Heap. Following is detailed algorithm

1. Create an output array of size $n*k$.
2. Create a min heap of size k and insert 1st element in all the arrays into a the heap
3. Repeat following steps $n*k$ times.
 - a) Get minimum element from heap (minimum is always at root) and store it in output array.
 - b) Replace heap root with next element from the array from which the element is extracted. If the array doesnt have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Following is C++ implementation of the above algorithm.

```
// C++ program to merge k sorted arrays of size n each.
#include<iostream>
#include<limits.h>
using namespace std;

#define n 4

// A min heap node
struct MinHeapNode
{
    int element; // The element to be stored
    int i; // index of the array from which the element is taken
    int j; // index of the next element to be picked from array
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{
    MinHeapNode *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int);

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x) { harr[0] = x; MinHeapify(0); }
};

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them together
// and prints the final sorted output.
int *mergeKArrays(int arr[][n], int k)
{
    int *output = new int[n*k]; // To store output array
```

```

// Create a min heap with k heap nodes. Every heap node
// has first element of an array
MinHeapNode *harr = new MinHeapNode[k];
for (int i = 0; i < k; i++)
{
    harr[i].element = arr[i][0]; // Store the first element
    harr[i].i = i; // index of array
    harr[i].j = 1; // Index of next element to be stored from array
}
MinHeap hp(harr, k); // Create the heap

// Now one by one get the minimum element from min
// heap and replace it with next element of its array
for (int count = 0; count < n*k; count++)
{
    // Get the minimum element and store it in output
    MinHeapNode root = hp.getMin();
    output[count] = root.element;

    // Find the next elelement that will replace current
    // root of heap. The next element belongs to same
    // array as the current root.
    if (root.j < n)
    {
        root.element = arr[root.i][root.j];
        root.j += 1;
    }
    // If root was the last element of its array
    else root.element = INT_MAX; //INT_MAX is for infinite

    // Replace root with next element of array
    hp.replaceMin(root);
}

return output;
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS
// FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element < harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x; *x = *y; *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)

```

```

        cout << arr[i] << " ";
    }

// Driver program to test above functions
int main()
{
    // Change n at the top to change number of elements
    // in an array
    int arr[][][n] = {{2, 6, 12, 34},
                      {1, 9, 20, 1000},
                      {23, 34, 90, 2000}};
    int k = sizeof(arr)/sizeof(arr[0]);

    int *output = mergeKArrays(arr, k);

    cout << "Merged array is " << endl;
    printArray(output, n*k);

    return 0;
}

```

Output:

```

Merged array is
1 2 6 9 12 20 23 34 34 90 1000 2000

```

Time Complexity: The main step is 3rd step, the loop runs $n*k$ times. In every iteration of loop, we call heapify which takes $O(\log k)$ time. Therefore, the time complexity is $O(nk \log k)$.

There are other interesting methods to merge k sorted arrays in $O(nk \log k)$, we will soon be discussing them as separate posts.

Radix Sort

The [lower bound for Comparison based sorting algorithm](#) (Merge Sort, Heap Sort, Quick-Sort .. etc) is $\Omega(n \log n)$, i.e., they cannot do better than $n \log n$.

[Counting sort](#) is a linear time sorting algorithm that sorts in $O(n+k)$ time when elements are in range from 1 to k .

What if the elements are in range from 1 to n^2 ?

We can't use counting sort because counting sort will take $O(n^2)$ which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?

[Radix Sort](#) is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The Radix Sort Algorithm

1) Do following for each digit i where i varies from least significant digit to the most significant digit.

.a) Sort input array using counting sort (or any stable sort) according to the i th digit.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

What is the running time of Radix Sort?

Let there be d digits in input integers. Radix Sort takes $O(d * (n + b))$ time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d ? If k is the maximum possible value, then d would be $O(\log_b(k))$. So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k . Let us first limit k . Let $k \leq n^c$ where c is a constant. In that case, the complexity becomes $O(n \log_b(n))$. But it still doesn't beat comparison based sorting algorithms.

What if we make value of b larger? What should be the value of b to make the time complexity linear? If we set b as n , we get the time complexity as $O(n)$. In other words, we can sort an array of integers with range from 1 to n^c if the numbers are represented in base n (or every digit takes $\log_2(n)$ bits).

Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

If we have $\log_2 n$ bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.

Implementation of Radix Sort

Following is a simple C++ implementation of Radix Sort. For simplicity, the value of d is assumed to be 10. We recommend you to see [Counting Sort](#) for details of countSort() function in below code.

C/C++

```
// C++ implementation of Radix Sort
#include<iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}
```

```

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i]/exp)%10]++;
    
    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i]/exp)%10] - 1] = arr[i];
        count[(arr[i]/exp)%10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr)/sizeof(arr[0]);
    radixsort(arr, n);
    print(arr, n);
    return 0;
}

```

Java

```

// Radix sort Java implementation
import java.io.*;
import java.util.*;

class Radix {

    // A utility function to get maximum value in arr[]
    static int getMax(int arr[], int n)
    {
        int mx = arr[0];
        for (int i = 1; i < n; i++)
            if (arr[i] > mx)
                mx = arr[i];
        return mx;
    }

    // A function to do counting sort of arr[] according to
    // the digit represented by exp.

```

```

static void countSort(int arr[], int n, int exp)
{
    int output[] = new int[n]; // output array
    int i;
    int count[] = new int[10];
    Arrays.fill(count, 0);

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i]/exp)%10]++;

    // Change count[i] so that count[i] now contains
    // actual position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i]/exp)%10] - 1] = arr[i];
        count[(arr[i]/exp)%10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
static void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// A utility function to print an array
static void print(int arr[], int n)
{
    for (int i=0; i<n; i++)
        System.out.print(arr[i]+" ");
}

/*Driver function to check for above function*/
public static void main (String[] args)
{
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = arr.length;
    radixsort(arr, n);
    print(arr, n);
}
}
/* This code is contributed by Devesh Agrawal */

```

2 24 45 66 75 90 170 802

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- [Selection Sort](#)
- [Bubble Sort](#)
- [Insertion Sort](#)
- [Merge Sort](#)
- [Heap Sort](#)
- [QuickSort](#)
- [Counting Sort](#)
- [Bucket Sort](#)
- [ShellSort](#)

References:

http://en.wikipedia.org/wiki/Radix_sort

<http://alg12.wikischolars.columbia.edu/file/view/RADIX.pdf>

[MIT Video Lecture](#)

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

Move all zeroes to end of array

Given an array of random numbers, Push all the zeros of a given array to the end of the array. For example, if the given arrays is {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0}, it should be changed to {1, 9, 8, 4, 2, 7, 6, 0, 0, 0}. The order of all other elements should be same. Expected time complexity is O(n) and extra space is O(1).

There can be many ways to solve this problem. Following is a simple and interesting way to solve this problem.

Traverse the given array arr from left to right. While traversing, maintain count of non-zero elements in array. Let the count be count. For every non-zero element arr[i], put the element at arr[count] and increment count. After complete traversal, all non-zero elements have already been shifted to front end and count is set as index of first 0. Now all we need to do is that run a loop which makes all elements zero from count till end of the array.

Below is C++ implementation of the above approach.

C

```
// A C++ program to move all zeroes at the end of array
#include <iostream>
using namespace std;

// Function which pushes all zeros to end of an array.
void pushZerosToEnd(int arr[], int n)
{
    int count = 0; // Count of non-zero elements

    // Traverse the array. If element encountered is non-
    // zero, then replace the element at index 'count'
    // with this element
    for (int i = 0; i < n; i++)
        if (arr[i] != 0)
            arr[count++] = arr[i]; // here count is
                                    // incremented

    // Now all non-zero elements have been shifted to
    // front and 'count' is set as index of first 0.
    // Make all elements 0 from count to end.
    while (count < n)
        arr[count++] = 0;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    pushZerosToEnd(arr, n);
    cout << "Array after pushing all zeros to end of array :\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Java

```
/* Java program to push zeroes to back of array */
import java.io.*;

class PushZero
{
    // Function which pushes all zeroes to end of an array.
    static void pushZerosToEnd(int arr[], int n)
    {
        int count = 0; // Count of non-zero elements

        // Traverse the array. If element encountered is
        // non-zero, then replace the element at index 'count'
        // with this element
        for (int i = 0; i < n; i++)
            if (arr[i] != 0)
                arr[count++] = arr[i]; // here count is
                                      // incremented

        // Now all non-zero elements have been shifted to
        // front and 'count' is set as index of first 0.
        // Make all elements 0 from count to end.
        while (count < n)
```

```
        arr[count++] = 0;
    }

/*Driver function to check for above functions*/
public static void main (String[] args)
{
    int arr[] = {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0, 9};
    int n = arr.length;
    pushZerosToEnd(arr, n);
    System.out.println("Array after pushing zeros to the back: ");
    for (int i=0; i<n; i++)
        System.out.print(arr[i]+" ");
}
/* This code is contributed by Devesh Agrawal */
```

Array after pushing all zeros to end of array :
1 9 8 4 2 7 6 9 0 0 0 0

Time Complexity: O(n) where n is number of elements in input array.

Auxiliary Space: O(1)

Find number of pairs (x, y) in an array such that $x^y > y^x$

Given two arrays X[] and Y[] of positive integers, find number of pairs such that $x^y > y^x$ where x is an element from X[] and y is an element from Y[].

Examples:

```
Input: X[] = {2, 1, 6}, Y = {1, 5}
Output: 3
// There are total 3 pairs where pow(x, y) is greater than pow(y, x)
// Pairs are (2, 1), (2, 5) and (6, 1)
```

```
Input: X[] = {10, 19, 18}, Y[] = {11, 15, 9};
Output: 2
// There are total 2 pairs where pow(x, y) is greater than pow(y, x)
// Pairs are (10, 11) and (10, 15)
```

The **brute force solution** is to consider each element of X[] and Y[], and check whether the given condition satisfies or not. Time Complexity of this solution is $O(m*n)$ where m and n are sizes of given arrays.

Following is C++ code based on brute force solution.

```
int countPairsBruteForce(int X[], int Y[], int m, int n)
{
    int ans = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if (pow(X[i], Y[j]) > pow(Y[j], X[i]))
                ans++;
    return ans;
}
```

Efficient Solution:

The problem can be solved in $O(n \log n + m \log n)$ time. The trick here is, if $y > x$ then $x^y > y^x$ with some exceptions. Following are simple steps based on this trick.

- 1) Sort array Y[].
- 2) For every x in X[], find the index idx of smallest number greater than x (also called ceil of x) in Y[] using binary search or we can use the inbuilt function `upper_bound()` in algorithm library.
- 3) All the numbers after idx satisfy the relation so just add ($n - idx$) to the count.

Base Cases and Exceptions:

Following are exceptions for x from X[] and y from Y[]

If $x = 0$, then the count of pairs for this x is 0.

If $x = 1$, then the count of pairs for this x is equal to count of 0s in Y[].

The following cases must be handled separately as they dont follow the general rule that x smaller than y means $x^y > y^x$.

a) $x = 2, y = 3$ or 4

b) $x = 3, y = 2$

Note that the case where $x = 4$ and $y = 2$ is not there

Following diagram shows all exceptions in tabular form. The value 1 indicates that the corresponding (x, y) form a valid pair.

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| X | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | 0 | 0 |
| | 2 | 1 | 1 | 0 | 0 | 0 |
| | 3 | 1 | 1 | 1 | 0 | 1 |
| | 4 | 1 | 1 | 0 | 0 | 0 |

Following is C++ implementation. In the following implementation, we pre-process the Y array and count 0, 1, 2, 3 and 4 in it, so that we can handle all exceptions in constant time. The array NoOfY[] is used to store the counts.

```
#include<iostream>
#include<algorithm>
using namespace std;

// This function return count of pairs with x as one element
```

```

// of the pair. It mainly looks for all values in Y[] where
// x ^ Y[i] > Y[i] ^ x
int count(int x, int Y[], int n, int NoOfY[])
{
    // If x is 0, then there cannot be any value in Y such that
    // x^Y[i] > Y[i]^x
    if (x == 0) return 0;

    // If x is 1, then the number of pairs is equal to number of
    // zeroes in Y[]
    if (x == 1) return NoOfY[0];

    // Find number of elements in Y[] with values greater than x
    // upper_bound() gets address of first greater element in Y[0..n-1]
    int* idx = upper_bound(Y, Y + n, x);
    int ans = (Y + n) - idx;

    // If we have reached here, then x must be greater than 1,
    // increase number of pairs for y=0 and y=1
    ans += (NoOfY[0] + NoOfY[1]);

    // Decrease number of pairs for x=2 and (y=4 or y=3)
    if (x == 2) ans -= (NoOfY[3] + NoOfY[4]);

    // Increase number of pairs for x=3 and y=2
    if (x == 3) ans += NoOfY[2];

    return ans;
}

// The main function that returns count of pairs (x, y) such that
// x belongs to X[], y belongs to Y[] and x^y > y^x
int countPairs(int X[], int Y[], int m, int n)
{
    // To store counts of 0, 1, 2, 3 and 4 in array Y
    int NoOfY[5] = {0};
    for (int i = 0; i < n; i++)
        if (Y[i] < 5)
            NoOfY[Y[i]]++;

    // Sort Y[] so that we can do binary search in it
    sort(Y, Y + n);

    int total_pairs = 0; // Initialize result

    // Take every element of X and count pairs with it
    for (int i=0; i<m; i++)
        total_pairs += count(X[i], Y, n, NoOfY);

    return total_pairs;
}

// Driver program to test above functions
int main()
{
    int X[] = {2, 1, 6};
    int Y[] = {1, 5};

    int m = sizeof(X)/sizeof(X[0]);
    int n = sizeof(Y)/sizeof(Y[0]);

    cout << "Total pairs = " << countPairs(X, Y, m, n);

    return 0;
}

```

Output:

```
Total pairs = 3
```

Time Complexity : Let m and n be the sizes of arrays X[] and Y[] respectively. The sort step takes O(nLogn) time. Then every element of X[] is searched in Y[] using binary search. This step takes O(mLogn) time. Overall time complexity is O(nLogn + mLogn).

Count all distinct pairs with difference equal to k

Given an integer array and a positive integer k, count all distinct pairs with difference equal to k.

Examples:

Input: arr[] = {1, 5, 3, 4, 2}, k = 3

Output: 2

There are 2 pairs with difference 3, the pairs are {1, 4} and {5, 2}

Input: arr[] = {8, 12, 16, 4, 0, 20}, k = 4

Output: 5

There are 5 pairs with difference 4, the pairs are {0, 4}, {4, 8}, {8, 12}, {12, 16} and {16, 20}

Method 1 (Simple)

A simple solution is to consider all pairs one by one and check difference between every pair. Following program implements the simple solution. We run two loops: the outer loop picks the first element of pair, the inner loop looks for the other element. This solution doesn't work if there are duplicates in array as the requirement is to count only distinct pairs.

```
/* A simple program to count pairs with difference k*/
#include<iostream>
using namespace std;

int countPairsWithDiffK(int arr[], int n, int k)
{
    int count = 0;

    // Pick all elements one by one
    for (int i = 0; i < n; i++)
    {
        // See if there is a pair of this picked element
        for (int j = i+1; j < n; j++)
            if (arr[i] - arr[j] == k || arr[j] - arr[i] == k )
                count++;
    }
    return count;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 5, 3, 4, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    cout << "Count of pairs with given diff is "
         << countPairsWithDiffK(arr, n, k);
    return 0;
}
```

Output:

Count of pairs with given diff is 2

Time Complexity of $O(n^2)$

Method 2 (Use Sorting)

We can find the count in $O(n \log n)$ time using a $O(n \log n)$ sorting algorithm like [Merge Sort](#), [Heap Sort](#), etc. Following are the detailed steps.

- 1) Initialize count as 0
- 2) Sort all numbers in increasing order.
- 3) Remove duplicates from array.
- 4) Do following for each element arr[i]
 - a) Binary Search for arr[i] + k in subarray from i+1 to n-1.
 - b) If arr[i] + k found, increment count.
- 5) Return count.

```
/* A sorting based program to count pairs with difference k*/
#include <iostream>
#include <algorithm>
using namespace std;

/* Standard binary search function */
int binarySearch(int arr[], int low, int high, int x)
{
    if (high >= low)
    {
        int mid = low + (high - low)/2;
```

```

        if (x == arr[mid])
            return mid;
        if (x > arr[mid])
            return binarySearch(arr, (mid + 1), high, x);
        else
            return binarySearch(arr, low, (mid - 1), x);
    }
    return -1;
}

/* Returns count of pairs with difference k in arr[] of size n. */
int countPairsWithDiffK(int arr[], int n, int k)
{
    int count = 0, i;
    sort(arr, arr+n); // Sort array elements

    /* code to remove duplicates from arr[] */

    // Pick a first element point
    for (i = 0; i < n-1; i++)
        if (binarySearch(arr, i+1, n-1, arr[i] + k) != -1)
            count++;

    return count;
}

```

Output:

Count of pairs with given diff is 2

Time complexity: The first step (sorting) takes $O(n\log n)$ time. The second step runs binary search n times, so the time complexity of second step is also $O(n\log n)$. Therefore, overall time complexity is $O(n\log n)$. The second step can be optimized to $O(n)$, see [this](#).

Method 3 (Use Self-balancing BST)

We can also use a self-balancing BST like [AVL tree](#) or Red Black tree to solve this problem. Following is detailed algorithm.

- 1) Initialize count as 0.
- 2) Insert all elements of arr[] in an [AVL tree](#). While inserting, ignore an element if already present in AVL tree.
- 3) Do following for each element arr[i].
 - a) Search for arr[i] + k in AVL tree, if found then increment count.
 - b) Search for arr[i] - k in AVL tree, if found then increment count.
 - c) Remove arr[i] from AVL tree.

Time complexity of above solution is also $O(n\log n)$ as search and delete operations take $O(\log n)$ time for a self-balancing binary search tree.

Method 4 (Use Hashing)

We can also use hashing to achieve the average time complexity as $O(n)$ for many cases.

- 1) Initialize count as 0.
- 2) Insert all distinct elements of arr[] in a hash map. While inserting, ignore an element if already present in the hash map.
- 3) Do following for each element arr[i].
 - a) Look for arr[i] + k in the hash map, if found then increment count.
 - b) Look for arr[i] - k in the hash map, if found then increment count.
 - c) Remove arr[i] from hash table.

A very simple case where hashing works in $O(n)$ time is the case where range of values is very small. For example, in the following implementation, range of numbers is assumed to be 0 to 99999. A simple hashing technique to use values as index can be used.

```

/* An efficient program to count pairs with difference k when the range
   numbers is small */
#define MAX 100000
int countPairsWithDiffK(int arr[], int n, int k)
{
    int count = 0; // Initialize count

    // Initialize empty hashmap.
    bool hashmap[MAX] = {false};

    // Insert array elements to hashmap
    for (int i = 0; i < n; i++)
        hashmap[arr[i]] = true;

    for (int i = 0; i < n; i++)
    {
        int x = arr[i];
        if (x - k >= 0 && hashmap[x - k])
            count++;
    }
}

```

```

        if (x + k < MAX && hashmap[x + k])
            count++;
        hashmap[x] = false;
    }
    return count;
}

```

Method 5 (Use Sorting)

- Sort the array arr
- Take two pointers, l and r, both pointing to 1st element
- Take the difference arr[r] - arr[l]
 - If value diff is K, increment count and move both pointers to next element
 - if value diff > k, move l to next element
 - if value diff < k, move r to next element
- return count

```

/* A sorting based program to count pairs with difference k*/
#include <iostream>
#include <algorithm>
using namespace std;

/* Returns count of pairs with difference k in arr[] of size n. */
int countPairsWithDiffK(int arr[], int n, int k)
{
    int count = 0;
    sort(arr, arr+n); // Sort array elements

    int l = 0;
    int r = 0;
    while(r < n)
    {
        if(arr[r] - arr[l] == k)
        {
            count++;
            l++;
            r++;
        }
        else if(arr[r] - arr[l] > k)
            l++;
        else // arr[r] - arr[l] < sum
            r++;
    }
    return count;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 5, 3, 4, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    cout << "Count of pairs with given diff is "
         << countPairsWithDiffK(arr, n, k);
    return 0;
}

```

Output:

Count of pairs with given diff is 2

Time Complexity: O(nlogn)

Find if there is a subarray with 0 sum

Given an array of positive and negative numbers, find if there is a subarray (of size at-least one) with 0 sum

Examples:

Input: {4, 2, -3, 1, 6}

Output: true

There is a subarray with zero sum from index 1 to 3.

Input: {4, 2, 0, 1, 6}

Output: true

There is a subarray with zero sum from index 2 to 2.

Input: {-3, 2, 3, 1, 6}

Output: false

There is no subarray with zero sum.

A **simple solution** is to consider all subarrays one by one and check the sum of every subarray. We can run two loops: the outer loop picks a starting point i and the inner loop tries all subarrays starting from i (See [this](#) for implementation). Time complexity of this method is $O(n^2)$.

We can also **use hashing**. The idea is to iterate through the array and for every element $arr[i]$, calculate sum of elements from 0 to i (this can simply be done as $sum += arr[i]$). If the current sum has been seen before, then there is a zero sum array. Hashing is used to store the sum values, so that we can quickly store sum and find out whether the current sum is seen before or not.

Following is Java implementation of the above approach.

```
// A Java program to find if there is a zero sum subarray
import java.util.HashMap;

class ZeroSumSubarray {

    // Returns true if arr[] has a subarray with zero sum
    static Boolean printZeroSumSubarray(int arr[])
    {
        // Creates an empty hashMap hm
        HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();

        // Initialize sum of elements
        int sum = 0;

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++)
        {
            // Add current element to sum
            sum += arr[i];

            // Return true in following cases
            // a) Current element is 0
            // b) sum of elements from 0 to i is 0
            // c) sum is already present in hash map
            if (arr[i] == 0 || sum == 0 || hm.get(sum) != null)
                return true;

            // Add sum to hash map
            hm.put(sum, i);
        }

        // We reach here only when there is no subarray with 0 sum
        return false;
    }

    public static void main(String arg[])
    {
        int arr[] = {4, 2, -3, 1, 6};
        if (printZeroSumSubarray(arr))
            System.out.println("Found a subarray with 0 sum");
        else
            System.out.println("No Subarray with 0 sum");
    }
}
```

Output:

Found a subarray with 0 sum

Time Complexity of this solution can be considered as $O(n)$ under the assumption that we have good hashing function that allows insertion and retrieval operations in $O(1)$ time.

Exercise:

Extend the above program to print starting and ending indexes of all subarrays with 0 sum.

Smallest subarray with sum greater than a given value

Given an array of integers and a number x, find the smallest subarray with sum greater than the given value.

Examples:

```
arr[] = {1, 4, 45, 6, 0, 19}
        x = 51
Output: 3
Minimum length subarray is {4, 45, 6}
```

```
arr[] = {1, 10, 5, 2, 7}
        x = 9
Output: 1
Minimum length subarray is {10}
```

```
arr[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250}
        x = 280
Output: 4
Minimum length subarray is {100, 1, 0, 200}
```

A **simple solution** is to use two nested loops. The outer loop picks a starting element, the inner loop considers all elements (on right side of current start) as ending element. Whenever sum of elements between current start and end becomes more than the given number, update the result if current length is smaller than the smallest length so far.

Following is C++ implementation of simple approach.

```
# include <iostream>
using namespace std;

// Returns length of smallest subarray with sum greater than x.
// If there is no subarray with given sum, then returns n+1
int smallestSubWithSum(int arr[], int n, int x)
{
    // Initialize length of smallest subarray as n+1
    int min_len = n + 1;

    // Pick every element as starting point
    for (int start=0; start<n; start++)
    {
        // Initialize sum starting with current start
        int curr_sum = arr[start];

        // If first element itself is greater
        if (curr_sum > x) return 1;

        // Try different ending points for current start
        for (int end=start+1; end<n; end++)
        {
            // add last element to current sum
            curr_sum += arr[end];

            // If sum becomes more than x and length of
            // this subarray is smaller than current smallest
            // length, update the smallest length (or result)
            if (curr_sum > x && (end - start + 1) < min_len)
                min_len = (end - start + 1);
        }
    }
    return min_len;
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {1, 4, 45, 6, 10, 19};
    int x = 51;
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    cout << smallestSubWithSum(arr1, n1, x) << endl;

    int arr2[] = {1, 10, 5, 2, 7};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    x = 9;
    cout << smallestSubWithSum(arr2, n2, x) << endl;

    int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    x = 280;
    cout << smallestSubWithSum(arr3, n3, x) << endl;

    return 0;
}
```

}

Output:

3
1
4

Time Complexity: Time complexity of the above approach is clearly $O(n^2)$.

Efficient Solution: This problem can be solved in **$O(n)$ time** using the idea used in [this](#) post. Thanks to Ankit and Nitin for suggesting this optimized solution.

```
// O(n) solution for finding smallest subarray with sum
// greater than x
#include <iostream>
using namespace std;

// Returns length of smallest subarray with sum greater than x.
// If there is no subarray with given sum, then returns n+1
int smallestSubWithSum(int arr[], int n, int x)
{
    // Initialize current sum and minimum length
    int curr_sum = 0, min_len = n+1;

    // Initialize starting and ending indexes
    int start = 0, end = 0;
    while (end < n)
    {
        // Keep adding array elements while current sum
        // is smaller than x
        while (curr_sum <= x && end < n)
            curr_sum += arr[end++];

        // If current sum becomes greater than x.
        while (curr_sum > x && start < n)
        {
            // Update minimum length if needed
            if (end - start < min_len)
                min_len = end - start;

            // remove starting elements
            curr_sum -= arr[start++];
        }
    }
    return min_len;
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {1, 4, 45, 6, 10, 19};
    int x = 51;
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    cout << smallestSubWithSum(arr1, n1, x) << endl;

    int arr2[] = {1, 10, 5, 2, 7};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    x = 9;
    cout << smallestSubWithSum(arr2, n2, x) << endl;

    int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    x = 280;
    cout << smallestSubWithSum(arr3, n3, x);

    return 0;
}
```

Output:

3
1
4

Sort an array according to the order defined by another array

Given two arrays A1[] and A2[], sort A1 in such a way that the relative order among the elements will be same as those are in A2. For the elements not present in A2, append them at last in sorted order.

```
Input: A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}
       A2[] = {2, 1, 8, 3}
Output: A1[] = {2, 2, 1, 1, 8, 8, 3, 5, 6, 7, 9}
```

The code should handle all cases like number of elements in A2[] may be more or less compared to A1[]. A2[] may have some elements which may not be there in A1[] and vice versa is also possible.

Source: [Amazon Interview | Set 110 \(On-Campus\)](#)

Method 1 (Using Sorting and Binary Search)

Let size of A1[] be m and size of A2[] be n.

- 1) Create a temporary array temp of size m and copy contents of A1[] to it.
- 2) Create another array visited[] and initialize all entries in it as false. visited[] is used to mark those elements in temp[] which are copied to A1[].
- 3) Sort temp[]
- 4) Initialize the output index ind as 0.
- 5) Do following for every element of A2[i] in A2[]
..a) Binary search for all occurrences of A2[i] in temp[], if present then copy all occurrences to A1[ind] and increment ind. Also mark the copied elements visited[]
- 6) Copy all unvisited elements from temp[] to A1[].

Time complexity: The steps 1 and 2 require O(m) time. Step 3 requires O(mLogm) time. Step 5 requires O(nLogm) time. Therefore overall time complexity is O(m + nLogm).

Thanks to [vivek](#) for suggesting this method. Following is C++ implementation of above algorithm

```
// A C++ program to sort an array according to the order defined
// by another array
#include <iostream>
#include <algorithm>
using namespace std;

/* A Binary Search based function to find index of FIRST occurrence
   of x in arr[]. If x is not present, then it returns -1 */
int first(int arr[], int low, int high, int x, int n)
{
    if (high >= low)
    {
        int mid = low + (high-low)/2; /* (low + high)/2; */
        if ((mid == 0 || x > arr[mid-1]) && arr[mid] == x)
            return mid;
        if (x > arr[mid])
            return first(arr, (mid + 1), high, x, n);
        return first(arr, low, (mid -1), x, n);
    }
    return -1;
}

// Sort A1[0..m-1] according to the order defined by A2[0..n-1].
void sortAccording(int A1[], int A2[], int m, int n)
{
    // The temp array is used to store a copy of A1[] and visited[]
    // is used mark the visited elements in temp[].
    int temp[m], visited[m];
    for (int i=0; i<m; i++)
    {
        temp[i] = A1[i];
        visited[i] = 0;
    }

    // Sort elements in temp
    sort(temp, temp + m);

    int ind = 0; // for index of output which is sorted A1[]

    // Consider all elements of A2[], find them in temp[]
    // and copy to A1[] in order.
    for (int i=0; i<n; i++)
    {
        // Find index of the first occurrence of A2[i] in temp
        int f = first(temp, 0, m-1, A2[i], m);
        if (f != -1)
            A1[ind] = temp[f];
        ind++;
    }
}
```

```

// If not present, no need to proceed
if (f == -1) continue;

// Copy all occurrences of A2[i] to A1[]
for (int j = f; (j < m && temp[j] == A2[i]); j++)
{
    A1[ind++] = temp[j];
    visited[j] = 1;
}
}

// Now copy all items of temp[] which are not present in A2[]
for (int i=0; i < m; i++)
    if (visited[i] == 0)
        A1[ind++] = temp[i];
}

// Utility function to print an array
void printArray(int arr[], int n)
{
    for (int i=0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above function.
int main()
{
    int A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8};
    int A2[] = {2, 1, 8, 3};
    int m = sizeof(A1)/sizeof(A1[0]);
    int n = sizeof(A2)/sizeof(A2[0]);
    cout << "Sorted array is \n";
    sortAccording(A1, A2, m, n);
    printArray(A1, m);
    return 0;
}

```

Output:

```

Sorted array is
2 2 1 1 8 8 3 5 6 7 9

```

Method 2 (Using Self-Balancing Binary Search Tree)

We can also use a selfbalancing BST like [AVL Tree](#), [Red Black Tree](#), etc. Following are detailed steps.

- 1) Create a self balancing BST of all elements in A1[]. In every node of BST, also keep track of count of occurrences of the key and a bool field visited which is initialized as false for all nodes.
- 2) Initialize the output index ind as 0.
- 3) Do following for every element of A2[i] in A2[]
 - ..a) Search for A2[i] in the BST, if present then copy all occurrences to A1[ind] and increment ind. Also mark the copied elements visited in the BST node.
- 4) Do an inorder traversal of BST and copy all unvisited keys to A1[].

Time Complexity of this method is same as the previous method. Note that in a selfbalancing Binary Search Tree, all operations require logm time.

Method 3 (Use Hashing)

1. Loop through A1[], store the count of every number in a HashMap (key: number, value: count of number).
2. Loop through A2[], check if it is present in HashMap, if so, put in output array that many times and remove the number from HashMap.
3. Sort the rest of the numbers present in HashMap and put in output array.

Thanks to [Anurag Singh](#) for suggesting this method.

The steps 1 and 2 on average take O(m+n) time under the assumption that we have a good hashing function that takes O(1) time for insertion and search on average. The third step takes O(pLogp) time where p is the number of elements remained after considering elements of A2[].

Method 4 (By Writing a Customized Compare Method)

We can also customize compare method of a sorting algorithm to solve the above problem. For example [qsort\(\) in C allows us to pass our own customized compare method](#).

1. If num1 and num2 both are in A2 then number with lower index in A2 will be treated smaller than other.
2. If only one of num1 or num2 present in A2, then that number will be treated smaller than the other which doesn't present in A2.
3. If both are not in A2, then natural ordering will be taken.

Time complexity of this method is O(mnLogm) if we use a O(nLogn) time complexity sorting algorithm. We can improve time complexity to O(mLogm) by using a Hashing instead of doing linear search.

Following is C implementation of this method.

```
// A C++ program to sort an array according to the order defined
// by another array
#include <stdio.h>
#include <stdlib.h>

// A2 is made global here so that it can be accessed by compareByA2()
// The syntax of qsort() allows only two parameters to compareByA2()
int A2[5];
int size = 5; // size of A2[]

int search(int key)
{
    int i=0, idx = 0;
    for (i=0; i<size; i++)
        if (A2[i] == key)
            return i;
    return -1;
}

// A custom compare method to compare elements of A1[] according
// to the order defined by A2[].
int compareByA2(const void * a, const void * b)
{
    int idx1 = search(*(int*)a);
    int idx2 = search(*(int*)b);
    if (idx1 != -1 && idx2 != -1)
        return idx1 - idx2;
    else if(idx1 != -1)
        return -1;
    else if(idx2 != -1)
        return 1;
    else
        return ( *(int*)a - *(int*)b );
}

// This method mainly uses qsort to sort A1[] according to A2[]
void sortA1ByA2(int A1[], int size1)
{
    qsort(A1, size1, sizeof (int), compareByA2);
}

// Driver program to test above function
int main(int argc, char *argv[])
{
    int A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8, 7, 5, 6, 9, 7, 5};

    //A2[] = {2, 1, 8, 3, 4};
    A2[0] = 2;
    A2[1] = 1;
    A2[2] = 8;
    A2[3] = 3;
    A2[4] = 4;
    int size1 = sizeof(A1)/sizeof(A1[0]);

    sortA1ByA2(A1, size1);

    printf("Sorted Array is ");
    int i;
    for (i=0; i<size1; i++)
        printf("%d ", A1[i]);
    return 0;
}
```

Output:

```
Sorted Array is 2 2 1 1 8 8 3 5 5 5 6 6 7 7 7 9 9
```

This method is based on comments by readers (Xinuo Chen, Pranay Doshi and javakurious) and compiled by Anurag Singh.

Maximum Sum Path in Two Arrays

Given two sorted arrays such the arrays may have some common elements. Find the sum of the maximum sum path to reach from beginning of any array to end of any of the two arrays. We can switch from one array to another array only at common elements.

Expected time complexity is O(m+n) where m is the number of elements in ar1[] and n is the number of elements in ar2[].

Examples:

```
Input: ar1[] = {2, 3, 7, 10, 12}, ar2[] = {1, 5, 7, 8}
Output: 35
35 is sum of 1 + 5 + 7 + 10 + 12.
We start from first element of arr2 which is 1, then we
move to 5, then 7. From 7, we switch to ar1 (7 is common)
and traverse 10 and 12.
```

```
Input: ar1[] = {10, 12}, ar2 = {5, 7, 9}
Output: 22
22 is sum of 10 and 12.
Since there is no common element, we need to take all
elements from the array with more sum.
```

```
Input: ar1[] = {2, 3, 7, 10, 12, 15, 30, 34}
       ar2[] = {1, 5, 7, 8, 10, 15, 16, 19}
Output: 122
122 is sum of 1, 5, 7, 8, 10, 12, 15, 30, 34
```

The idea is to do something similar to merge process of [merge sort](#). We need to calculate sums of elements between all common points for both arrays. Whenever we see a common point, we compare the two sums and add the maximum of two to the result. Following are detailed steps.

1) Initialize result as 0. Also initialize two variables sum1 and sum2 as 0. Here sum1 and sum2 are used to store sum of element in ar1[] and ar2[] respectively. These sums are between two common points.

2) Now run a loop to traverse elements of both arrays. While traversing compare current elements of ar1[] and ar2[].

2.a) If current element of ar1[] is smaller than current element of ar2[], then update sum1, else if current element of ar2[] is smaller, then update sum2.

2.b) If current element of ar1[] and ar2[] are same, then take the maximum of sum1 and sum2 and add it to the result. Also add the common element to the result.

Following is C++ implementation of above approach.

C++

```
#include<iostream>
using namespace std;

// Utility function to find maximum of two integers
int max(int x, int y) { return (x > y)? x : y; }

// This function returns the sum of elements on maximum path
// from beginning to end
int maxPathSum(int ar1[], int ar2[], int m, int n)
{
    // initialize indexes for ar1[] and ar2[]
    int i = 0, j = 0;

    // Initialize result and current sum through ar1[] and ar2[].
    int result = 0, sum1 = 0, sum2 = 0;

    // Below 3 loops are similar to merge in merge sort
    while (i < m && j < n)
    {
        // Add elements of ar1[] to sum1
        if (ar1[i] < ar2[j])
            sum1 += ar1[i++];

        // Add elements of ar2[] to sum2
        else if (ar1[i] > ar2[j])
            sum2 += ar2[j++];

        else // we reached a common point
        {
            // Take the maximum of two sums and add to result
            result += max(sum1, sum2);
            sum1 = sum2 = 0;
        }
    }

    // Add remaining elements of ar1[] to sum1
    while (i < m)
        sum1 += ar1[i++];

    // Add remaining elements of ar2[] to sum2
    while (j < n)
        sum2 += ar2[j++];

    // Take the maximum of two sums and add to result
    result += max(sum1, sum2);

    return result;
}
```

```

// Update sum1 and sum2 for elements after this
// intersection point
sum1 = 0, sum2 = 0;

// Keep updating result while there are more common
// elements
while (i < m && j < n && ar1[i] == ar2[j])
{
    result = result + ar1[i++];
    j++;
}
}

// Add remaining elements of ar1[]
while (i < m)
    sum1 += ar1[i++];

// Add remaining elements of ar2[]
while (j < n)
    sum2 += ar2[j++];

// Add maximum of two sums of remaining elements
result += max(sum1, sum2);

return result;
}

// Driver program to test above function
int main()
{
    int ar1[] = {2, 3, 7, 10, 12, 15, 30, 34};
    int ar2[] = {1, 5, 7, 8, 10, 15, 16, 19};
    int m = sizeof(ar1)/sizeof(ar1[0]);
    int n = sizeof(ar2)/sizeof(ar2[0]);
    cout << "Maximum sum path is "
         << maxPathSum(ar1, ar2, m, n);
    return 0;
}

```

Python

```

# Python program to find maximum sum path

# This function returns the sum of elements on maximum path from
# beginning to end
def maxPathSum(ar1, ar2, m, n):

    # initialize indexes for ar1[] and ar2[]
    i, j = 0, 0

    # Initialize result and current sum through ar1[] and ar2[]
    result, sum1, sum2 = 0, 0, 0

    # Below 3 loops are similar to merge in merge sort
    while (i < m and j < n):

        # Add elements of ar1[] to sum1
        if ar1[i] < ar2[j]:
            sum1 += ar1[i]
            i+=1

        # Add elements of ar2[] to sum1
        elif ar1[i] > ar2[j]:
            sum2 += ar2[j]
            j+=1

        else:   # we reached a common point

            # Take the maximum of two sums and add to result
            result += max(sum1, sum2)

            # Update sum1 and sum2 for elements after this intersection point
            sum1, sum2 = 0, 0

            # Keep updating result while there are more common elements
            while (i < m and j < n and ar1[i]==ar2[j]):
                result += ar1[i]
                i+=1

```

```

j+=1

# Add remaining elements of ar1[]
while i < m:
    sum1 += ar1[i]
    i+=1
# Add remaining elements of b[]
while j < n:
    sum2 += ar2[j]
    j+=1

# Add maximum of two sums of remaining elements
result += max(sum1,sum2)

return result

# Driver function
ar1 = [2, 3, 7, 10, 12, 15, 30, 34]
ar2 = [1, 5, 7, 8, 10, 15, 16, 19]
m = len(ar1)
n = len(ar2)
print "Maximum sum path is", maxPathSum(ar1, ar2, m, n)

# This code is contributed by __Devesh Agrawal__

```

Maximum sum path is 122

Time complexity: In every iteration of while loops, we process an element from either of the two arrays. There are total $m + n$ elements. Therefore, time complexity is $O(m+n)$.

Check if a given array contains duplicate elements within k distance from each other

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than size of array. Write a function that returns true if array contains duplicates within k distance.

Examples:

Input: $k = 3$, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}

Output: false

All duplicates are more than k distance away.

Input: $k = 3$, arr[] = {1, 2, 3, 1, 4, 5}

Output: true

1 is repeated at distance 3.

Input: $k = 3$, arr[] = {1, 2, 3, 4, 5}

Output: false

Input: $k = 3$, arr[] = {1, 2, 3, 4, 4}

Output: true

A **Simple Solution** is to run two loops. The outer loop picks every element $\text{arr}[i]$ as a starting element, the inner loop compares all elements which are within k distance of $\text{arr}[i]$. The time complexity of this solution is $O(kn)$.

We can solve this problem in $O(n)$ time using Hashing. The idea is to one by add elements to hash. We also remove elements which are at more than k distance from current element. Following is detailed algorithm.

- 1) Create an empty hashtable.
- 2) Traverse all elements from left from right. Let the current element be $\text{arr}[i]$
 - a) If current element $\text{arr}[i]$ is present in hashtable, then return true.
 - b) Else add $\text{arr}[i]$ to hash and remove $\text{arr}[i-k]$ from hash if i is greater than or equal to k

```
/* Java program to Check if a given array contains duplicate
elements within k distance from each other */
import java.util.*;

class Main
{
    static boolean checkDuplicatesWithinK(int arr[], int k)
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the input array
        for (int i=0; i<arr.length; i++)
        {
            // If already present n hash, then we found
            // a duplicate within k distance
            if (set.contains(arr[i]))
                return true;

            // Add this item to hashset
            set.add(arr[i]);

            // Remove the k+1 distant item
            if (i >= k)
                set.remove(arr[i-k]);
        }
        return false;
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 5, 3, 4, 3, 5, 6};
        if (checkDuplicatesWithinK(arr, 3))
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

Output:

Yes

Sort an array in wave form

Given an unsorted array of integers, sort the array into a wave like array. An array arr[0..n-1] is sorted in wave form if arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= ..

Examples:

```
Input: arr[] = {10, 5, 6, 3, 2, 20, 100, 80}
Output: arr[] = {10, 5, 6, 2, 20, 3, 100, 80} OR
        {20, 5, 10, 2, 80, 6, 100, 3} OR
        any other array that is in wave form
```

```
Input: arr[] = {20, 10, 8, 6, 4, 2}
Output: arr[] = {20, 8, 10, 4, 6, 2} OR
        {10, 8, 20, 2, 6, 4} OR
        any other array that is in wave form
```

```
Input: arr[] = {2, 4, 6, 8, 10, 20}
Output: arr[] = {4, 2, 8, 6, 20, 10} OR
        any other array that is in wave form
```

```
Input: arr[] = {3, 6, 5, 10, 7, 20}
Output: arr[] = {6, 3, 10, 5, 20, 7} OR
        any other array that is in wave form
```

A **Simple Solution** is to use sorting. First sort the input array, then swap all adjacent elements.

For example, let the input array be {3, 6, 5, 10, 7, 20}. After sorting, we get {3, 5, 6, 7, 10, 20}. After swapping adjacent elements, we get {5, 3, 7, 6, 20, 10}.

Below are implementations of this simple approach.

C++

```
// A C++ program to sort an array in wave form using a sorting function
#include<iostream>
#include<algorithm>
using namespace std;

// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e.,
// arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]..
void sortInWave(int arr[], int n)
{
    // Sort the input array
    sort(arr, arr+n);

    // Swap adjacent elements
    for (int i=0; i<n-1; i += 2)
        swap(&arr[i], &arr[i+1]);
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Python

```
# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

    #sort the array
```

```

arr.sort()

# Swap adjacent elements
for i in range(0,n-1,2):
    arr[i], arr[i+1] = arr[i+1], arr[i]

# Driver program
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print arr[i],

# This code is contributed by _Devesh Agrawal_

```

2 1 10 5 49 23 90

The time complexity of the above solution is $O(n \log n)$ if a $O(n \log n)$ sorting algorithm like [Merge Sort](#), [Heap Sort](#), .. etc is used.

This can be done in **$O(n)$ time by doing a single traversal** of given array. The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don't need to worry about odd positioned element. Following are simple steps.

- 1) Traverse all even positioned elements of input array, and do following.
 - a) If current element is smaller than previous odd element, swap previous and current.
 - b) If current element is smaller than next odd element, swap next and current.

Below are implementations of above simple algorithm.

C++

```

// A O(n) program to sort an input array in wave form
#include<iostream>
using namespace std;

// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=
// arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....
void sortInWave(int arr[], int n)
{
    // Traverse all even elements
    for (int i = 0; i < n; i+=2)
    {
        // If current even element is smaller than previous
        if (i>0 && arr[i-1] > arr[i] )
            swap(&arr[i], &arr[i-1]);

        // If current even element is smaller than next
        if (i<n-1 && arr[i] < arr[i+1] )
            swap(&arr[i], &arr[i + 1]);
    }
}

```

```

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}

```

Python

```

# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

```

```
# Traverse all even elements
for i in range(0, n, 2):

    # If current even element is smaller than previous
    if (i> 0 and arr[i] < arr[i-1]):
        arr[i],arr[i-1]=arr[i-1],arr[i]

    # If current even element is smaller than next
    if (i < n-1 and arr[i] < arr[i+1]):
        arr[i],arr[i+1]=arr[i+1],arr[i]

# Driver program
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print arr[i],
```

This code is contributed by __Devesh Agrawal__

Output:

90 10 49 1 5 2 23

Kth Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the kth smallest element in the given array. It is given that all array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 3

Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 4

Output: 10

We have discussed a similar [problem to print k largest elements](#).

Method 1 (Simple Solution)

A Simple Solution is to sort the given array using a O(nlogn) sorting algorithm like [Merge Sort](#), [Heap Sort](#), etc and return the element at index k-1 in the sorted array. Time Complexity of this solution is O(nLogn).

```
// Simple C++ program to find k'th smallest element
#include<iostream>
#include<algorithm>
using namespace std;

// Function to return k'th smallest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Sort the given array
    sort(arr, arr+n);

    // Return k'th element in the sorted array
    return arr[k-1];
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 19};
    int n = sizeof(arr)/sizeof(arr[0]), k = 2;
    cout << "K'th smallest element is " << kthSmallest(arr, n, k);
    return 0;
}
```

K'th smallest element is 5

Method 2 (Using Min Heap HeapSelect)

We can find kth smallest element in time complexity better than O(nLogn). A simple optimization is to create a [Min Heap](#) of the given n elements and call extractMin() k times.

The following is C++ implementation of above method.

```
// A C++ program to find k'th smallest element using min heap
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int a[], int size); // Constructor
    void MinHeapify(int i); // To minheapify subtree rooted with index i
    int parent(int i) { return (i-1)/2; }
    int left(int i) { return (2*i + 1); }
    int right(int i) { return (2*i + 2); }
}
```

```

int extractMin(); // extracts root (minimum) element
int getMin() { return harr[0]; } // Returns minimum
};

MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the minimum value.
    int root = harr[0];

    // If there are more than 1 items, move the last item to root
    // and call heapify.
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        MinHeapify(0);
    }
    heap_size--;

    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th smallest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of n elements: O(n) time
    MinHeap mh(arr, n);

    // Do extract min (k-1) times
    for (int i=0; i<k-1; i++)
        mh.extractMin();

    // Return root
    return mh.getMin();
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 19};
}

```

```

int n = sizeof(arr)/sizeof(arr[0]), k = 2;
cout << "K'th smallest element is " << kthSmallest(arr, n, k);
return 0;
}

```

Output:

K'th smallest element is 5

Time complexity of this solution is O(n + kLogn).

Method 3 (Using Max-Heap)

We can also use Max Heap for finding the kth smallest element. Following is algorithm.

1) Build a Max-Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.

a) If the element is less than the root then make it root and call heapify for MH

b) Else ignore it.

// The step 2 is O((n-k)*logk)

3) Finally, root of the MH is the kth smallest element.

Time complexity of this solution is O(k + (n-k)*Logk)

The following is C++ implementation of above algorithm

```

// A C++ program to find k'th smallest element using max heap
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Max Heap
class MaxHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of max heap
    int heap_size; // Current number of elements in max heap
public:
    MaxHeap(int a[], int size); // Constructor
    void maxHeapify(int i); // To maxHeapify subtree rooted with index i
    int parent(int i) { return (i-1)/2; }
    int left(int i) { return (2*i + 1); }
    int right(int i) { return (2*i + 2); }

    int extractMax(); // extracts root (maximum) element
    int getMax() { return harr[0]; } // Returns maximum

    // to replace root with new node x and heapify() new root
    void replaceMax(int x) { harr[0] = x; maxHeapify(0); }
};

MaxHeap::MaxHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        maxHeapify(i);
        i--;
    }
}

// Method to remove maximum element (or root) from max heap
int MaxHeap::extractMax()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the maximum value.
    int root = harr[0];

    // If there are more than 1 items, move the last item to root

```

```

// and call heapify.
if (heap_size > 1)
{
    harr[0] = harr[heap_size-1];
    maxHeapify(0);
}
heap_size--;

return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MaxHeap::maxHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int largest = i;
    if (l < heap_size && harr[l] > harr[i])
        largest = l;
    if (r < heap_size && harr[r] > harr[largest])
        largest = r;
    if (largest != i)
    {
        swap(&harr[i], &harr[largest]);
        maxHeapify(largest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th largest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of first k elements: O(k) time
    MaxHeap mh(arr, k);

    // Process remaining n-k elements. If current element is
    // smaller than root, replace root with current element
    for (int i=k; i<n; i++)
        if (arr[i] < mh.getMax())
            mh.replaceMax(arr[i]);

    // Return root
    return mh.getMax();
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 19};
    int n = sizeof(arr)/sizeof(arr[0]), k = 4;
    cout << "K'th smallest element is " << kthSmallest(arr, n, k);
    return 0;
}

```

Output:

K'th smallest element is 5

Method 4 (QuickSelect)

This is an optimization over method 1 if [QuickSort](#) is used as a sorting algorithm in first step. In QuickSort, we pick a pivot element, then move the pivot element to its correct position and partition the array around it. The idea is, not to do complete quicksort, but stop at the point where pivot itself is kth smallest element. Also, not to recur for both left and right sides of pivot, but recur for one of them according to the position of pivot.

The worst case time complexity of this method is $O(n^2)$, but it works in $O(n)$ on average.

```

#include<iostream>
#include<climits>
using namespace std;

int partition(int arr[], int l, int r);

```

```

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around last element and get
        // position of pivot element in sorted array
        int pos = partition(arr, l, r);

        // If position is same as k
        if (pos-1 == k-1)
            return arr[pos];
        if (pos-1 > k-1) // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort(). It considers the last
// element as pivot and moves all smaller element to left of it
// and greater elements to right
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}

```

Output:

K'th smallest element is 5

There are two more solutions which are better than above discussed ones: One solution is to do randomized version of quickSelect() and other solution is worst case linear time algorithm (see the following posts).

[Kth Smallest/Largest Element in Unsorted Array | Set 2 \(Expected Linear Time\)](#)

[Kth Smallest/Largest Element in Unsorted Array | Set 3 \(Worst Case Linear Time\)](#)

References:

<http://www.ics.uci.edu/~eppstein/161/960125.html>

http://www.cs.rit.edu/~ib/Classes/CS515_Spring12-13/Slides/022-SelectMasterThm.pdf

Kth Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

[Kth Smallest/Largest Element in Unsorted Array | Set 1](#)

Given an array and a number k where k is smaller than size of array, we need to find the kth smallest element in the given array. It is given that all array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 4
Output: 10

We have discussed three different solutions [here](#).

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the [previous](#) post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, [rand\(\)](#) to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is C++ implementation of above Randomized QuickSelect.

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos-1 == k-1)
            return arr[pos];
        if (pos-1 > k-1) // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort(). It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
}
```

```

        i++;
    }
}
swap(&arr[i], &arr[r]);
return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] around the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}

```

Output:

K'th smallest element is 5

Time Complexity:

The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see [CLRS book](#) or [MIT video lecture](#) for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

Sources:

[MIT Video Lecture on Order Statistics, Median](#)

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

Kth Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

We recommend to read following posts as a prerequisite of this post.

[Kth Smallest/Largest Element in Unsorted Array | Set 1](#)

[Kth Smallest/Largest Element in Unsorted Array | Set 2 \(Expected Linear Time\)](#)

Given an array and a number k where k is smaller than size of array, we need to find the kth smallest element in the given array. It is given that all array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 3

Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 4

Output: 10

In [previous post](#), we discussed an expected linear time algorithm. In this post, a worst case linear time method is discussed. *The idea in this new method is similar to quickSelect(), we get worst case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on other side).* After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of pivot.

Following is complete algorithm.

kthSmallest(arr[0..n-1], k)

- 1) Divide arr[] into $\lceil n/5 \rceil$ groups where size of each group is 5 except possibly the last group which may have less than 5 elements.
 - 2) Sort the above created $\lceil n/5 \rceil$ groups and find median of all groups. Create an auxiliary array 'median[]' and store medians of all $\lceil n/5 \rceil$ groups in this median array.
- // Recursively call this method to find median of median[0.. $\lceil n/5 \rceil - 1$]
- 3) medOfMed = kthSmallest(median[0.. $\lceil n/5 \rceil - 1$], $n/10$)
- 4) Partition arr[] around medOfMed and obtain its position.
pos = partition(arr, n, medOfMed)
 - 5) If pos == k return medOfMed
 - 6) If pos < k return kthSmallest(arr[1..pos-1], k)
 - 7) If pos > k return kthSmallest(arr[pos+1..r], k-pos+1-1)

In above algorithm, last 3 steps are same as algorithm in [previous post](#). The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).

Following is C++ implementation of above algorithm.

```
// C++ implementation of worst case linear time algorithm
// to find k'th smallest element
#include<iostream>
#include<algorithm>
#include<climits>
using namespace std;

int partition(int arr[], int l, int r, int k);

// A simple function to find median of arr[]. This is called
// only for an array of size 5 in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n); // Sort the array
    return arr[n/2]; // Return middle element
}

// Returns k'th smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of size 5, calculate median
```

```

// of every group and store it in median[] array.
int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;
for (i=0; i<n/5; i++)
    median[i] = findMedian(arr+l+i*5, 5);
if (i*5 < n) //For last group with less than 5 elements
{
    median[i] = findMedian(arr+l+i*5, n%5);
    i++;
}

// Find median of all medians using recursive call.
// If median[] has only one element, then no need
// of recursive call
int medOfMed = (i == 1)? median[i-1]:
    kthSmallest(median, 0, i-1, i/2);

// Partition the array around a random element and
// get position of pivot element in sorted array
int pos = partition(arr, l, r, medOfMed);

// If position is same as k
if (pos-1 == k-1)
    return arr[pos];
if (pos-1 > k-1) // If position is more, recur for left
    return kthSmallest(arr, l, pos-1, k);

// Else recur for right subarray
return kthSmallest(arr, pos+1, r, k-pos+l-1);
}

// If k is more than number of elements in array
return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
            break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is "
        << kthSmallest(arr, 0, n-1, k);
    return 0;
}

```

Output:

K'th smallest element is 5

Time Complexity:

The worst case time complexity of the above algorithm is O(n). Let us analyze all steps.

The steps 1) and 2) take O(n) time as finding median of an array of size 5 takes O(1) time and there are n/5 arrays of size 5.

The step 3) takes T(n/5) time. The step 4 is standard partition and takes O(n) time.

The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls. The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.

How many elements are greater than medOfMed and how many are smaller?

At least half of the medians found in step 2 are greater than or equal to medOfMed. Thus, at least half of the n/5 groups contribute 3 elements that are greater than medOfMed, except for the one group that has fewer than 5 elements. Therefore, the number of elements greater than medOfMed is at least.

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$

Similarly, the number of elements that are less than medOfMed is at least 3n/10 - 6. In the worst case, the function recurs for at most n (3n/10 - 6) which is 7n/10 + 6 elements.

Note that $7n/10 + 6 < n$ for $n > 20$ and that any input of 80 or fewer elements requires O(1) time. We can therefore obtain the recurrence

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 80, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 80. \end{cases}$$

We show that the running time is linear by substitution. Assume that $T(n) \leq cn$ for some constant c and all $n > 80$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\leq cn/5 + c(7n/10 + 6) + O(n) \\ &\leq cn/5 + c + 7cn/10 + 6c + O(n) \\ &\leq 9cn/10 + 7c + O(n) \\ &\leq cn, \end{aligned}$$

since we can pick c large enough so that $c(n/10 - 7)$ is larger than the function described by the $O(n)$ term for all $n > 80$. The worst-case running time of is therefore linear (Source: <http://staff.ustc.edu.cn/~csl/graduate/algorithms/book6/chap10.htm>).

Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. Therefore, this algorithm doesn't work well in practical situations, [randomized quickSelect](#) works much better and preferred.

Sources:

[MIT Video Lecture on Order Statistics, Median](#)

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

<http://staff.ustc.edu.cn/~csl/graduate/algorithms/book6/chap10.htm>

Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array

Given an array of 0s and 1s, find the position of 0 to be replaced with 1 to get longest continuous sequence of 1s. Expected time complexity is O(n) and auxiliary space is O(1).

Example:

Input:
arr[] = {1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1}

Output:

Index 9

Assuming array index starts from 0, replacing 0 with 1 at index 9 causes the maximum continuous sequence of 1s.

Input:
arr[] = {1, 1, 1, 1, 0}

Output:

Index 4

A **Simple Solution** is to traverse the array, for every 0, count the number of 1s on both sides of it. Keep track of maximum count for any 0.

Finally return index of the 0 with maximum number of 1s around it. The time complexity of this solution is O(n²).

Using an **Efficient Solution**, the problem can solved in O(n) time. The idea is to keep track of three indexes, current index (*curr*), previous zero index (*prev_zero*) and previous to previous zero index (*prev_prev_zero*). Traverse the array, if current element is 0, calculate the difference between *curr* and *prev_prev_zero* (This difference minus one is the number of 1s around the *prev_zero*). If the difference between *curr* and *prev_prev_zero* is more than maximum so far, then update the maximum. Finally return index of the *prev_zero* with maximum difference.

Following are C++ and Java implementations of the above algorithm

C++

```
// C++ program to find Index of 0 to be replaced with 1 to get
// longest continuous sequence of 1s in a binary array
#include<iostream>
using namespace std;

// Returns index of 0 to be replaced with 1 to get longest
// continuous sequence of 1s. If there is no 0 in array, then
// it returns -1.
int maxOnesIndex(bool arr[], int n)
{
    int max_count = 0; // for maximum number of 1 around a zero
    int max_index; // for storing result
    int prev_zero = -1; // index of previous zero
    int prev_prev_zero = -1; // index of previous to previous zero

    // Traverse the input array
    for (int curr=0; curr<n; ++curr)
    {
        // If current element is 0, then calculate the difference
        // between curr and prev_prev_zero
        if (arr[curr] == 0)
        {
            // Update result if count of 1s around prev_zero is more
            if (curr - prev_prev_zero > max_count)
            {
                max_count = curr - prev_prev_zero;
                max_index = prev_zero;
            }
            // Update for next iteration
            prev_prev_zero = prev_zero;
            prev_zero = curr;
        }
    }

    // Check for the last encountered zero
    if (n-prev_prev_zero > max_count)
        max_index = prev_zero;

    return max_index;
}

// Driver program
int main()
{
```

```

bool arr[] = {1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1};
int n = sizeof(arr)/sizeof(arr[0]);
cout << "Index of 0 to be replaced is "
     << maxOnesIndex(arr, n);
return 0;
}

```

Java

```

// Java program to find Index of 0 to be replaced with 1 to get
// longest continuous sequence of 1s in a binary array

import java.io.*;

class Binary
{
    // Returns index of 0 to be replaced with 1 to get longest
    // continuous sequence of 1s. If there is no 0 in array, then
    // it returns -1.
    static int maxOnesIndex(int arr[], int n)
    {
        int max_count = 0; // for maximum number of 1 around a zero
        int max_index=0; // for storing result
        int prev_zero = -1; // index of previous zero
        int prev_prev_zero = -1; // index of previous to previous zero

        // Traverse the input array
        for (int curr=0; curr<n; ++curr)
        {
            // If current element is 0, then calculate the difference
            // between curr and prev_prev_zero
            if (arr[curr] == 0)
            {
                // Update result if count of 1s around prev_zero is more
                if (curr - prev_prev_zero > max_count)
                {
                    max_count = curr - prev_prev_zero;
                    max_index = prev_zero;
                }

                // Update for next iteration
                prev_prev_zero = prev_zero;
                prev_zero = curr;
            }
        }

        // Check for the last encountered zero
        if (n-prev_prev_zero > max_count)
            max_index = prev_zero;

        return max_index;
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1};
        int n = arr.length;
        System.out.println("Index of 0 to be replaced is "+
                           maxOnesIndex(arr, n));
    }
}
/* This code is contributed by Devesh Agrawal */

```

Index of 0 to be replaced is 9

Time Complexity: O(n)

Auxiliary Space: O(1)

Find the closest pair from two sorted arrays

Given two sorted arrays and a number x , find the pair whose sum is closest to x and the pair has an element from each array.

We are given two arrays $ar1[0..m-1]$ and $ar2[0..n-1]$ and a number x , we need to find the pair $ar1[i] + ar2[j]$ such that absolute value of $(ar1[i] + ar2[j] - x)$ is minimum.

Example:

Input: $ar1[] = \{1, 4, 5, 7\}$;
 $ar2[] = \{10, 20, 30, 40\}$;
 $x = 32$

Output: 1 and 30

Input: $ar1[] = \{1, 4, 5, 7\}$;
 $ar2[] = \{10, 20, 30, 40\}$;
 $x = 50$

Output: 7 and 40

A **Simple Solution** is to run two loops. The outer loop considers every element of first array and inner loop checks for the pair in second array. We keep track of minimum difference between $ar1[i] + ar2[j]$ and x .

We can do it in **O(n)** time using following steps.

1) Merge given two arrays into an auxiliary array of size $m+n$ using [merge process of merge sort](#). While merging keep another boolean array of size $m+n$ to indicate whether the current element in merged array is from $ar1[]$ or $ar2[]$.

2) Consider the merged array and use the [linear time algorithm to find the pair with sum closest to x](#). One extra thing we need to consider only those pairs which have one element from $ar1[]$ and other from $ar2[]$, we use the boolean array for this purpose.

Can we do it in a single pass and O(1) extra space?

The idea is to start from left side of one array and right side of another array, and use the algorithm same as step 2 of above approach. Following is detailed algorithm.

- 1) Initialize a variable $diff$ as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.
- 2) Initialize two index variables l and r in the given sorted array.
 - (a) Initialize first to the leftmost index in $ar1$: $l = 0$
 - (b) Initialize second the rightmost index in $ar2$: $r = n-1$
- 3) Loop while $l < m$ and $r \geq 0$
 - (a) If $abs(ar1[l] + ar2[r] - sum) < diff$ then update diff and result
 - (b) Else if($ar1[l] + ar2[r] < sum$) then $l++$
 - (c) Else $r--$
- 4) Print the result.

Following is C++ implementation of this approach.

```
// C++ program to find the pair from two sorted arrays such
// that the sum of pair is closest to a given number x
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

// ar1[0..m-1] and ar2[0..n-1] are two given sorted arrays
// and x is given number. This function prints the pair from
// both arrays such that the sum of the pair is closest to x.
void printClosest(int ar1[], int ar2[], int m, int n, int x)
{
    // Initialize the diff between pair sum and x.
    int diff = INT_MAX;

    // res_l and res_r are result indexes from ar1[] and ar2[]
    // respectively
    int res_l, res_r;

    // Start from left side of ar1[] and right side of ar2[]
    int l = 0, r = n-1;
    while (l < m && r >= 0)
    {
        // If this pair is closer to x than the previously
        // found closest, then update res_l, res_r and diff
        if (abs(ar1[l] + ar2[r] - x) < diff)
        {
            res_l = l;
            res_r = r;
            diff = abs(ar1[l] + ar2[r] - x);
        }
    }
}
```

```

}

// If sum of this pair is more than x, move to smaller
// side
if (ar1[l] + ar2[r] > x)
    r--;
else // move to the greater side
    l++;
}

// Print the result
cout << "The closest pair is [" << ar1[res_l] << ", "
    << ar2[res_r] << "] \n";
}

// Driver program to test above functions
int main()
{
    int ar1[] = {1, 4, 5, 7};
    int ar2[] = {10, 20, 30, 40};
    int m = sizeof(ar1)/sizeof(ar1[0]);
    int n = sizeof(ar2)/sizeof(ar2[0]);
    int x = 38;
    printClosest(ar1, ar2, m, n, x);
    return 0;
}

```

Output:

The closest pair is [7, 30]

Given a sorted array and a number x, find the pair in array whose sum is closest to x

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

Examples:

Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54
Output: 22 and 30

Input: arr[] = {1, 3, 4, 7, 10}, x = 15
Output: 4 and 10

A simple solution is to consider every pair and keep track of closest pair (absolute difference between pair sum and x is minimum). Finally print the closest pair. Time complexity of this solution is $O(n^2)$

An efficient solution can find the pair in $O(n)$ time. The idea is similar to method 2 of [this](#) post. Following is detailed algorithm.

- 1) Initialize a variable diff as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.
- 2) Initialize two index variables l and r in the given sorted array.
 - (a) Initialize first to the leftmost index: l = 0
 - (b) Initialize second the rightmost index: r = n-1
- 3) Loop while l < r.
 - (a) If $\text{abs}(\text{arr}[l] + \text{arr}[r] - \text{sum}) < \text{diff}$ then update diff and result
 - (b) Else if($\text{arr}[l] + \text{arr}[r] < \text{sum}$) then l++
 - (c) Else r--

Following is C++ implementation of above algorithm.

C++

```
// Simple C++ program to find the pair with sum closest to a given no.
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

// Prints the pair with sum closest to x
void printClosest(int arr[], int n, int x)
{
    int res_l, res_r; // To store indexes of result pair

    // Initialize left and right indexes and difference between
    // pair sum and x
    int l = 0, r = n-1, diff = INT_MAX;

    // While there are elements between l and r
    while (r > l)
    {
        // Check if this pair is closer than the closest pair so far
        if (abs(arr[l] + arr[r] - x) < diff)
        {
            res_l = l;
            res_r = r;
            diff = abs(arr[l] + arr[r] - x);
        }

        // If this pair has more sum, move to smaller values.
        if (arr[l] + arr[r] > x)
            r--;
        else // Move to larger values
            l++;
    }

    cout << " The closest pair is " << arr[res_l] << " and " << arr[res_r];
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
    int n = sizeof(arr)/sizeof(arr[0]);
    printClosest(arr, n, x);
    return 0;
}
```

Java

```
// Java program to find pair with sum closest to x
import java.io.*;
import java.util.*;
import java.lang.Math;

class CloseSum {

    // Prints the pair with sum closest to x
    static void printClosest(int arr[], int n, int x)
    {
        int res_l=0, res_r=0; // To store indexes of result pair

        // Initialize left and right indexes and difference between
        // pair sum and x
        int l = 0, r = n-1, diff = Integer.MAX_VALUE;

        // While there are elements between l and r
        while (r > l)
        {
            // Check if this pair is closer than the closest pair so far
            if (Math.abs(arr[l] + arr[r] - x) < diff)
            {
                res_l = l;
                res_r = r;
                diff = Math.abs(arr[l] + arr[r] - x);
            }

            // If this pair has more sum, move to smaller values.
            if (arr[l] + arr[r] > x)
                r--;
            else // Move to larger values
                l++;
        }

        System.out.println(" The closest pair is "+arr[res_l]+" and "+ arr[res_r]);
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
        int n = arr.length;
        printClosest(arr, n, x);
    }
}

/*This code is contributed by Devesh Agrawal*/
```

The closest pair is 22 and 30

Count 1s in a sorted binary array

Given a binary array sorted in non-increasing order, count the number of 1s in it.

Examples:

Input: arr[] = {1, 1, 0, 0, 0, 0, 0}
Output: 2

Input: arr[] = {1, 1, 1, 1, 1, 1, 1}
Output: 7

Input: arr[] = {0, 0, 0, 0, 0, 0, 0}
Output: 0

A simple solution is to linearly traverse the array. The time complexity of the simple solution is O(n). We can use [Binary Search](#) to find count in O(Logn) time. The idea is to look for last occurrence of 1 using Binary Search. Once we find the index last occurrence, we return index + 1 as count.

The following is C++ implementation of above idea.

C++

```
// C++ program to count one's in a boolean array
#include <iostream>
using namespace std;

/* Returns counts of 1's in arr[low..high]. The array is
   assumed to be sorted in non-increasing order */
int countOnes(bool arr[], int low, int high)
{
    if (high >= low)
    {
        // get the middle index
        int mid = low + (high - low)/2;

        // check if the element at middle index is last 1
        if ((mid == high || arr[mid+1] == 0) && (arr[mid] == 1))
            return mid+1;

        // If element is not last 1, recur for right side
        if (arr[mid] == 1)
            return countOnes(arr, (mid + 1), high);

        // else recur for left side
        return countOnes(arr, low, (mid -1));
    }
    return 0;
}

/* Driver program to test above functions */
int main()
{
    bool arr[] = {1, 1, 1, 1, 0, 0, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Count of 1's in given array is " << countOnes(arr, 0, n-1);
    return 0;
}
```

Python

```
# Python program to count one's in a boolean array

# Returns counts of 1's in arr[low..high]. The array is
# assumed to be sorted in non-increasing order
def countOnes(arr,low,high):

    if high>=low:

        # get the middle index
        mid = low + (high-low)/2

        # check if the element at middle index is last 1
        if ((mid == high or arr[mid+1]==0) and (arr[mid]==1)):

            return mid+1

        # If element is not last 1, recur for right side
```

```
if arr[mid]==1:  
    return countOnes(arr, (mid+1), high)  
  
# else recur for left side  
return countOnes(arr, low, mid-1)  
  
return 0  
  
# Driver function  
arr=[1, 1, 1, 1, 0, 0, 0]  
print "Count of 1's in given array is",countOnes(arr, 0 , len(arr)-1)  
  
# This code is contributed by __Devesh Agrawal__
```

Output:

Count of 1's in given array is 4

Time complexity of the above solution is O(Logn)

Print All Distinct Elements of a given integer array

Given an integer array, print all distinct elements in array. The given array may contain duplicates and the output should print every element only once. The given array is not sorted.

Examples:

Input: arr[] = {12, 10, 9, 45, 2, 10, 10, 45}
Output: 12, 10, 9, 45, 2

Input: arr[] = {1, 2, 3, 4, 5}
Output: 1, 2, 3, 4, 5

Input: arr[] = {1, 1, 1, 1, 1}
Output: 1

A **Simple Solution** is to use two nested loops. The outer loop picks an element one by one starting from the leftmost element. The inner loop checks if the element is present on left side of it. If present, then ignores the element, else prints the element. Following is C++ implementation of the simple algorithm.

```
// C++ program to print all distinct elements in a given array
#include <iostream>
#include <algorithm>
using namespace std;

void printDistinct(int arr[], int n)
{
    // Pick all elements one by one
    for (int i=0; i<n; i++)
    {
        // Check if the picked element is already printed
        int j;
        for (j=0; j<i; j++)
            if (arr[i] == arr[j])
                break;

        // If not printed earlier, then print it
        if (i == j)
            cout << arr[i] << " ";
    }
}

// Driver program to test above function
int main()
{
    int arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    printDistinct(arr, n);
    return 0;
}
```

Output:

6 10 5 4 9 120

Time Complexity of above solution is $O(n^2)$. We can **Use Sorting** to solve the problem in $O(n \log n)$ time. The idea is simple, first sort the array so that all occurrences of every element become consecutive. Once the occurrences become consecutive, we can traverse the sorted array and print distinct elements in $O(n)$ time. Following is C++ implementation of the idea.

```
// C++ program to print all distinct elements in a given array
#include <iostream>
#include <algorithm>
using namespace std;

void printDistinct(int arr[], int n)
{
    // First sort the array so that all occurrences become consecutive
    sort(arr, arr + n);

    // Traverse the sorted array
    for (int i=0; i<n; i++)
    {
        // Move the index ahead while there are duplicates
        while (i < n-1 && arr[i] == arr[i+1])
            i++;

        // print last occurrence of the current element
        cout << arr[i] << " ";
    }
}
```

```

    }
}

// Driver program to test above function
int main()
{
    int arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    printDistinct(arr, n);
    return 0;
}

```

Output:

```
4 5 6 9 10 120
```

We can Use [Hashing](#) to solve this in O(n) time on average. The idea is to traverse the given array from left to right and keep track of visited elements in a hash table. Following is Java implementation of the idea.

```

/* Java program to print all distinct elements of a given array */
import java.util.*;

class Main
{
    // This function prints all distinct elements
    static void printDistinct(int arr[])
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the input array
        for (int i=0; i<arr.length; i++)
        {
            // If not present, then put it in hashtable and print it
            if (!set.contains(arr[i]))
            {
                set.add(arr[i]);
                System.out.print(arr[i] + " ");
            }
        }
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 5, 3, 4, 3, 5, 6};
        printDistinct(arr);
    }
}

```

Output:

```
10 5 3 4 6
```

One more advantage of hashing over sorting is, the elements are printed in same order as they are in input array.

Construct an array from its pair-sum array

Given a pair-sum array and size of the original array (n), construct the original array.

A pair-sum array for an array is the array that contains sum of all pairs in ordered form. For example pair-sum array for $\text{arr}[] = \{6, 8, 3, 4\}$ is $\{14, 9, 10, 11, 12, 7\}$.

In general, pair-sum array for $\text{arr}[0..n-1]$ is $\{\text{arr}[0]+\text{arr}[1], \text{arr}[0]+\text{arr}[2], \dots, \text{arr}[1]+\text{arr}[2], \text{arr}[1]+\text{arr}[3], \dots, \text{arr}[2]+\text{arr}[3], \text{arr}[2]+\text{arr}[4], \dots, \text{arr}[n-2]+\text{arr}[n-1]\}$.

Given a pair-sum array, construct the original array.

Let the given array be $\text{pair}[]$ and let there be n elements in original array. If we take a look at few examples, we can observe that $\text{arr}[0]$ is half of $\text{pair}[0] + \text{pair}[1]$ $\text{pair}[n-1]$. Note that the value of $\text{pair}[0] + \text{pair}[1]$ $\text{pair}[n-1]$ is $(\text{arr}[0] + \text{arr}[1]) + (\text{arr}[0] + \text{arr}[2])$ ($\text{arr}[1] + \text{arr}[2]$).

Once we have evaluated $\text{arr}[0]$, we can evaluate other elements by subtracting $\text{arr}[0]$. For example $\text{arr}[1]$ can be evaluated by subtracting $\text{arr}[0]$ from $\text{pair}[0]$, $\text{arr}[2]$ can be evaluated by subtracting $\text{arr}[0]$ from $\text{pair}[1]$.

Following are C++ and Java implementations of the above idea.

C++

```
#include <iostream>
using namespace std;

// Fills element in arr[] from its pair sum array pair[].
// n is size of arr[]
void constructArr(int arr[], int pair[], int n)
{
    arr[0] = (pair[0]+pair[1]-pair[n-1]) / 2;
    for (int i=1; i<n; i++)
        arr[i] = pair[i-1]-arr[0];
}

// Driver program to test above function
int main()
{
    int pair[] = {15, 13, 11, 10, 12, 10, 9, 8, 7, 5};
    int n = 5;
    int arr[n];
    constructArr(arr, pair, n);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Java

```
import java.io.*;

class PairSum {

    // Fills element in arr[] from its pair sum array pair[].
    // n is size of arr[]
    static void constructArr(int arr[], int pair[], int n)
    {
        arr[0] = (pair[0]+pair[1]-pair[n-1]) / 2;
        for (int i=1; i<n; i++)
            arr[i] = pair[i-1]-arr[0];
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int pair[] = {15, 13, 11, 10, 12, 10, 9, 8, 7, 5};
        int n = 5;
        int[] arr = new int[n];
        constructArr(arr, pair, n);
        for (int i = 0; i < n; i++)
            System.out.print(arr[i]+" ");
    }
}
/* This code is contributed by Devesh Agrawal */
```

8 7 5 3 2

Time complexity of constructArr() is O(n) where n is number of elements in arr[].

Find common elements in three sorted arrays

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

Examples:

```
ar1[] = {1, 5, 10, 20, 40, 80}
ar2[] = {6, 7, 20, 80, 100}
ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}
Output: 20, 80
```

```
ar1[] = {1, 5, 5}
ar2[] = {3, 4, 5, 5, 10}
ar3[] = {5, 5, 10, 20}
Output: 5, 5
```

A simple solution is to first find [intersection of two arrays](#) and store the intersection in a temporary array, then find the intersection of third array and temporary array. Time complexity of this solution is $O(n_1 + n_2 + n_3)$ where n_1 , n_2 and n_3 are sizes of $ar1[]$, $ar2[]$ and $ar3[]$ respectively.

The above solution requires extra space and two loops, we can find the common elements using a single loop and without extra space. The idea is similar to [intersection of two arrays](#). Like two arrays loop, we run a loop and traverse three arrays.

Let the current element traversed in $ar1[]$ be x , in $ar2[]$ be y and in $ar3[]$ be z . We can have following cases inside the loop.

- 1) If x , y and z are same, we can simply print any of them as common element and move ahead in all three arrays.
- 2) Else If $x < y$, we can move ahead in $ar1[]$ as x cannot be a common element
- 3) Else If $y < z$, we can move ahead in $ar2[]$ as y cannot be a common element
- 4) Else (We reach here when $x > y$ and $y > z$), we can simply move ahead in $ar3[]$ as z cannot be a common element.

Following are implementations of the above idea.

C++

```
// C++ program to print common elements in three arrays
#include <iostream>
using namespace std;

// This function prints common elements in ar1
int findCommon(int ar1[], int ar2[], int ar3[], int n1, int n2, int n3)
{
    // Initialize starting indexes for ar1[], ar2[] and ar3[]
    int i = 0, j = 0, k = 0;

    // Iterate through three arrays while all arrays have elements
    while (i < n1 && j < n2 && k < n3)
    {
        // If x = y and y = z, print any of them and move ahead
        // in all arrays
        if (ar1[i] == ar2[j] && ar2[j] == ar3[k])
        {
            cout << ar1[i] << " ";
            i++;
            j++;
            k++;
        }

        // x < y
        else if (ar1[i] < ar2[j])
            i++;

        // y < z
        else if (ar2[j] < ar3[k])
            j++;

        // We reach here when x > y and z < y, i.e., z is smallest
        else
            k++;
    }
}

// Driver program to test above function
int main()
{
    int ar1[] = {1, 5, 10, 20, 40, 80};
    int ar2[] = {6, 7, 20, 80, 100};
    int ar3[] = {3, 4, 15, 20, 30, 70, 80, 120};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    int n3 = sizeof(ar3)/sizeof(ar3[0]);

    cout << "Common Elements are ";
    findCommon(ar1, ar2, ar3, n1, n2, n3);
    return 0;
}
```

Python

```
# Python function to print common elements in three sorted arrays
def findCommon(ar1, ar2, ar3, n1, n2, n3):

    # Initialize starting indexes for ar1[], ar2[] and ar3[]
    i, j, k = 0, 0, 0

    # Iterate through three arrays while all arrays have elements
    # (i < n1 and j < n2 and k< n3):
    #
    # If x = y and y = z, print any of them and move ahead
    # in all arrays
    if (ar1[i] == ar2[j] and ar2[j] == ar3[k]):
        print ar1[i],
        i += 1
        j += 1
        k += 1

    # x < y
    elif ar1[i] < ar2[j]:
        i += 1

    # y < z
    elif ar2[j] < ar3[k]:
        j += 1

    # We reach here when x > y and z < y, i.e., z is smallest
    else:
        k += 1

#Driver program to check above function
ar1 = [1, 5, 10, 20, 40, 80]
ar2 = [6, 7, 20, 80, 100]
ar3 = [3, 4, 15, 20, 30, 70, 80, 120]
n1 = len(ar1)
n2 = len(ar2)
n3 = len(ar3)
print "Common elements are",
findCommon(ar1, ar2, ar3, n1, n2, n3)

# This code is contributed by __Devesh Agrawal__
```

Common Elements are 20 80

Time complexity of the above solution is $O(n1 + n2 + n3)$. In worst case, the largest sized array may have all small elements and middle sized array has all middle elements.

Find the first repeating element in an array of integers

Given an array of integers, find the first repeating element in it. We need to find the element that occurs more than once and whose index of first occurrence is smallest.

Examples:

Input: arr[] = {10, 5, 3, 4, 3, 5, 6}
Output: 5 [5 is the first element that repeats]

Input: arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10}
Output: 6 [6 is the first element that repeats]

A **Simple Solution** is to use two nested loops. The outer loop picks an element one by one, the inner loop checks whether the element is repeated or not. Once we find an element that repeats, we break the loops and print the element. Time Complexity of this solution is $O(n^2)$

We can **Use Sorting** to solve the problem in $O(n \log n)$ time. Following are detailed steps.

- 1) Copy the given array to an auxiliary array temp[].
- 2) Sort the temp array using a $O(n \log n)$ time sorting algorithm.
- 3) Scan the input array from left to right. For every element, [count its occurrences in temp\[\] using binary search](#). As soon as we find an element that occurs more than once, we return the element. This step can be done in $O(n \log n)$ time.

We can **Use Hashing** to solve this in $O(n)$ time on average. The idea is to traverse the given array from right to left and update the minimum index whenever we find an element that has been visited on right side. Thanks to Mohammad Shahid for suggesting this solution.

Following is Java implementation of this idea.

```
/* Java program to find first repeating element in arr[] */  
import java.util.*;  
  
class Main  
{  
    // This function prints the first repeating element in arr[]  
    static void printFirstRepeating(int arr[])  
    {  
        // Initialize index of first repeating element  
        int min = -1;  
  
        // Creates an empty hashset  
        HashSet<Integer> set = new HashSet<>();  
  
        // Traverse the input array from right to left  
        for (int i=arr.length-1; i>=0; i--)  
        {  
            // If element is already in hash set, update min  
            if (set.contains(arr[i]))  
                min = i;  
  
            else // Else add element to hash set  
                set.add(arr[i]);  
        }  
  
        // Print the result  
        if (min != -1)  
            System.out.println("The first repeating element is " + arr[min]);  
        else  
            System.out.println("There are no repeating elements");  
    }  
  
    // Driver method to test above method  
    public static void main (String[] args) throws java.lang.Exception  
    {  
        int arr[] = {10, 5, 3, 4, 3, 5, 6};  
        printFirstRepeating(arr);  
    }  
}
```

Output:

The first repeating element is 5

Find the smallest positive integer value that cannot be represented as sum of any subset of a given array

Given a sorted array (sorted in non-decreasing order) of positive numbers, find the smallest positive integer value that cannot be represented as sum of elements of any subset of given set.

Expected time complexity is O(n).

Examples:

Input: arr[] = {1, 3, 6, 10, 11, 15};
Output: 2

Input: arr[] = {1, 1, 1, 1};
Output: 5

Input: arr[] = {1, 1, 3, 4};
Output: 10

Input: arr[] = {1, 2, 5, 10, 20, 40};
Output: 4

Input: arr[] = {1, 2, 3, 4, 5, 6};
Output: 22

A **Simple Solution** is to start from value 1 and check all values one by one if they can sum to values in the given array. This solution is very inefficient as it reduces to [subset sum problem](#) which is a well known [NP Complete Problem](#).

We can solve this problem in **O(n)** time using a simple loop. Let the input array be arr[0..n-1]. We initialize the result as 1 (smallest possible outcome) and traverse the given array. Let the smallest element that cannot be represented by elements at indexes from 0 to (i-1) be res, there are following two possibilities when we consider element at index i:

1) We decide that res is the final result: If arr[i] is greater than res, then we found the gap which is res because the elements after arr[i] are also going to be greater than res.

2) The value of res is incremented after considering arr[i]: The value of res is incremented by arr[i] (why? If elements from 0 to (i-1) can represent 1 to res-1?, then elements from 0 to i can represent from 1 to res + arr[i] ? be adding arr[i] to all subsets that represent 1 to res)

Following is C++ implementation of above idea.

```
// C++ program to find the smallest positive value that cannot be
// represented as sum of subsets of a given sorted array
#include <iostream>
using namespace std;

// Returns the smallest number that cannot be represented as sum
// of subset of elements from set represented by sorted array arr[0..n-1]
int findSmallest(int arr[], int n)
{
    int res = 1; // Initialize result

    // Traverse the array and increment 'res' if arr[i] is
    // smaller than or equal to 'res'.
    for (int i = 0; i < n && arr[i] <= res; i++)
        res = res + arr[i];

    return res;
}

// Driver program to test above function
int main()
{
    int arr1[] = {1, 3, 4, 5};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    cout << findSmallest(arr1, n1) << endl;

    int arr2[] = {1, 2, 6, 10, 11, 15};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    cout << findSmallest(arr2, n2) << endl;

    int arr3[] = {1, 1, 1, 1};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    cout << findSmallest(arr3, n3) << endl;

    int arr4[] = {1, 1, 3, 4};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    cout << findSmallest(arr4, n4) << endl;
}
```

```
    return 0;  
}
```

Output:

```
2  
4  
5  
10
```

Time Complexity of above program is $O(n)$.

Rearrange an array such that arr[j] becomes i if arr[i] is j

Given an array of size n where all elements are in range from 0 to n-1, change contents of arr[] so that arr[i] = j is changed to arr[j] = i.

Examples:

Example 1:

Input: arr[] = {1, 3, 0, 2};
Output: arr[] = {2, 0, 3, 1};
Explanation for the above output.
Since arr[0] is 1, arr[1] is changed to 0
Since arr[1] is 3, arr[3] is changed to 1
Since arr[2] is 0, arr[0] is changed to 2
Since arr[3] is 2, arr[2] is changed to 3

Example 2:

Input: arr[] = {2, 0, 1, 4, 5, 3};
Output: arr[] = {1, 2, 0, 5, 3, 4};

Example 3:

Input: arr[] = {0, 1, 2, 3};
Output: arr[] = {0, 1, 2, 3};

Example 4:

Input: arr[] = {3, 2, 1, 0};
Output: arr[] = {3, 2, 1, 0};

A Simple Solution is to create a temporary array and one by one copy i to temp[arr[i]] where i varies from 0 to n-1.

Below is C implementation of the above idea.

```
// A simple C program to rearrange contents of arr[]
// such that arr[j] becomes j if arr[i] is j
#include<stdio.h>

// A simple method to rearrange 'arr[0..n-1]' so that 'arr[j]'
// becomes 'i' if 'arr[i]' is 'j'
void rearrangeNaive(int arr[], int n)
{
    // Create an auxiliary array of same size
    int temp[n], i;

    // Store result in temp[]
    for (i=0; i<n; i++)
        temp[arr[i]] = i;

    // Copy temp back to arr[]
    for (i=0; i<n; i++)
        arr[i] = temp[i];
}

// A utility function to print contents of arr[0..n-1]
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Drive program
int main()
{
    int arr[] = {1, 3, 0, 2};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, n);

    rearrangeNaive(arr, n);

    printf("Modified array is \n");
    printArray(arr, n);
    return 0;
}
```

Output:

Given array is

```

1 3 0 2
Modified array is
2 0 3 1

```

Time complexity of the above solution is O(n) and auxiliary space needed is O(n).

Can we solve this in O(n) time and O(1) auxiliary space?

The idea is based on the fact that the modified array is basically a permutation of input array. We can find the target permutation by storing the next item before updating it.

Let us consider array {1, 3, 0, 2} for example. We start with i = 0, arr[i] is 1. So we go to arr[1] and change it to 0 (because i is 0). Before we make the change, we store old value of arr[1] as the old value is going to be our new index i. In next iteration, we have i = 3, arr[3] is 2, so we change arr[2] to 3. Before making the change we store next i as old value of arr[2].

The below code gives idea about this approach.

```

// This function works only when output is a permutation
// with one cycle.
void rearrangeUtil(int arr[], int n)
{
    // 'val' is the value to be stored at 'arr[i]'
    int val = 0;    // The next value is determined
                    // using current index
    int i = arr[0]; // The next index is determined
                    // using current value

    // While all elements in cycle are not processed
    while (i != 0)
    {
        // Store value at index as it is going to be
        // used as next index
        int new_i = arr[i];

        // Update arr[]
        arr[i] = val;

        // Update value and index for next iteration
        val = i;
        i = new_i;
    }

    arr[0] = val; // Update the value at arr[0]
}

```

The above function doesn't work for inputs like {2, 0, 1, 4, 5, 3}; as there are two cycles. One cycle is (2, 0, 1) and other cycle is (4, 5, 3). How to handle multiple cycles with the O(1) space constraint?

The idea is to process all cycles one by one. To check whether an element is processed or not, we change the value of processed items arr[i] as -arr[i]. Since 0 can not be made negative, we first change all arr[i] to arr[i] + 1. In the end, we make all values positive and subtract 1 to get old values back.

```

// A space efficient C program to rearrange contents of
// arr[] such that arr[j] becomes j if arr[i] is j
#include<stdio.h>

// A utility function to rearrange elements in the cycle
// starting at arr[i]. This function assumes values in
// arr[] be from 1 to n. It changes arr[j-1] to i+1
// if arr[i-1] is j+1
void rearrangeUtil(int arr[], int n, int i)
{
    // 'val' is the value to be stored at 'arr[i]'
    int val = -(i+1); // The next value is determined
                      // using current index
    i = arr[i] - 1; // The next index is determined
                    // using current value

    // While all elements in cycle are not processed
    while (arr[i] > 0)
    {
        // Store value at index as it is going to be
        // used as next index
        int new_i = arr[i] - 1;

        // Update arr[]
        arr[i] = val;

        // Update value and index for next iteration
        val = -(i + 1);
    }
}

```

```

        i = new_i;
    }

// A space efficient method to rearrange 'arr[0..n-1]'
// so that 'arr[j]' becomes 'i' if 'arr[i]' is 'j'
void rearrange(int arr[], int n)
{
    // Increment all values by 1, so that all elements
    // can be made negative to mark them as visited
    int i;
    for (i=0; i<n; i++)
        arr[i]++;
}

// Process all cycles
for (i=0; i<n; i++)
{
    // Process cycle starting at arr[i] if this cycle is
    // not already processed
    if (arr[i] > 0)
        rearrangeUtil(arr, n, i);
}

// Change sign and values of arr[] to get the original
// values back, i.e., values in range from 0 to n-1
for (i=0; i<n; i++)
    arr[i] = (-arr[i]) - 1;
}

// A utility function to print contents of arr[0..n-1]
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Drive program
int main()
{
    int arr[] = {2, 0, 1, 4, 5, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, n);

    rearrange(arr, n);

    printf("Modified array is \n");
    printArray(arr, n);
    return 0;
}

```

Output:

```

Given array is
2 0 1 4 5 3
Modified array is
1 2 0 5 3 4

```

The time complexity of this method seems to be more than $O(n)$ at first look. If we take a closer look, we can notice that no element is processed more than constant number of times.

Find position of an element in a sorted array of infinite numbers

Suppose you have a sorted array of infinite numbers, how would you search an element in the array?

Source: Amazon Interview Experience.

Since array is sorted, the first thing clicks into mind is binary search, but the problem here is that we dont know size of array. If the array is infinite, that means we dont have proper bounds to apply binary search. So in order to find position of key, first we find bounds and then apply binary search algorithm.

Let low be pointing to 1st element and high pointing to 2nd element of array. Now compare key with high index element,

->if it is greater than high index element then copy high index in low index and double the high index.

->if it is smaller, then apply binary search on high and low indices found.

Below are implementations of above algorithm

C++

```
// C++ program to demonstrate working of an algorithm that finds
// an element in an array of infinite size
#include<iostream>
using namespace std;

// Simple binary search algorithm
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);
        return binarySearch(arr, mid+1, r, x);
    }
    return -1;
}

// function takes an infinite size array and a key to be
// searched and returns its position if found else -1.
// We don't know size of arr[] and we can assume size to be
// infinite in this function.
// NOTE THAT THIS FUNCTION ASSUMES arr[] TO BE OF INFINITE SIZE
// THEREFORE, THERE IS NO INDEX OUT OF BOUND CHECKING
int findPos(int arr[], int key)
{
    int l = 0, h = 1;
    int val = arr[0];

    // Find h to do binary search
    while (val < key)
    {
        l = h;          // store previous high
        h = 2*h;        // double high index
        val = arr[h]; // update new val
    }

    // at this point we have updated low and high indices,
    // thus use binary search between them
    return binarySearch(arr, l, h, key);
}

// Driver program
int main()
{
    int arr[] = {3, 5, 7, 9, 10, 90, 100, 130, 140, 160, 170};
    int ans = findPos(arr, 10);
    if (ans == -1)
        cout << "Element not found";
    else
        cout << "Element found at index " << ans;
    return 0;
}
```

```

// Simple binary search algorithm
int binarySearch(int arr[], int l, int r, int x)
{
    if (r>=l)
    {
        int mid = l + (r - 1)/2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);
        return binarySearch(arr, mid+1, r, x);
    }
    return -1;
}

// function takes an infinite size array and a key to be
// searched and returns its position if found else -1.
// We don't know size of arr[] and we can assume size to be
// infinite in this function.
// NOTE THAT THIS FUNCTION ASSUMES arr[] TO BE OF INFINITE SIZE
int findPos(int arr[], int key)
{
    int l = 0, h = 1;
    int val = arr[0];

    // Find h to do binary search
    while (val < key)
    {
        l = h;           // store previous high
        h = 2*h;         // double high index
        val = arr[h];   // update new val
    }

    // at this point we have updated low and high indices,
    // thus use binary search between them
    return binarySearch(arr, l, h, key);
}

// Driver program
int main()
{
    int arr[] = {3, 5, 7, 9, 10, 90, 100, 130, 140, 160, 170};
    int ans = findPos(arr, 10);
    if (ans== -1)
        cout << "Element not found";
    else
        cout << "Element found at index " << ans;
    return 0;
}

```

Python

```

# Python Program to demonstrate working of an algorithm that finds
# an element in an array of infinite size

# Binary search algorithm implementation
def binary_search(arr,l,r,x):

    if r >= l:
        mid = l+(r-l)/2

        if arr[mid] == x:
            return mid

        if arr[mid] > x:
            return binary_search(arr,l,mid-1,x)

        return binary_search(arr,mid+1,r,x)

    return -1

# function takes an infinite size array and a key to be
# searched and returns its position if found else -1.
# We don't know size of a[] and we can assume size to be
# infinite in this function.
# NOTE THAT THIS FUNCTION ASSUMES a[] TO BE OF INFINITE SIZE
# THEREFORE, THERE IS NO INDEX OUT OF BOUND CHECKING
def findPos(a, key):

    l, h, val = 0, 1, arr[0]

```

```

# Find h to do binary search
while val < key:
    l = h                  #store previous high
    h = 2*h                #double high index
    val = arr[h]            #update new val

# at this point we have updated low and high indices,
# thus use binary search between them
return binary_search(a, l, h, key)

# Driver function
arr = [3, 5, 7, 9, 10, 90, 100, 130, 140, 160, 170]
ans = findPos(arr,10)
if ans == -1:
    print "Element not found"
else:
    print"Element found at index",ans

# This code is contributed by __Devesh Agrawal__

```

Output:

Element found at index 4

Let p be the position of element to be searched. Number of steps for finding high index h is $O(\log p)$. The value of h must be less than 2^*p . The number of elements between $h/2$ and h must be $O(p)$. Therefore, time complexity of Binary Search step is also $O(\log p)$ and overall time complexity is $2^*O(\log p)$ which is $O(\log p)$.

Can QuickSort be implemented in O(nLogn) worst case time complexity?

The worst case time complexity of a typical implementation of [QuickSort](#) is $O(n^2)$. The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

Although randomized QuickSort works well even when the array is sorted, there is still possibility that the randomly picked element is always an extreme. Can the worst case be reduced to $O(nLogn)$?

The answer is yes, we can achieve $O(nLogn)$ worst case. The idea is based on the fact that the [median element of an unsorted array can be found in linear time](#). So we find the median first, then partition the array around the median element.

Following is C++ implementation based on above idea. Most of the functions in below program are copied from [Kth Smallest/Largest Element in Unsorted Array | Set 3 \(Worst Case Linear Time\)](#)

```
/* A worst case O(nLogn) implementation of quicksort */
#include<cstring>
#include<iostream>
#include<algorithm>
#include<climits>
using namespace std;

// Following functions are taken from http://goo.gl/ih05BF
int partition(int arr[], int l, int r, int k);
int kthSmallest(int arr[], int l, int r, int k);

/* A O(nLogn) time complexity function for sorting arr[l..h] */
void quickSort(int arr[], int l, int h)
{
    if (l < h)
    {
        // Find size of current subarray
        int n = h-l+1;

        // Find median of arr[].
        int med = kthSmallest(arr, l, h, n/2);

        // Partition the array around median
        int p = partition(arr, l, h, med);

        // Recur for left and right of partition
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, h);
    }
}

// A simple function to find median of arr[]. This is called
// only for an array of size 5 in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n); // Sort the array
    return arr[n/2]; // Return middle element
}

// Returns k'th smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of size 5, calculate median
        // of every group and store it in median[] array.
        int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;
        for (i=0; i<n/5; i++)
            median[i] = findMedian(arr+l+i*5, 5);
        if (i*5 < n) //For last group with less than 5 elements
        {
            median[i] = findMedian(arr+l+i*5, n%5);
            i++;
        }

        // Find median of all medians using recursive call.
        // If median[] has only one element, then no need
        // of recursive call
        int medOfMed = (i == 1)? median[i-1]:
                           kthSmallest(median, 0, i-1, i/2);
    }
}
```

```

// Partition the array around a random element and
// get position of pivot element in sorted array
int pos = partition(arr, l, r, medOfMed);

// If position is same as k
if (pos-1 == k-1)
    return arr[pos];
if (pos-1 > k-1) // If position is more, recur for left
    return kthSmallest(arr, l, pos-1, k);

// Else recur for right subarray
return kthSmallest(arr, pos+1, r, k-pos+l-1);
}

// If k is more than number of elements in array
return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
            break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1000, 10, 7, 8, 9, 30, 900, 1, 5, 6, 20};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    cout << "Sorted array is\n";
    printArray(arr, n);
    return 0;
}

```

Output:

```

Sorted array is
1 5 6 7 8 9 10 20 30 900 1000

```

How is QuickSort implemented in practice is above approach used?

Although worst case time complexity of the above approach is $O(n\log n)$, it is never used in practical implementations. The hidden constants in this approach are high compared to normal Quicksort. Following are some techniques used in practical implementations of QuickSort.

- 1) Randomly picking up to make worst case less likely to occur (Randomized QuickSort)

- 2) Calling insertion sort for small sized arrays to reduce recursive calls.
- 3) QuickSort is [tail recursive](#), so tail call optimizations is done.

So the approach discussed above is more of a theoretical approach with $O(n \log n)$ worst case time complexity.

Check if a given array contains duplicate elements within k distance from each other

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than size of array. Write a function that returns true if array contains duplicates within k distance.

Examples:

Input: $k = 3$, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}
Output: false
All duplicates are more than k distance away.

Input: $k = 3$, arr[] = {1, 2, 3, 1, 4, 5}
Output: true
1 is repeated at distance 3.

Input: $k = 3$, arr[] = {1, 2, 3, 4, 5}
Output: false

Input: $k = 3$, arr[] = {1, 2, 3, 4, 4}
Output: true

A **Simple Solution** is to run two loops. The outer loop picks every element $\text{arr}[i]$ as a starting element, the inner loop compares all elements which are within k distance of $\text{arr}[i]$. The time complexity of this solution is $O(kn)$.

We can solve this problem in $O(n)$ time using Hashing. The idea is to one by add elements to hash. We also remove elements which are at more than k distance from current element. Following is detailed algorithm.

- 1) Create an empty hashtable.
- 2) Traverse all elements from left from right. Let the current element be $\text{arr}[i]$
 - a) If current element $\text{arr}[i]$ is present in hashtable, then return true.
 - b) Else add $\text{arr}[i]$ to hash and remove $\text{arr}[i-k]$ from hash if i is greater than or equal to k

```
/* Java program to Check if a given array contains duplicate
elements within k distance from each other */
import java.util.*;

class Main
{
    static boolean checkDuplicatesWithinK(int arr[], int k)
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the input array
        for (int i=0; i<arr.length; i++)
        {
            // If already present n hash, then we found
            // a duplicate within k distance
            if (set.contains(arr[i]))
                return true;

            // Add this item to hashset
            set.add(arr[i]);

            // Remove the k+1 distant item
            if (i >= k)
                set.remove(arr[i-k]);
        }
        return false;
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 5, 3, 4, 3, 5, 6};
        if (checkDuplicatesWithinK(arr, 3))
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

Output:

Yes

Find the element that appears once

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.

Examples:

```
Input: arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3}
Output: 2
```

We can use sorting to do it in O(nLogn) time. We can also use hashing, but the worst case time complexity of hashing may be more than O(n) and hashing requires extra space.

The idea is to use bitwise operators for a solution that is O(n) time and uses O(1) extra space. The solution is not easy like other XOR based solutions, because all elements appear odd number of times here. The idea is taken from [here](#).

Run a loop for all elements in array. At the end of every iteration, maintain following two values.

ones: The bits that have appeared 1st time or 4th time or 7th time .. etc.

twos: The bits that have appeared 2nd time or 5th time or 8th time .. etc.

Finally, we return the value of *ones*

How to maintain the values of ones and twos?

ones and *twos* are initialized as 0. For every new element in array, find out the common set bits in the new element and previous value of *ones*. These common set bits are actually the bits that should be added to *twos*. So do bitwise OR of the common set bits with *twos*. *twos* also gets some extra bits that appear third time. These extra bits are removed later.

Update *ones* by doing XOR of new element with previous value of *ones*. There may be some bits which appear 3rd time. These extra bits are also removed later.

Both *ones* and *twos* contain those extra bits which appear 3rd time. Remove these extra bits by finding out common set bits in *ones* and *twos*.

```
#include <stdio.h>

int getSingle(int arr[], int n)
{
    int ones = 0, twos = 0 ;

    int common_bit_mask;

    // Let us take the example of {3, 3, 2, 3} to understand this
    for( int i=0; i< n; i++ )
    {
        /* The expression "one & arr[i]" gives the bits that are
         * there in both 'ones' and new element from arr[]. We
         * add these bits to 'twos' using bitwise OR

        Value of 'twos' will be set as 0, 3, 3 and 1 after 1st,
        2nd, 3rd and 4th iterations respectively */
        twos = twos | (ones & arr[i]);

        /* XOR the new bits with previous 'ones' to get all bits
         * appearing odd number of times

        Value of 'ones' will be set as 3, 0, 2 and 3 after 1st,
        2nd, 3rd and 4th iterations respectively */
        ones = ones ^ arr[i];

        /* The common bits are those bits which appear third time
         * So these bits should not be there in both 'ones' and 'twos'.
         * common_bit_mask contains all these bits as 0, so that the bits can
         * be removed from 'ones' and 'twos'

        Value of 'common_bit_mask' will be set as 00, 00, 01 and 10
        after 1st, 2nd, 3rd and 4th iterations respectively */
        common_bit_mask = ~(ones & twos);

        /* Remove common bits (the bits that appear third time) from 'ones'
         * Value of 'ones' will be set as 3, 0, 0 and 2 after 1st,
         * 2nd, 3rd and 4th iterations respectively */
        ones &= common_bit_mask;
    }
}
```

```

/* Remove common bits (the bits that appear third time) from 'twos'
   Value of 'twos' will be set as 0, 3, 1 and 0 after 1st,
   2nd, 3rd and 4th iterations respectively */
twos &= common_bit_mask;

// uncomment this code to see intermediate values
//printf ("%d %d \n", ones, twos);
}

return ones;
}

int main()
{
    int arr[] = {3, 3, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ",
           getSingle(arr, n));
    return 0;
}

```

Output:

2

Time Complexity: O(n)

Auxiliary Space: O(1)

Following is another O(n) time complexity and O(1) extra space method suggested by *aj*. We can sum the bits in same positions for all the numbers and take modulo with 3. The bits for which sum is not multiple of 3, are the bits of number with single occurrence.

Let us consider the example array {5, 5, 5, 8}. The 101, 101, 101, 1000

Sum of first bits%3 = (1 + 1 + 1 + 0)%3 = 0;

Sum of second bits%3 = (0 + 0 + 0 + 0)%0 = 0;

Sum of third bits%3 = (1 + 1 + 1 + 0)%3 = 0;

Sum of fourth bits%3 = (1)%3 = 1;

Hence number which appears once is 1000

```

#include <stdio.h>
#define INT_SIZE 32

int getSingle(int arr[], int n)
{
    // Initialize result
    int result = 0;

    int x, sum;

    // Iterate through every bit
    for (int i = 0; i < INT_SIZE; i++)
    {
        // Find sum of set bits at ith position in all
        // array elements
        sum = 0;
        x = (1 << i);
        for (int j=0; j< n; j++ )
        {
            if (arr[j] & x)
                sum++;
        }

        // The bits with sum not multiple of 3, are the
        // bits of element with single occurrence.
        if (sum % 3)
            result |= x;
    }

    return result;
}

// Driver program to test above function
int main()
{
    int arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 2, 2, 3, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ",
           getSingle(arr, n));
    return 0;
}

```


Replace every array element by multiplication of previous and next

Given an array of integers, update every element with multiplication of previous and next elements with following exceptions.

- a) First element is replaced by multiplication of first and second.
- b) Last element is replaced by multiplication of last and second last.

```
Input: arr[] = {2, 3, 4, 5, 6}
Output: arr[] = {6, 8, 15, 24, 30}

// We get the above output using following
// arr[] = {2*3, 2*4, 3*5, 4*6, 5*6}
```

Source: [Top 25 Interview Questions](#)

A Simple Solution is to create an auxiliary array, copy contents of given array to auxiliary array. Finally traverse the auxiliary array and update given array using copied values. Time complexity of this solution is O(n), but it requires O(n) extra space.

An efficient solution can solve the problem in O(n) time and O(1) space. The idea is to keep track of previous element in loop. Below is C++ implementation of this idea.

C

```
// C++ program to update every array element with
// multiplication of previous and next numbers in array
#include<iostream>
using namespace std;

void modify(int arr[], int n)
{
    // Nothing to do when array size is 1
    if (n <= 1)
        return;

    // store current value of arr[0] and update it
    int prev = arr[0];
    arr[0] = arr[0] * arr[1];

    // Update rest of the array elements
    for (int i=1; i<n-1; i++)
    {
        // Store current value of next iteration
        int curr = arr[i];

        // Update current value using previous value
        arr[i] = prev * arr[i+1];

        // Update previous value
        prev = curr;
    }

    // Update last array element
    arr[n-1] = prev * arr[n-1];
}

// Driver program
int main()
{
    int arr[] = {2, 3, 4, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    modify(arr, n);
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Java

```
// Java program to update every array element with
// multiplication of previous and next numbers in array
import java.io.*;
import java.util.*;
import java.lang.Math;

class Multiply
{
    static void modify(int arr[], int n)
```

```

{
    // Nothing to do when array size is 1
    if (n <= 1)
        return;

    // store current value of arr[0] and update it
    int prev = arr[0];
    arr[0] = arr[0] * arr[1];

    // Update rest of the array elements
    for (int i=1; i<n-1; i++)
    {
        // Store current value of next interation
        int curr = arr[i];

        // Update current value using previos value
        arr[i] = prev * arr[i+1];

        // Update previous value
        prev = curr;
    }

    // Update last array element
    arr[n-1] = prev * arr[n-1];
}

// Driver program to test above function
public static void main(String[] args)
{
    int arr[] = {2, 3, 4, 5, 6};
    int n = arr.length;
    modify(arr, n);
    for (int i=0; i<n; i++)
        System.out.print(arr[i]+" ");
}
/* This code is contributed by Devesh Agrawal */

```

6 8 15 24 30

Check if any two intervals overlap among a given set of intervals

An interval is represented as a combination of start time and end time. Given a set of intervals, check if any two intervals overlap.

Input: arr[] = {{1,3}, {5,7}, {2,4}, {6,8}}
Output: true
The intervals {1,3} and {2,4} overlap

Input: arr[] = {{1,3}, {7,9}, {4,6}, {10,13}}
Output: false
No pair of intervals overlap.

Expected time complexity is O(nLogn) where n is number of intervals.

A **Simple Solution** is to consider every pair of intervals and check if the pair overlaps or not. The time complexity of this solution is $O(n^2)$

A better solution is to **Use Sorting**. Following is complete algorithm.

- 1) Sort all intervals in increasing order of start time. This step takes $O(nLogn)$ time.
- 2) In the sorted array, if start time of an interval is less than end of previous interval, then there is an overlap. This step takes $O(n)$ time.

So overall time complexity of the algorithm is $O(nLogn) + O(n)$ which is $O(nLogn)$.

Below is C++ implementation of above idea.

```
// A C++ program to check if any two intervals overlap
#include <iostream>
#include <algorithm>
using namespace std;

// An interval has start time and end time
struct Interval
{
    int start;
    int end;
};

// Compares two intervals according to their starting time.
// This is needed for sorting the intervals using library
// function std::sort(). See http://goo.gl/iGspV
bool compareInterval(Interval i1, Interval i2)
{ return (i1.start < i2.start)? true: false; }

// Function to check if any two intervals overlap
bool isOverlap(Interval arr[], int n)
{
    // Sort intervals in increasing order of start time
    sort(arr, arr+n-1, compareInterval);

    // In the sorted array, if start time of an interval
    // is less than end of previous interval, then there
    // is an overlap
    for (int i=1; i<n; i++)
        if (arr[i-1].end > arr[i].start)
            return true;

    // If we reach here, then no overlap
    return false;
}

// Driver program
int main()
{
    Interval arr1[] = {{1,3}, {7,9}, {4,6}, {10,13}};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    isOverlap(arr1, n1)? cout << "Yes\n" : cout << "No\n";

    Interval arr2[] = { {6,8},{1,3},{2,4},{4,7} };
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    isOverlap(arr2, n2)? cout << "Yes\n" : cout << "No\n";

    return 0;
}
```

Output:

No
Yes

Delete an element from array (Using two traversals and one traversal)

Given an array and a number x, write a function to delete x from the given array.

Input: arr[] = {3, 1, 2, 5, 90}, x = 2
Output: arr[] = {3, 2, 5, 90}

A simple solution is to first search x in array, then elements that are on right side of x to one position back. The following are C++ and Java implementation of this simple approach.

C/C++

```
// C++ program to remove a given element from an array
#include<iostream>
using namespace std;

// This function removes an element x from arr[] and
// returns new size after removal (size is reduced only
// when x is present in arr[])
int deleteElement(int arr[], int n, int x)
{
    // Search x in array
    int i;
    for (i=0; i<n; i++)
        if (arr[i] == x)
            break;

    // If x found in array
    if (i < n)
    {
        // reduce size of array and move all
        // elements on space ahead
        n = n - 1;
        for (int j=i; j<n; j++)
            arr[j] = arr[j+1];
    }

    return n;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {11, 15, 6, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 6;

    // Delete x from arr[]
    n = deleteElement(arr, n, x);

    cout << "Modified array is \n";
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Java

```
// Java program to remove a given element from an array
import java.io.*;

class Deletion {

    // This function removes an element x from arr[] and
    // returns new size after removal (size is reduced only
    // when x is present in arr[])
    static int deleteElement(int arr[], int n, int x)
    {
        // Search x in array
        int i;
        for (i=0; i<n; i++)
            if (arr[i] == x)
                break;

        // If x found in array
        if (i < n)
        {
```

```

        // reduce size of array and move all
        // elements on space ahead
        n = n - 1;
        for (int j=i; j<n; j++)
            arr[j] = arr[j+1];
    }

    return n;
}

// Driver program to test above function
public static void main(String[] args)
{
    int arr[] = {11, 15, 6, 8, 9, 10};
    int n = arr.length;
    int x = 6;

    // Delete x from arr[]
    n = deleteElement(arr, n, x);

    System.out.println("Modified array is");
    for (int i = 0; i < n; i++)
        System.out.print(arr[i]+" ");
}
}
/*This code is contributed by Devesh Agrawal*/

```

Modified array is
11 15 8 9 10

The above method requires two traversals of array, one for searching and other for moving elements.

Can we delete the element using one traversal?

This is possible under the assumption that the element is always present in array. The idea is to start from right most element and keep moving elements while searching for x. Below are C++ and Java implementations of this approach. Note that this approach may give unexpected result when x is not present in array.

C++

```

// C++ program to remove a given element from an array
#include<iostream>
using namespace std;

// This function removes an element x from arr[] and
// returns new size after removal.
// Returned size is n-1 when element is present.
// Otherwise 0 is returned to indicate failure.
int deleteElement(int arr[], int n, int x)
{
    // If x is last element, nothing to do
    if (arr[n-1] == x)
        return (n-1);

    // Start from rightmost element and keep moving
    // elements one position ahead.
    int prev = arr[n-1], i;
    for (i=n-2; i>=0 && arr[i]!=x; i--)
    {
        int curr = arr[i];
        arr[i] = prev;
        prev = curr;
    }

    // If element was not found
    if (i < 0)
        return 0;

    // Else move the next element in place of x
    arr[i] = prev;

    return (n-1);
}

/* Driver program to test above function */
int main()
{

```

```

int arr[] = {11, 15, 6, 8, 9, 10};
int n = sizeof(arr)/sizeof(arr[0]);
int x = 6;

// Delete x from arr[]
n = deleteElement(arr, n, x);

cout << "Modified array is \n";
for (int i=0; i<n; i++)
    cout << arr[i] << " ";

return 0;
}

```

Java

```

// Java program to remove a given element from an array
import java.io.*;

class Deletion
{
    // This function removes an element x from arr[] and
    // returns new size after removal.
    // Returned size is n-1 when element is present.
    // Otherwise 0 is returned to indicate failure.
    static int deleteElement(int arr[], int n, int x)
    {
        // If x is last element, nothing to do
        if (arr[n-1] == x)
            return (n-1);

        // Start from rightmost element and keep moving
        // elements one position ahead.
        int prev = arr[n-1], i;
        for (i=n-2; i>=0 && arr[i]!=x; i--)
        {
            int curr = arr[i];
            arr[i] = prev;
            prev = curr;
        }

        // If element was not found
        if (i < 0)
            return 0;

        // Else move the next element in place of x
        arr[i] = prev;

        return (n-1);
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {11, 15, 6, 8, 9, 10};
        int n = arr.length;
        int x = 6;

        // Delete x from arr[]
        n = deleteElement(arr, n, x);

        System.out.println("Modified array is");
        for (int i = 0; i < n; i++)
            System.out.print(arr[i]+ " ");

    }
}
/*This code is contributed by Devesh Agrawal*/

```

```

Modified array is
11 15 8 9 10

```

Deleting an element from an array takes O(n) time even if we are given index of the element to be deleted. The time complexity remains O(n) for sorted arrays as well.

In linked list, if we know the pointer to the previous node of the node to be deleted, we can do deletion in O(1) time.

Given a sorted array and a number x, find the pair in array whose sum is closest to x

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

Examples:

Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54
Output: 22 and 30

Input: arr[] = {1, 3, 4, 7, 10}, x = 15
Output: 4 and 10

A simple solution is to consider every pair and keep track of closest pair (absolute difference between pair sum and x is minimum). Finally print the closest pair. Time complexity of this solution is $O(n^2)$

An efficient solution can find the pair in $O(n)$ time. The idea is similar to method 2 of [this](#) post. Following is detailed algorithm.

- 1) Initialize a variable diff as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.
- 2) Initialize two index variables l and r in the given sorted array.
 - (a) Initialize first to the leftmost index: l = 0
 - (b) Initialize second the rightmost index: r = n-1
- 3) Loop while l < r.
 - (a) If $\text{abs}(\text{arr}[l] + \text{arr}[r] - \text{sum}) < \text{diff}$ then update diff and result
 - (b) Else if($\text{arr}[l] + \text{arr}[r] < \text{sum}$) then l++
 - (c) Else r--

Following is C++ implementation of above algorithm.

C++

```
// Simple C++ program to find the pair with sum closest to a given no.
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

// Prints the pair with sum closest to x
void printClosest(int arr[], int n, int x)
{
    int res_l, res_r; // To store indexes of result pair

    // Initialize left and right indexes and difference between
    // pair sum and x
    int l = 0, r = n-1, diff = INT_MAX;

    // While there are elements between l and r
    while (r > l)
    {
        // Check if this pair is closer than the closest pair so far
        if (abs(arr[l] + arr[r] - x) < diff)
        {
            res_l = l;
            res_r = r;
            diff = abs(arr[l] + arr[r] - x);
        }

        // If this pair has more sum, move to smaller values.
        if (arr[l] + arr[r] > x)
            r--;
        else // Move to larger values
            l++;
    }

    cout << " The closest pair is " << arr[res_l] << " and " << arr[res_r];
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
    int n = sizeof(arr)/sizeof(arr[0]);
    printClosest(arr, n, x);
    return 0;
}
```

Java

```
// Java program to find pair with sum closest to x
import java.io.*;
import java.util.*;
import java.lang.Math;

class CloseSum {

    // Prints the pair with sum closest to x
    static void printClosest(int arr[], int n, int x)
    {
        int res_l=0, res_r=0; // To store indexes of result pair

        // Initialize left and right indexes and difference between
        // pair sum and x
        int l = 0, r = n-1, diff = Integer.MAX_VALUE;

        // While there are elements between l and r
        while (r > l)
        {
            // Check if this pair is closer than the closest pair so far
            if (Math.abs(arr[l] + arr[r] - x) < diff)
            {
                res_l = l;
                res_r = r;
                diff = Math.abs(arr[l] + arr[r] - x);
            }

            // If this pair has more sum, move to smaller values.
            if (arr[l] + arr[r] > x)
                r--;
            else // Move to larger values
                l++;
        }

        System.out.println(" The closest pair is "+arr[res_l]+" and "+ arr[res_r]);
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
        int n = arr.length;
        printClosest(arr, n, x);
    }
}

/*This code is contributed by Devesh Agrawal*/
```

The closest pair is 22 and 30

Find the largest pair sum in an unsorted array

Given an unsorted of distinct integers, find the largest pair sum in it. For example, the largest pair sum in {12, 34, 10, 6, 40} is 74.

Difficulty Level: Rookie

Expected Time Complexity: O(n) [Only one traversal of array is allowed]

This problem mainly boils down to finding the largest and second largest element in array. We can find the largest and second largest in O(n) time by traversing array once.

- 1) Initialize both first and second largest
first = max(arr[0], arr[1])
second = min(arr[0], arr[1])
- 2) Loop through remaining elements (from 3rd to end)
 - a) If the current element is greater than first, then update first and second.
 - b) Else if the current element is greater than second then update second
- 3) Return (first + second)

Below is C++ implementation of above algorithm

```
// C++ program to find largest pair sum in a given array
#include<iostream>
using namespace std;

/* Function to return largest pair sum. Assumes that
there are at-least two elements in arr[] */
int findLargestSumPair(int arr[], int n)
{
    // Initialize first and second largest element
    int first, second;
    if (arr[0] > arr[1])
    {
        first = arr[0];
        second = arr[1];
    }
    else
    {
        first = arr[1];
        second = arr[0];
    }

    // Traverse remaining array and find first and second largest
    // elements in overall array
    for (int i = 2; i < n; i++)
    {
        /* If current element is greater than first then update both
           first and second */
        if (arr[i] > first)
        {
            second = first;
            first = arr[i];
        }

        /* If arr[i] is in between first and second then update second */
        else if (arr[i] > second && arr[i] != first)
            second = arr[i];
    }
    return (first + second);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {12, 34, 10, 6, 40};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Max Pair Sum is " << findLargestSumPair(arr, n);
    return 0;
}
```

Output:

Max Pair Sum is 74

Time complexity of above solution is O(n).

Online algorithm for checking palindrome in a stream

Given a stream of characters (characters are received one by one), write a function that prints Yes if a character makes the complete string palindrome, else prints No.

Examples:

```
Input: str[] = "abcba"
Output: a Yes    // "a" is palindrome
        b No     // "ab" is not palindrome
        c No     // "abc" is palindrome
        b No     // "abcb" is not palindrome
        a Yes    // "abcba" is palindrome

Input: str[] = "aabaacaabaa"
Output: a Yes   // "a" is palindrome
        a Yes   // "aa" is palindrome
        b No    // "aab" is not palindrome
        a No    // "aaba" is not palindrome
        a Yes   // "aabaa" is palindrome
        c No    // "aabaac" is not palindrome
        a No    // "aabaaca" is not palindrome
        a No    // "aabaacaa" is not palindrome
        b No    // "aabaacaab" is not palindrome
        a No    // "aabaacaaaba" is not palindrome
        a Yes   // "aabaacaaabaa" is palindrome
```

Let input string be $\text{str}[0..n-1]$. A **Simple Solution** is to do following for every character $\text{str}[i]$ in input string. Check if substring $\text{str}[0:i]$ is palindrome, then print yes, else print no.

A **Better Solution** is to use the idea of Rolling Hash used in [Rabin Karp algorithm](#). The idea is to keep track of reverse of first half and second half (we also use first half and reverse of second half) for every index. Below is complete algorithm.

- 1) The first character is always a palindrome, so print yes for first character.
- 2) Initialize reverse of first half as "a" and second half as "b". Let the hash value of first half reverse be 'firstr' and that of second half be 'second'.
- 3) Iterate through string starting from second character, do following for every character $\text{str}[i]$, i.e., i varies from 1 to $n-1$.
 - a) If 'firstr' and 'second' are same, then character by character check the substring ending with current character and print "Yes" if palindrome.
Note that if hash values match, then strings need not be same.
For example, hash values of "ab" and "ba" are same, but strings are different. That is why we check complete string after hash.
 - b) Update 'firstr' and 'second' for next iteration.
If ' i ' is even, then add next character to the beginning of 'firstr' and end of second half and update hash values.
If ' i ' is odd, then keep 'firstr' as it is, remove leading character from second and append a next character at end.

Let us see all steps for example string **abcba**

Initial values of firstr and second
 $\text{firstr} = \text{hash}(a)$, $\text{second} = \text{hash}(b)$

Start from second character, i.e.,
 $i=1$

- a) Compare firstr and second, they dont match, so print no.
- b) Calculate hash values for next iteration, i.e., $i=2$

Since i is odd, firstr is not changed and second becomes hash(c)

$i=2$

- a) Compare firstr and second, they dont match, so print no.
- b) Calculate hash values for next iteration, i.e., $i=3$

Since i is even, firstr becomes hash(ba) and second becomes hash(cb)

$i=3$

- a) Compare first and second, they dont match, so print no.

b) Calculate hash values for next iteration, i.e., $i = 4$

Since i is odd, firstr is not changed and second becomes $\text{hash}(\text{ba})$

$i = 4$

a) firstr and second match, compare the whole strings, they match, so print yes

b) We don't need to calculate next hash values as this is last index

The idea of using rolling hashes is, next hash value can be calculated from previous in $O(1)$ time by just doing some constant number of arithmetic operations.

Below is C implementation of above approach.

```
// C program for online algorithm for palindrome checking
#include<stdio.h>
#include<string.h>

// d is the number of characters in input alphabet
#define d 256

// q is a prime number used for evaluating Rabin Karp's Rolling hash
#define q 103

void checkPalindromes(char str[])
{
    // Length of input string
    int N = strlen(str);

    // A single character is always a palindrome
    printf("%c Yes\n", str[0]);

    // Return if string has only one character
    if (N == 1) return;

    // Initialize first half reverse and second half for
    // as  $\text{firstr}$  and  $\text{second}$  characters
    int firstr = str[0] % q;
    int second = str[1] % q;

    int h = 1, i, j;

    // Now check for palindromes from second character
    // onward
    for (i=1; i<N; i++)
    {
        // If the hash values of ' $\text{firstr}$ ' and ' $\text{second}$ '
        // match, then only check individual characters
        if (firstr == second)
        {
            /* Check if  $\text{str}[0..i]$  is palindrome using
               simple character by character match */
            for (j = 0; j < i/2; j++)
            {
                if (str[j] != str[i-j])
                    break;
            }
            (j == i/2)? printf("%c Yes\n", str[i]):
            printf("%c No\n", str[i]);
        }
        else printf("%c No\n", str[i]);

        // Calculate hash values for next iteration.
        // Don't calculate hash for next characters if
        // this is the last character of string
        if (i != N-1)
        {
            // If  $i$  is even (next  $i$  is odd)
            if (i%2 == 0)
            {
                // Add next character after first half at beginning
                // of ' $\text{firstr}$ '
                h = (h*d) % q;
                firstr = (firstr + h*str[i/2])%q;

                // Add next character after second half at the end
                // of second half.
                second = (second*d + str[i+1])%q;
            }
            else
            {

```

```

        // If next i is odd (next i is even) then we
        // need not to change firstr, we need to remove
        // first character of second and append a
        // character to it.
        second = (d*(second + q - str[(i+1)/2]*h)%q
                  + str[i+1])%q;
    }
}
}

/* Driver program to test above function */
int main()
{
    char *txt = "aabaacaabaa";
    checkPalindromes(txt);
    getchar();
    return 0;
}

```

Output:

```

a Yes
a Yes
b No
a No
a Yes
c No
a No
a No
b No
a No
a Yes

```

The worst case time complexity of the above solution remains $O(n^2)$, but in general, it works much better than simple approach as we avoid complete substring comparison most of the time by first comparing hash values. The worst case occurs for input strings with all same characters like aaaaaa.

Find Union and Intersection of two unsorted arrays

Given two unsorted arrays that represent two sets (elements in every array are distinct), find union and intersection of two arrays.

For example, if the input arrays are:

```
arr1[] = {7, 1, 5, 2, 3, 6}  
arr2[] = {3, 8, 6, 20, 7}
```

Then your program should print Union as {1, 2, 3, 5, 6, 7, 8, 20} and Intersection as {3, 6}. Note that the elements of union and intersection can be printed in any order.

Method 1 (Naive)

Union:

- 1) Initialize union U as empty.
- 2) Copy all elements of first array to U.
- 3) Do following for every element x of second array:
.a) If x is not present in first array, then copy x to U.
- 4) Return U.

Intersection:

- 1) Initialize intersection I as empty.
- 2) Do following for every element x of first array
.a) If x is present in second array, then copy x to I.
- 4) Return I.

Time complexity of this method is $O(mn)$ for both operations. Here m and n are number of elements in arr1[] and arr2[] respectively.

Method 2 (Use Sorting)

- 1) Sort arr1[] and arr2[]. This step takes $O(mLogm + nLogn)$ time.
- 2) Use [O\(m+n\) algorithms to find union and intersection of two sorted arrays](#).

Overall time complexity of this method is $O(mLogm + nLogn)$.

Method 3 (Use Sorting and Searching)

Union:

- 1) Initialize union U as empty.
- 2) Find smaller of m and n and sort the smaller array.
- 3) Copy the smaller array to U.
- 4) For every element x of larger array, do following
.b) Binary Search x in smaller array. If x is not present, then copy it to U.
- 5) Return U.

Intersection:

- 1) Initialize intersection I as empty.
- 2) Find smaller of m and n and sort the smaller array.
- 3) For every element x of larger array, do following
.b) Binary Search x in smaller array. If x is present, then copy it to I.
- 4) Return I.

Time complexity of this method is $\min(mLogm + nLogn, mLogn + nLogm)$ which can also be written as $O((m+n)Logm, (m+n)Logn)$. This approach works much better than the previous approach when difference between sizes of two arrays is significant.

Thanks to [use_the_force](#) for suggesting this method in a comment [here](#).

Below is C++ implementation of this method.

```
// A C++ program to print union and intersection of two unsorted arrays  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int binarySearch(int arr[], int l, int r, int x);  
  
// Prints union of arr1[0..m-1] and arr2[0..n-1]  
void printUnion(int arr1[], int arr2[], int m, int n)  
{  
    // Before finding union, make sure arr1[0..m-1] is smaller  
    if (m > n)  
    {  
        int *temp = arr1;  
        arr1 = arr2;  
        arr2 = temp;  
    }  
    sort(arr1, arr1 + m);  
    sort(arr2, arr2 + n);  
    int i = 0, j = 0, k = 0;  
    while (i < m && j < n)  
    {  
        if (arr1[i] < arr2[j])  
            cout << arr1[i] << " ";  
        else if (arr2[j] < arr1[i])  
            cout << arr2[j] << " ";  
        else  
            cout << arr2[j] << " ";  
        i++;  
        j++;  
    }  
    while (i < m)  
        cout << arr1[i] << " ";  
    while (j < n)  
        cout << arr2[j] << " ";  
}
```

```

arr2 = temp;

    int temp = m;
    m = n;
    n = temp;
}

// Now arr1[] is smaller

// Sort the first array and print its elements (these two
// steps can be swapped as order in output is not important)
sort(arr1, arr1 + m);
for (int i=0; i<m; i++)
    cout << arr1[i] << " ";

// Search every element of bigger array in smaller array
// and print the element if not found
for (int i=0; i<n; i++)
    if (binarySearch(arr1, 0, m-1, arr2[i]) == -1)
        cout << arr2[i] << " ";
}

// Prints intersection of arr1[0..m-1] and arr2[0..n-1]
void printIntersection(int arr1[], int arr2[], int m, int n)
{
    // Before finding intersection, make sure arr1[0..m-1] is smaller
    if (m > n)
    {
        int *temp = arr1;
        arr1 = arr2;
        arr2 = temp;

        int temp = m;
        m = n;
        n = temp;
    }

    // Now arr1[] is smaller

    // Sort smaller array arr1[0..m-1]
    sort(arr1, arr1 + m);

    // Search every element of bigger array in smaller array
    // and print the element if found
    for (int i=0; i<n; i++)
        if (binarySearch(arr1, 0, m-1, arr2[i]) != -1)
            cout << arr2[i] << " ";
}

// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x) return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {7, 1, 5, 2, 3, 6};
    int arr2[] = {3, 8, 6, 20, 7};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);
    cout << "Union of two arrays is \n";
    printUnion(arr1, arr2, m, n);
}

```

```
cout << "\nIntersection of two arrays is \n";
printIntersection(arr1, arr2, m, n);
return 0;
}
```

Output:

```
Union of two arrays is
3 6 7 8 20 1 5 2
Intersection of two arrays is
7 3 6
```

Method 4 (Use Hashing)

Union:

- 1) Initialize union U as empty.
- 1) Initialize an empty hash table.
- 2) Iterate through first array and put every element of first array in the hash table, and in U.
- 4) For every element x of second array, do following
.a) Search x in the hash table. If x is not present, then copy it to U.
- 5) Return U.

Intersection:

- 1) Initialize intersection I as empty.
- 2) In initialize an empty hash table.
- 3) Iterate through first array and put every element of first array in the hash table.
- 4) For every element x of second array, do following
.a) Search x in the hash table. If x is present, then copy it to I.
- 5) Return I.

Time complexity of this method is $\Theta(m+n)$ under the assumption that hash table search and insert operations take $\Theta(1)$ time.

Pythagorean Triplet in an array

Given an array of integers, write a function that returns true if there is a triplet (a, b, c) that satisfies $a^2 + b^2 = c^2$.

Example:

```
Input: arr[] = {3, 1, 4, 6, 5}
Output: True
There is a Pythagorean triplet (3, 4, 5).
```

```
Input: arr[] = {10, 4, 6, 12, 5}
Output: False
There is no Pythagorean triplet.
```

Method 1 (Naive)

A simple solution is to run three loops, three loops pick three array elements and check if current three elements form a Pythagorean Triplet.

Below is C++ implementation of simple solution.

C++

```
// A C++ program that returns true if there is a Pythagorean
// Triplet in a given array.
#include <iostream>
using namespace std;

// Returns true if there is Pythagorean triplet in ar[0..n-1]
bool isTriplet(int ar[], int n)
{
    for (int i=0; i<n; i++)
    {
        for (int j=i+1; j<n; j++)
        {
            for (int k=j+1; k<n; k++)
            {
                // Calculate square of array elements
                int x = ar[i]*ar[i], y = ar[j]*ar[j], z = ar[k]*ar[k];

                if (x == y + z || y == x + z || z == x + y)
                    return true;
            }
        }
    }

    // If we reach here, no triplet found
    return false;
}

/* Driver program to test above function */
int main()
{
    int ar[] = {3, 1, 4, 6, 5};
    int ar_size = sizeof(ar)/sizeof(ar[0]);
    isTriplet(ar, ar_size)? cout << "Yes": cout << "No";
    return 0;
}
```

Java

```
// A Java program that returns true if there is a Pythagorean
// Triplet in a given array.
import java.io.*;

class PythagoreanTriplet {

    // Returns true if there is Pythagorean triplet in ar[0..n-1]
    static boolean isTriplet(int ar[], int n)
    {
        for (int i=0; i<n; i++)
        {
            for (int j=i+1; j<n; j++)
            {
                for (int k=j+1; k<n; k++)
                {
                    // Calculate square of array elements
                    int x = ar[i]*ar[i], y = ar[j]*ar[j], z = ar[k]*ar[k];
```

```

        if (x == y + z || y == x + z || z == x + y)
            return true;
    }
}

// If we reach here, no triplet found
return false;
}

// Driver program to test above function
public static void main(String[] args)
{
    int ar[] = {3, 1, 4, 6, 5};
    int ar_size = ar.length;
    if(isTriplet(ar,ar_size)==true)
        System.out.println("Yes");
    else
        System.out.println("No");
}
/* This code is contributed by Devesh Agrawal */

```

Yes

Time Complexity of the above solution is $O(n^3)$.

Method 2 (Use Sorting)

We can solve this in $O(n^2)$ time by sorting the array first.

- 1) Do square of every element in input array. This step takes $O(n)$ time.
- 2) Sort the squared array in increasing order. This step takes $O(n \log n)$ time.
- 3) To find a triplet (a, b, c) such that $a = b + c$, do following
 - a. Fix a as last element of sorted array.
 - b. Now search for pair (b, c) in subarray between first element and a. A pair (b, c) with given sum can be found in $O(n)$ time using meet in middle algorithm discussed in method 1 of [this](#) post.
 - c. If no pair found for current a, then move a one position back and repeat step 3.b.

Below is C++ implementation of above algorithm

C++

```

// A C++ program that returns true if there is a Pythagorean
// Triplet in a given array.
#include <iostream>
#include <algorithm>
using namespace std;

// Returns true if there is a triplet with following property
// A[i]*A[i] = A[j]*A[j] + A[k]*[k]
// Note that this function modifies given array
bool isTriplet(int arr[], int n)
{
    // Square array elements
    for (int i=0; i<n; i++)
        arr[i] = arr[i]*arr[i];

    // Sort array elements
    sort(arr, arr + n);

    // Now fix one element one by one and find the other two
    // elements
    for (int i = n-1; i >= 2; i--)
    {
        // To find the other two elements, start two index
        // variables from two corners of the array and move
        // them toward each other
        int l = 0; // index of the first element in arr[0..i-1]
        int r = i-1; // index of the last element in arr[0..i-1]
        while (l < r)
        {

```

```

        // A triplet found
        if (arr[l] + arr[r] == arr[i])
            return true;

        // Else either move 'l' or 'r'
        (arr[l] + arr[r] < arr[i])? l++: r--;
    }

// If we reach here, then no triplet found
return false;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {3, 1, 4, 6, 5};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    isTriplet(arr, arr_size)? cout << "Yes": cout << "No";
    return 0;
}

```

Java

```

// A Java program that returns true if there is a Pythagorean
// Triplet in a given array.
import java.io.*;
import java.util.*;

class PythagoreanTriplet
{
    // Returns true if there is a triplet with following property
    // A[i]*A[i] = A[j]*A[j] + A[k]*[k]
    // Note that this function modifies given array
    static boolean isTriplet(int arr[], int n)
    {
        // Square array elements
        for (int i=0; i<n; i++)
            arr[i] = arr[i]*arr[i];

        // Sort array elements
        Arrays.sort(arr);

        // Now fix one element one by one and find the other two
        // elements
        for (int i = n-1; i >= 2; i--)
        {
            // To find the other two elements, start two index
            // variables from two corners of the array and move
            // them toward each other
            int l = 0; // index of the first element in arr[0..i-1]
            int r = i-1; // index of the last element in arr[0..i-1]
            while (l < r)
            {
                // A triplet found
                if (arr[l] + arr[r] == arr[i])
                    return true;

                // Else either move 'l' or 'r'
                if (arr[l] + arr[r] < arr[i])
                    l++;
                else
                    r--;
            }
        }

        // If we reach here, then no triplet found
        return false;
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {3, 1, 4, 6, 5};
        int arr_size = arr.length;
        if (isTriplet(arr,arr_size)==true)
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}

```

```
    }  
}  
/*This code is contributed by Devesh Agrawal*/
```

Output:

Yes

Time complexity of this method is $O(n^2)$.

Maximum profit by buying and selling a share at most twice

In a daily share trading, a buyer buys shares in the morning and sells it on same day. If the trader is allowed to make at most 2 transactions in a day, where as second transaction can only start after first one is complete (Sell->buy->sell->buy). Given stock prices throughout day, find out maximum profit that a share trader could have made.

Examples:

```
Input: price[] = {10, 22, 5, 75, 65, 80}
Output: 87
Trader earns 87 as sum of 12 and 75
Buy at price 10, sell at 22, buy at 5 and sell at 80

Input: price[] = {2, 30, 15, 10, 8, 25, 80}
Output: 100
Trader earns 100 as sum of 28 and 72
Buy at price 2, sell at 30, buy at 8 and sell at 80

Input: price[] = {100, 30, 15, 10, 8, 25, 80};
Output: 72
Buy at price 8 and sell at 80.

Input: price[] = {90, 80, 70, 60, 50}
Output: 0
Not possible to earn.
```

A **Simple Solution** is to consider every index i and do following

```
Max profit with at most two transactions =
MAX {max profit with one transaction and subarray price[0..i] +
      max profit with one transaction and subarray price[i+1..n-1] }
i varies from 0 to n-1.
```

Maximum possible using one transaction can be calculated using following O(n) algorithm

[Maximum difference between two elements such that larger element appears after the smaller number](#)

Time complexity of above simple solution is $O(n^2)$.

We can do this O(n) using following **Efficient Solution**. The idea is to store maximum possible profit of every subarray and solve the problem in following two phases.

- 1) Create a table profit[0..n-1] and initialize all values in it 0.
- 2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]
- 3) Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0..i].
- 4) Return profit[n-1]

To do step 1, we need to keep track of maximum price from right to left side and to do step 2, we need to keep track of minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in second step, we use same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After an iteration i , the array profit[0..i] contains maximum profit with 2 transactions and profit[i+1..n-1] contains profit with two transactions.

Below are implementations of above idea.

C++

```
// C++ program to find maximum possible profit with at most
// two transactions
#include<iostream>
using namespace std;

// Returns maximum profit with two transactions on a given
// list of stock prices, price[0..n-1]
int maxProfit(int price[], int n)
{
    // Create profit array and initialize it as 0
    int *profit = new int[n];
    for (int i=0; i<n; i++)
        profit[i] = 0;
```

```

/* Get the maximum profit with only one transaction
   allowed. After this loop, profit[i] contains maximum
   profit from price[i..n-1] using at most one trans. */
int max_price = price[n-1];
for (int i=n-2;i>=0;i--)
{
    // max_price has maximum of price[i..n-1]
    if (price[i] > max_price)
        max_price = price[i];

    // we can get profit[i] by taking maximum of:
    // a) previous maximum, i.e., profit[i+1]
    // b) profit by buying at price[i] and selling at
    //     max_price
    profit[i] = max(profit[i+1], max_price-price[i]);
}

/* Get the maximum profit with two transactions allowed
   After this loop, profit[n-1] contains the result */
int min_price = price[0];
for (int i=1; i<n; i++)
{
    // min_price is minimum price in price[0..i]
    if (price[i] < min_price)
        min_price = price[i];

    // Maximum profit is maximum of:
    // a) previous maximum, i.e., profit[i-1]
    // b) (Buy, Sell) at (min_price, price[i]) and add
    //     profit of other trans. stored in profit[i]
    profit[i] = max(profit[i-1], profit[i] +
                    (price[i]-min_price) );
}
int result = profit[n-1];

delete [] profit; // To avoid memory leak

return result;
}

// Drive program
int main()
{
    int price[] = {2, 30, 15, 10, 8, 25, 80};
    int n = sizeof(price)/sizeof(price[0]);
    cout << "Maximum Profit = " << maxProfit(price, n);
    return 0;
}

```

Python

```

# Returns maximum profit with two transactions on a given
# list of stock prices price[0..n-1]
def maxProfit(price,n):

    # Create profit array and initialize it as 0
    profit = [0]*n

    # Get the maximum profit with only one transaction
    # allowed. After this loop, profit[i] contains maximum
    # profit from price[i..n-1] using at most one trans.
    max_price=price[n-1]

    for i in range( n-2, 0 ,-1):

        if price[i]> max_price:
            max_price = price[i]

        # we can get profit[i] by taking maximum of:
        # a) previous maximum, i.e., profit[i+1]
        # b) profit by buying at price[i] and selling at
        #     max_price
        profit[i] = max(profit[i+1], max_price - price[i])

    # Get the maximum profit with two transactions allowed
    # After this loop, profit[n-1] contains the result
    min_price=price[0]

    for i in range(1,n):

```

```

if price[i] < min_price:
    min_price = price[i]

# Maximum profit is maximum of:
# a) previous maximum, i.e., profit[i-1]
# b) (Buy, Sell) at (min_price, A[i]) and add
#     profit of other trans. stored in profit[i]
profit[i] = max(profit[i-1], profit[i]+(price[i]-min_price))

result = profit[n-1]

return result

# Driver function
price = [2, 30, 15, 10, 8, 25, 80]
print "Maximum profit is", maxProfit(price, len(price))

# This code is contributed by __Devesh Agrawal__

```

Maximum Profit = 100

Time complexity of the above solution is O(n).

Algorithmic Paradigm: Dynamic Programming

Linked List | Set 1 (Introduction)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

- 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- 2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, in a system if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.

Representation in C:

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

Each node in a list consists of at least two parts:

- 1) data
- 2) pointer to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

In Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

C

```
// A linked list node
struct node
{
    int data;
    struct node *next;
};
```

Java

```
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized as null
        Node(int d) {data = d;}
    }
}
```

Python

```
# Node class
class Node:
```

```

# Function to initialize the node object
def __init__(self, data):
    self.data = data # Assign data
    self.next = None # Initialize next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked List object
    def __init__(self):
        self.head = None

```

First Simple Linked List in C Let us create a simple linked list with 3 nodes.

C

```

// A simple C program to introduce a linked list
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *next;
};

// Program to create a simple linked list with 3 nodes
int main()
{
    struct node* head = NULL;
    struct node* second = NULL;
    struct node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct node*)malloc(sizeof(struct node));
    second = (struct node*)malloc(sizeof(struct node));
    third = (struct node*)malloc(sizeof(struct node));

    /* Three blocks have been allocated dynamically.
       We have pointers to these three blocks as first, second and third
       head           second          third
       |             |               |
       |             |               |
    +---+-----+     +---+-----+     +---+-----+
    | # | # |     | # | # |     | # | # |
    +---+-----+     +---+-----+     +---+-----+
    */

    # represents any random value.
    Data is random because we havent assigned anything yet */

    head->data = 1; //assign data in first node
    head->next = second; // Link first node with the second node

    /* data has been assigned to data part of first block (block
       pointed by head). And next pointer of first block points to
       second. So they both are linked.

       head           second          third
       |             |               |
       |             |               |
    +---+-----+     +---+-----+     +---+-----+
    | 1 | o----->| # | # |     | # | # |
    +---+-----+     +---+-----+     +---+-----+
    */

    second->data = 2; //assign data to second node
    second->next = third; // Link second node with the third node

    /* data has been assigned to data part of second block (block pointed by
       second). And next pointer of the second block points to third block.
       So all three blocks are linked.

       head           second          third
       |             |               |
       |             |               |
    +---+-----+     +---+-----+     +---+-----+
    | 1 | o----->| 2 | o----->| # | # |
    +---+-----+     +---+-----+     +---+-----+
    */

    third->data = 3; //assign data to third node

```

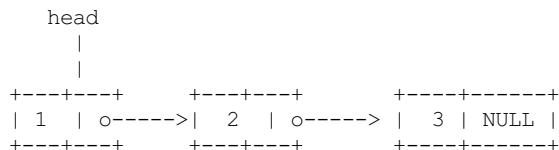
```

third->next = NULL;

/* data has been assigned to data part of third block (block pointed
 by third). And next pointer of the third block is made NULL to indicate
 that the linked list is terminated here.

```

We have the linked list ready.



Note that only head is sufficient to represent the whole list. We can traverse the complete list by following next pointers. */

```

return 0;
}

```

Java

```

// A simple Java program to introduce a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node. This inner class is made static so that
       main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next=null; } // Constructor
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();

        llist.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);

        /* Three nodes have been allocated dynamically.
           We have references to these three blocks as first,
           second and third

        llist.head      second      third
        |            |            |
        +---+---+   +---+---+   +---+---+
        | 1 | null |   | 2 | null |   | 3 | null |
        +---+---+   +---+---+   +---+---+ */

        llist.head.next = second; // Link first node with the second node

        /* Now next of first Node refers to second. So they
           both are linked.

        llist.head      second      third
        |            |            |
        |            |            |
        +---+---+   +---+---+   +---+---+
        | 1 | o----->| 2 | null |   | 3 | null |
        +---+---+   +---+---+   +---+---+ */

        second.next = third; // Link second node with the third node

        /* Now next of second Node refers to third. So all three
           nodes are linked.

        llist.head      second      third
        |            |            |
        |            |            |
        +---+---+   +---+---+   +---+---+
        | 1 | o----->| 2 | o----->| 3 | null |
        +---+---+   +---+---+   +---+---+

```

```

    +---+-----+    +---+-----+    +---+-----+ */
}
}
```

Python

```
# A simple Python program to introduce a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data    # Assign data
        self.next = None    # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

# Code execution starts here
if __name__ == '__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    """
    Three nodes have been created.
    We have references to these three blocks as first,
    second and third

    llist.head      second       third
    |             |           |
    |             |           |
    +---+-----+    +---+-----+    +---+-----+
    | 1 | None |    | 2 | None |    | 3 | None |
    +---+-----+    +---+-----+    +---+-----+
    """

    llist.head.next = second; # Link first node with second

    """
    Now next of first Node refers to second. So they
    both are linked.

    llist.head      second       third
    |             |           |
    |             |           |
    +---+-----+    +---+-----+    +---+-----+
    | 1 | o----->| 2 | null |    | 3 | null |
    +---+-----+    +---+-----+    +---+-----+
    """

    second.next = third; # Link second node with the third node

    """
    Now next of second Node refers to third. So all three
    nodes are linked.

    llist.head      second       third
    |             |           |
    |             |           |
    +---+-----+    +---+-----+    +---+-----+
    | 1 | o----->| 2 | o----->| 3 | null |
    +---+-----+    +---+-----+    +---+-----+
    """

```

Linked List Traversal

C

```
// A simple C program for traversal of a linked list
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *next;
};

// This function prints contents of linked list starting from
// the given node
void printList(struct node *n)
{
    while (n != NULL)
    {
        printf(" %d ", n->data);
        n = n->next;
    }
}

int main()
{
    struct node* head = NULL;
    struct node* second = NULL;
    struct node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct node*)malloc(sizeof(struct node));
    second = (struct node*)malloc(sizeof(struct node));
    third = (struct node*)malloc(sizeof(struct node));

    head->data = 1; //assign data in first node
    head->next = second; // Link first node with second

    second->data = 2; //assign data to second node
    second->next = third;

    third->data = 3; //assign data to third node
    third->next = NULL;

    printList(head);

    return 0;
}
```

Java

```
// A simple Java program for traversal of a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node. This inner class is made static so that
       main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next=null; } // Constructor
    }

    /* This function prints contents of linked list starting from head */
    public void printList()
    {
        Node n = head;
        while (n != null)
        {
            System.out.print(n.data+" ");
            n = n.next;
        }
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();
```

```

        llist.head      = new Node(1);
        Node second    = new Node(2);
        Node third     = new Node(3);

        llist.head.next = second; // Link first node with the second node
        second.next = third; // Link first node with the second node

        llist.printList();
    }
}

```

Python

```

# A simple Python program for traversal of a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data    # Assign data
        self.next = None    # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next

# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    llist.head.next = second; # Link first node with second
    second.next = third; # Link second node with the third node

    llist.printList()

```

1 2 3

You may like to try [Practice MCQ Questions on Linked List](#)

We will soon be publishing more posts on Linked Lists.

[GATE Corner](#)[Quiz Corner](#)

Linked List vs Array

Difficulty Level: Rookie

Both Arrays and [Linked List](#) can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Linked lists have following drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Arrays have better cache locality that can make a pretty big difference in performance.

Please also see [this](#) thread.

References:

<http://cslibrary.stanford.edu/103/LinkedListBasics.pdf>

Linked List | Set 2 (Inserting a node)

We have introduced Linked Lists in the [previous post](#). We also created a simple linked list with 3 nodes and discussed linked list traversal.

All programs discussed in this post consider following representations of linked list .

C

```
// A linked list node
struct node
{
    int data;
    struct node *next;
};
```

Java

```
// Linked List Class
class LinkedList
{
    Node head; // head of list

    /* Node Class */
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {data = d; next = null; }
    }
}
```

Python

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked List object
    def __init__(self):
        self.head = None
```

1)

2)

3)

Add a node at the front: (A 4 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See [this](#))

Following are the 4 steps to add node at the front.

C

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
```

```

/* 2. put in the data */
new_node->data = new_data;

/* 3. Make next of new node as head */
new_node->next = (*head_ref);

/* 4. move the head to point to the new node */
(*head_ref) = new_node;
}

```

Java

```

/* This function is in LinkedList class. Inserts a
   new Node at front of the list. This method is
   defined inside LinkedList class shown above */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

```

Python

```

# This function is in LinkedList class
# Function to insert a new node at the beginning
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    #          Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node

```

Add a node after a given node: (5 steps process)

We are given pointer to a node, and the new node is inserted after the given node.

C

```

/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}

```

Java

```

/* This function is in LinkedList class.
   Inserts a new node after the given prev_node. This method is
   defined inside LinkedList class shown above */
public void insertAfter(Node prev_node, int new_data)
{
    /* 1. Check if the given Node is null */
    if (prev_node == null)
    {
        System.out.println("The given previous node cannot be null");
        return;
    }

    /* 2. Allocate the Node &
       3. Put in the data*/
    Node new_node = new Node(new_data);

    /* 4. Make next of new Node as next of prev_node */
    new_node.next = prev_node.next;

    /* 5. make next of prev_node as new_node */
    prev_node.next = new_node;
}

```

Python

```

# This function is in LinkedList class.
# Inserts a new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print "The given previous node must inLinkedList."
        return

    # 2. Create new node &
    # 3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node

```

Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

Following are the 6 steps to add node at the end.

C

```

/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next
       of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }
}

```

```

/* 5. Else traverse till the last node */
while (last->next != NULL)
    last = last->next;

/* 6. Change the next of last node */
last->next = new_node;
return;
}

```

Java

```

/* Appends a new node at the end. This method is
   defined inside LinkedList class shown above */
public void append(int new_data)
{
    /* 1. Allocate the Node &
       2. Put in the data
       3. Set next as null */
    Node new_node = new Node(new_data);

    /* 4. If the Linked List is empty, then make the
       new node as head */
    if (head == null)
    {
        head = new Node(new_data);
        return;
    }

    /* 4. This new node is going to be the last node, so
       make next of it as null */
    new_node.next = null;

    /* 5. Else traverse till the last node */
    Node last = head;
    while (last.next != null)
        last = last.next;

    /* 6. Change the next of last node */
    last.next = new_node;
    return;
}

```

Python

```

# This function is defined in Linked List class
# Appends a new node at the end. This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    #     new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node

```

Following is a complete program that uses all of the above methods to create a linked list.

C

```
// A complete working C program to demonstrate all insertion methods
```

```

// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and
   an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next of
       it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
}

```

```

        return;
    }

// This function prints contents of linked list starting from head
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);

    printf("\n Created Linked list is: ");
    printList(head);

    return 0;
}

```

Java

```

// A complete working Java program to demonstrate all insertion methods
// on linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Inserts a new node after the given prev_node. */
    public void insertAfter(Node prev_node, int new_data)
    {
        /* 1. Check if the given Node is null */
        if (prev_node == null)
        {
            System.out.println("The given previous node cannot be null");
            return;
        }
    }
}

```

```

/* 2 & 3: Allocate the Node &
   Put in the data*/
Node new_node = new Node(new_data);

/* 4. Make next of new Node as next of prev_node */
new_node.next = prev_node.next;

/* 5. make next of prev_node as new_node */
prev_node.next = new_node;
}

/* Appends a new node at the end. This method is
   defined inside LinkedList class shown above */
public void append(int new_data)
{
    /* 1. Allocate the Node &
       2. Put in the data
       3. Set next as null */
    Node new_node = new Node(new_data);

    /* 4. If the Linked List is empty, then make the
       new node as head */
    if (head == null)
    {
        head = new Node(new_data);
        return;
    }

    /* 4. This new node is going to be the last node, so
       make next of it as null */
    new_node.next = null;

    /* 5. Else traverse till the last node */
    Node last = head;
    while (last.next != null)
        last = last.next;

    /* 6. Change the next of last node */
    last.next = new_node;
    return;
}

/* This function prints contents of linked list starting from
   the given node */
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Drier program to test above functions. Ideally this function
   should be in a separate user class. It is kept here to keep
   code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    // Insert 6. So linked list becomes 6->NULLlist
    llist.append(6);

    // Insert 7 at the beginning. So linked list becomes
    // 7->6->NULLlist
    llist.push(7);

    // Insert 1 at the beginning. So linked list becomes
    // 1->7->6->NULLlist
    llist.push(1);

    // Insert 4 at the end. So linked list becomes
    // 1->7->6->4->NULLlist
    llist.append(4);

    // Insert 8, after 7. So linked list becomes
    // 1->7->8->6->4->NULLlist
    llist.insertAfter(llist.head.next, 8);
}

```

```

        System.out.println("\nCreated Linked list is: ");
        llist.printList();
    }
}

// This code is contributed by Rajat Mishra

```

Python

```

# A complete working Python program to demonstrate all
# insertion methods of linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data    # Assign data
        self.next = None    # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        #          Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node

    # This function is in LinkedList class. Inserts a
    # new node after the given prev_node. This method is
    # defined inside LinkedList class shown above */
    def insertAfter(self, prev_node, new_data):

        # 1. check if the given prev_node exists
        if prev_node is None:
            print "The given previous node must inLinkedList."
            return

        # 2. create new node &
        #      Put in the data
        new_node = Node(new_data)

        # 4. Make next of new Node as next of prev_node
        new_node.next = prev_node.next

        # 5. make next of prev_node as new_node
        prev_node.next = new_node

    # This function is defined in Linked List class
    # Appends a new node at the end. This method is
    # defined inside LinkedList class shown above */
    def append(self, new_data):

        # 1. Create a new node
        # 2. Put in the data
        # 3. Set next as None
        new_node = Node(new_data)

        # 4. If the Linked List is empty, then make the
        #     new node as head
        if self.head is None:
            self.head = new_node
            return

        # 5. Else traverse till the last node

```

```

last = self.head
while (last.next):
    last = last.next

# 6. Change the next of last node
last.next = new_node

# Utility function to print the linked list
def printList(self):
    temp = self.head
    while (temp):
        print temp.data,
        temp = temp.next

# Code execution starts here
if __name__=='__main__':
    # Start with the empty list
    llist = LinkedList()

    # Insert 6. So linked list becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print 'Created linked list is:',
    llist.printList()

# This code is contributed by Manikantan Narasimhan

```

Created Linked list is: 1 7 8 6 4

You may like to try [Practice MCQ Questions on Linked List](#)

[GATE Corner](#)[Quiz Corner](#)

Linked List | Set 3 (Deleting a node)

We have discussed [Linked List Introduction](#) and [Linked List Insertion](#) in previous posts on singly linked list.

Let us formulate the problem statement to understand the deletion process. *Given a key, delete the first occurrence of this key in linked list.*
To delete a node from linked list, we need to do following steps.

- 1) Find previous node of the node to be deleted.
- 2) Change next of previous node.
- 3) Free memory for the node to be deleted.

Since every node of linked list is dynamically allocated using malloc() in C, we need to call [free\(\)](#) for freeing memory allocated for the node to be deleted.

C/C++

```
// A complete working C program to demonstrate deletion in singly
// linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
   and a key, deletes the first occurrence of key in linked list */
void deleteNode(struct node **head_ref, int key)
{
    // Store head node
    struct node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;      // Changed head
        free(temp);                  // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;

    free(temp); // Free memory
}

// This function prints contents of linked list starting from
// the given node
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```

```

}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);

    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 1);
    puts("\nLinked List after Deletion of 1: ");
    printList(head);
    return 0;
}

```

Java

```

// A complete working C program to demonstrate deletion in singly
// linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Given a key, deletes the first occurrence of key in linked list */
    void deleteNode(int key)
    {
        // Store head node
        Node temp = head, prev;

        // If head node itself holds the key to be deleted
        if (temp != null && temp.data == key)
        {
            head = temp.next; // Changed head
            return;
        }

        // Search for the key to be deleted, keep track of the
        // previous node as we need to change temp.next
        while (temp.next != null && temp.next.data != key)
            temp = temp.next;

        // If key was not present in linked list
        if (temp == null || temp.next == null) return;

        // Unlink the node from linked list
        temp.next = temp.next.next;
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        Node new_node = new Node(new_data);
        new_node.next = head;
        head = new_node;
    }

    /* This function prints contents of linked list starting from
     * the given node */
    public void printList()
    {
        Node tnode = head;
        while (tnode != null)

```

```

    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Drier program to test above functions. Ideally this function
should be in a separate user class. It is kept here to keep
code compact */
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    llist.push(7);
    llist.push(1);
    llist.push(3);
    llist.push(2);

    System.out.println("\nCreated Linked list is:");
    llist.printList();

    llist.deleteNode(1); // Delete node at position 4

    System.out.println("\nLinked List after Deletion at position 4:");
    llist.printList();
}
}

```

Created Linked List:
 2 3 1 7
 Linked List after Deletion of 1:
 2 3 7

[GATE Corner](#)[Quiz Corner](#)

Delete a Linked List node at a given position

Given a singly linked list and a position, delete a linked list node at the given position.

Example:

```
Input: position = 1, Linked List = 8->2->3->1->7
Output: Linked List = 8->3->1->7
```

```
Input: position = 0, Linked List = 8->2->3->1->7
Output: Linked List = 2->3->1->7
```

Below is C implementation of above idea.

C/C++

```
// A complete working C program to delete a node in a linked list
// at a given position
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
   and a position, deletes the node at the given position */
void deleteNode(struct node **head_ref, int position)
{
    // If linked list is empty
    if (*head_ref == NULL)
        return;

    // Store head node
    struct node* temp = *head_ref;

    // If head needs to be removed
    if (position == 0)
    {
        *head_ref = temp->next;    // Change head
        free(temp);                // free old head
        return;
    }

    // Find previous node of the node to be deleted
    for (int i=0; temp!=NULL && i<position-1; i++)
        temp = temp->next;

    // If position is more than number of nodes
    if (temp == NULL || temp->next == NULL)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    struct node *next = temp->next->next;

    // Unlink the node from linked list
    free(temp->next); // Free memory

    temp->next = next; // Unlink the deleted node from list
}

// This function prints contents of linked list starting from
// the given node
void printList(struct node *node)
{
```

```

while (node != NULL)
{
    printf(" %d ", node->data);
    node = node->next;
}
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);
    push(&head, 8);

    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 4);
    puts("\nLinked List after Deletion at position 4: ");
    printList(head);
    return 0;
}

```

Java

```

// A complete working Java program to delete a node in a linked list
// at a given position
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Given a reference (pointer to pointer) to the head of a list
       and a position, deletes the node at the given position */
    void deleteNode(int position)
    {
        // If linked list is empty
        if (head == null)
            return;

        // Store head node
        Node temp = head;

        // If head needs to be removed
        if (position == 0)
        {
            head = temp.next; // Change head
            return;
        }

        // Find previous node of the node to be deleted

```

```

for (int i=0; temp!=null && i<position-1; i++)
    temp = temp.next;

// If position is more than number of nodes
if (temp == null || temp.next == null)
    return;

// Node temp->next is the node to be deleted
// Store pointer to the next of node to be deleted
Node next = temp.next.next;

    temp.next = next; // Unlink the deleted node from list
}

/* This function prints contents of linked list starting from
   the given node */
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Driver program to test above functions. Ideally this function
   should be in a separate user class. It is kept here to keep
   code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    llist.push(7);
    llist.push(1);
    llist.push(3);
    llist.push(2);
    llist.push(8);

    System.out.println("\nCreated Linked list is: ");
    llist.printList();

    llist.deleteNode(4); // Delete node at position 4

    System.out.println("\nLinked List after Deletion at position 4: ");
    llist.printList();
}
}

```

Created Linked List:
8 2 3 1 7
Linked List after Deletion at position 4:
8 2 3 1

A Programmers approach of looking at Array vs. Linked List

In general, array is considered a data structure for which size is fixed at the compile time and array memory is allocated either from Data section (e.g. global array) or Stack section (e.g. local array).

Similarly, linked list is considered a data structure for which size is not fixed and memory is allocated from Heap section (e.g. using malloc() etc.) as and when needed. In this sense, array is taken as a static data structure (residing in Data or Stack section) while linked list is taken as a dynamic data structure (residing in Heap section). Memory representation of array and linked list can be visualized as follows:

An array of 4 elements (integer type) which have been initialized with 1, 2, 3 and 4. Suppose, these elements are allocated at memory addresses 0x100, 0x104, 0x108 and 0x10B respectively.

| | | | |
|---------|---------|---------|---------|
| [(1)] | [(2)] | [(3)] | [(4)] |
| 0x100 | 0x104 | 0x108 | 0x10B |

A linked list with 4 nodes where each node has integer as data and these data are initialized with 1, 2, 3 and 4. Suppose, these nodes are allocated via malloc() and memory allocated for them is 0x200, 0x308, 0x404 and 0x20B respectively.

| | | | |
|-----------------|-----------------|-----------------|----------------|
| [(1) , 0x308] | [(2) , 0x404] | [(3) , 0x20B] | [(4) , NULL] |
| 0x200 | 0x308 | 0x404 | 0x20B |

Anyone with even little understanding of array and linked-list might not be interested in the above explanation. I mean, it is well known that the array elements are allocated memory in sequence i.e. contiguous memory while nodes of a linked list are non-contiguous in memory. Though it sounds trivial yet this is the most important difference between array and linked list. It should be noted that due to this contiguous versus non-contiguous memory, array and linked list are different. In fact, this difference is what makes array vs. linked list! In the following sections, we will try to explore on this very idea further.

Since elements of array are contiguous in memory, we can access any element randomly using index e.g. intArr[3] will access directly fourth element of the array. (For newbies, array indexing starts from 0 and that's why fourth element is indexed with 3). Also, due to contiguous memory for successive elements in array, no extra information is needed to be stored in individual elements i.e. no overhead of metadata in arrays. Contrary to this, linked list nodes are non-contiguous in memory. It means that we need some mechanism to traverse or access linked list nodes. To achieve this, each node stores the location of next node and this forms the basis of the link from one node to next node. Therefore, it's called Linked list. Though storing the location of next node is overhead in linked list but it's required. Typically, we see linked list node declaration as follows:

```
struct llNode
{
    int dataInt;

    /* nextNode is the pointer to next node in linked list*/
    struct llNode * nextNode;
};
```

So array elements are contiguous in memory and therefore not requiring any metadata. And linked list nodes are non-contiguous in memory thereby requiring metadata in the form of location of next node. Apart from this difference, we can see that array could have several unused elements because memory has already been allocated. But linked list will have only the required no. of data items. All the above information about array and linked list has been mentioned in several textbooks though in different ways.

What if we need to allocate array memory from Heap section (i.e. at run time) and linked list memory from Data/Stack section. First of all, is it possible? Before that, one might ask why would someone need to do this? Now, I hope that the remaining article would make you rethink about the idea of array vs. linked-list 😊

Now consider the case when we need to store certain data in array (because array has the property of random access due to contiguous memory) but we don't know the total size apriori. One possibility is to allocate memory of this array from Heap at run time. For example, as follows:

```
/*At run-time, suppose we know the required size for integer array (e.g. input size from user). Say, the array size is stored in variable arrSize.
Allocate this array from Heap as follows*/
```

```
int * dynArr = (int *)malloc(sizeof(int)*arrSize);
```

Though the memory of this array is allocated from Heap, the elements can still be accessed via index mechanism e.g. dynArr[i]. Basically, based on the programming problem, we have combined one benefit of array (i.e. random access of elements) and one benefit of linked list (i.e. delaying the memory allocation till run time and allocating memory from Heap). Another advantage of having this type of dynamic array is that, this method of allocating array from Heap at run time could reduce code-size (of course, it depends on certain other factors e.g. program format etc.)

Now consider the case when we need to store data in a linked list (because no. of nodes in linked list would be equal to actual data items stored i.e. no extra space like array) but we aren't allowed to get this memory from Heap again and again for each node. This might look hypothetical situation to some folks but it's not very uncommon requirement in embedded systems. Basically, in several embedded programs, allocating memory via malloc() etc. isn't allowed due to multiple reasons. One obvious reason is performance i.e. allocating memory via malloc() is costly in terms of time complexity because your embedded program is required to be deterministic most of the times. Another reason could be module specific memory management i.e. it's possible that each module in embedded system manages its own memory. In short, if we need to perform our own memory management, instead of relying on system provided APIs of malloc() and free(), we might choose the linked list which is simulated using

array. I hope that you got some idea why we might need to simulate linked list using array. Now, let us first see how this can be done. Suppose, type of a node in linked list (i.e. underlying array) is declared as follows:

```
struct sllNode
{
    int dataInt;

    /*Here, note that nextIndex stores the location of next node in
     linked list*/
    int nextIndex;
};

struct sllNode arrayLL[5];
```

If we initialize this linked list (which is actually an array), it would look as follows in memory:

| | | | | |
|------------|------------|------------|------------|------------|
| [(0), -1] | [(0), -1] | [(0), -1] | [(0), -1] | [(0), -1] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

The important thing to notice is that all the nodes of the linked list are contiguous in memory (each one occupying 8 bytes) and nextIndex of each node is set to -1. This (i.e. -1) is done to denote that the each node of the linked list is empty as of now. This linked list is denoted by head index 0.

Now, if this linked list is updated with four elements of data part 4, 3, 2 and 1 successively, it would look as follows in memory. This linked list can be viewed as 0x500 -> 0x508 -> 0x510 -> 0x518.

| | | | | |
|-----------|-----------|-----------|------------|------------|
| [(1), 1] | [(2), 2] | [(3), 3] | [(4), -2] | [(0), -1] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

The important thing to notice is nextIndex of last node (i.e. fourth node) is set to -2. This (i.e. -2) is done to denote the end of linked list. Also, head node of the linked list is index 0. This concept of simulating linked list using array would look more interesting if we delete say second node from the above linked list. In that case, the linked list will look as follows in memory:

| | | | | |
|-----------|------------|-----------|------------|------------|
| [(1), 2] | [(0), -1] | [(3), 3] | [(4), -2] | [(0), -1] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

The resultant linked list is 0x500 -> 0x510 -> 0x518. Here, it should be noted that even though we have deleted second node from our linked list, the memory for this node is still there because underlying array is still there. But the nextIndex of first node now points to third node (for which index is 2).

Hopefully, the above examples would have given some idea that for the simulated linked list, we need to write our own API similar to malloc() and free() which would basically be used to insert and delete a node. Now this is what's called own memory management. Let us see how this can be done in algorithmic manner.

There are multiple ways to do so. If we take the simplistic approach of creating linked list using array, we can use the following logic. For inserting a node, traverse the underlying array and find a node whose nextIndex is -1. It means that this node is empty. Use this node as a new node. Update the data part in this new node and set the nextIndex of this node to current head node (i.e. head index) of the linked list. Finally, make the index of this new node as head index of the linked list. To visualize it, let us take an example. Suppose the linked list is as follows where head Index is 0 i.e. linked list is 0x500 -> 0x508 -> 0x518 -> 0x520

| | | | | |
|-----------|-----------|------------|-----------|------------|
| [(1), 1] | [(2), 3] | [(0), -1] | [(4), 4] | [(5), -2] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

After inserting a new node with data 8, the linked list would look as follows with head index as 2.

| | | | | |
|-----------|-----------|-----------|-----------|------------|
| [(1), 1] | [(2), 3] | [(8), 0] | [(4), 4] | [(5), -2] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

So the linked list nodes would be at addresses 0x510 -> 0x500 -> 0x508 -> 0x518 -> 0x520

For deleting a node, we need to set the nextIndex of the node as -1 so that the node is marked as empty node. But, before doing so, we need to make sure that the nextIndex of the previous node is updated correctly to index of next node of this node to be deleted. We can see that we have done own memory management for creating a linked list out of the array memory. But, this is one way of inserting and deleting nodes in this linked list. It can be easily noticed that finding an empty node is not so efficient in terms of time complexity. Basically, we're searching the complete array linearly to find an empty node.

Let us see if we can optimize it further. Basically we can maintain a linked list of empty nodes as well in the same array. In that case, the linked list would be denoted by two indexes one index would be for linked list which has the actual data values i.e. nodes which have been inserted so far and other index would for linked list of empty nodes. By doing so, whenever, we need to insert a new node in existing linked list, we can quickly find an empty node. Let us take an example:

| | | | | | |
|-----------|-----------|-----------|------------|-----------|------------|
| [(4), 2] | [(0), 3] | [(5), 5] | [(0), -1] | [(0), 1] | [(9), -1] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 | 0x528 |

The above linked list which is represented using two indexes (0 and 5) has two linked lists: one for actual values and another for empty nodes. The linked list with actual values has nodes at address 0x500 -> 0x510 -> 0x528 while the linked list with empty nodes has nodes at addresses 0x520 -> 0x508 -> 0x518. It can be seen that finding an empty node (i.e. writing own API similar to malloc()) should be relatively faster now because we can quickly find a free node. In real world embedded programs, a fixed chunk of memory (normally called memory pool) is allocated using malloc() only once by a module. And then the management of this memory pool (which is basically an array) is done by that module itself using techniques mentioned earlier. Sometimes, there are multiple memory pools each one having different size of node. Of course, there are several other aspects of own memory management but well leave it here itself. But its worth mentioning that there are several methods by which the insertion (which requires our own memory allocation) and deletion (which requires our own memory freeing) can be improved further.

If we look carefully, it can be noticed that the Heap section of memory is basically a big array of bytes which is being managed by the underlying operating system (OS). And OS is providing this memory management service to programmers via malloc(), free() etc. Aha !!

The important take-aways from this article can be summed as follows:

- A) Array means contiguous memory. It can exist in any memory section be it Data or Stack or Heap.
- B) Linked List means non-contiguous linked memory. It can exist in any memory section be it Heap or Data or Stack.
- C) As a programmer, looking at a data structure from memory perspective could provide us better insight in choosing a particular data structure or even designing a new data structure. For example, we might create an array of linked lists etc.

[GATE Corner](#)[Quiz Corner](#)

Find Length of a Linked List (Iterative and Recursive)

Write a C function to count number of nodes in a given singly linked list.

For example, the function should return 5 for linked list 1->3->1->2->1.

Iterative Solution

- 1) Initialize count as 0
- 2) Initialize a node pointer, current = head.
- 3) Do following while current is not NULL
 - a) current = current -> next
 - b) count++;
- 4) Return count

Following are C and Python implementations of above algorithm to find count of nodes.

C/C++

```
// Iterative C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Counts no. of nodes in linked list */
int getCount(struct node* head)
{
    int count = 0; // Initialize count
    struct node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
       1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of nodes is %d", getCount(head));
}
```

```
    return 0;  
}
```

Python

```
# A complete working Python program to find length of a  
# Linked List iteratively  
  
# Node class  
class Node:  
    # Function to initialise the node object  
    def __init__(self, data):  
        self.data = data    # Assign data  
        self.next = None    # Initialize next as null  
  
# Linked List class contains a Node object  
class LinkedList:  
  
    # Function to initialize head  
    def __init__(self):  
        self.head = None  
  
    # This function is in LinkedList class. It inserts  
    # a new node at the beginning of Linked List.  
    def push(self, new_data):  
  
        # 1 & 2: Allocate the Node &  
        #          Put in the data  
        new_node = Node(new_data)  
  
        # 3. Make next of new Node as head  
        new_node.next = self.head  
  
        # 4. Move the head to point to new Node  
        self.head = new_node  
  
    # This function counts number of nodes in Linked List  
    # iteratively, given 'node' as starting node.  
    def getCount(self, node):  
        temp = node # Initialise temp  
        count = 0 # Initialise count  
  
        # Loop while end of linked list is not reached  
        while (temp):  
            count += 1  
            temp = temp.next  
        return count  
  
    # Code execution starts here  
if __name__ == '__main__':  
    llist = LinkedList()  
    llist.push(1)  
    llist.push(3)  
    llist.push(1)  
    llist.push(2)  
    llist.push(1)  
    print ('Count of nodes is :',llist.getCount(llist.head))
```

```
count of nodes is 5
```

Recursive Solution

```
int getCount(head)  
1) If head is NULL, return 0.  
2) Else return 1 + getCount(head->next)
```

Following are C and Python implementations of above algorithm to find count of nodes.

C/C++

```

// Recursive C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Counts the no. of occurrences of a node
   (search_for) in a linked list (head) */
int getCount(struct node* head)
{
    // Base case
    if (head == NULL)
        return 0;

    // count is 1 + count of remaining list
    return 1 + getCount(head->next);
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
       1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of nodes is %d", getCount(head));
    return 0;
}

```

Python

```

# A complete working Python program to find length of a
# Linked List recursively

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data    # Assign data
        self.next = None   # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

```

```

# This function is in LinkedList class. It inserts
# a new node at the beginning of Linked List.
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    #          Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node

# This function counts number of nodes in Linked List
# recursively, given 'node' as starting node.
def getCount(self, node):
    if (not node): # Base case
        return 0
    else:
        return 1 + self.getCount(node.next)

# Code execution starts here
if __name__ == '__main__':
    llist = LinkedList()
    llist.push(1)
    llist.push(3)
    llist.push(1)
    llist.push(2)
    llist.push(1)
    print 'Count of nodes is :',llist.getCount(llist.head)

```

count of nodes is 5

[GATE Corner](#)[Quiz Corner](#)

How to write C functions that modify head pointer of a Linked List?

Consider simple representation (without any dummy node) of Linked List. Functions that operate on such Linked lists can be divided in two categories:

- 1) Functions that do not modify the head pointer:** Examples of such functions include, printing a linked list, updating data members of nodes like adding given a value to all nodes, or some other operation which access/update data of nodes
It is generally easy to decide prototype of functions of this category. We can always pass head pointer as an argument and traverse/update the list.
For example, the following function that adds x to data members of all nodes.

```
void addXtoList(struct node *node, int x)
{
    while(node != NULL)
    {
        node->data = node->data + x;
        node = node->next;
    }
}
```

- 2) Functions that modify the head pointer:** Examples include, inserting a node at the beginning (head pointer is always modified in this function), inserting a node at the end (head pointer is modified only when the first node is being inserted), deleting a given node (head pointer is modified when the deleted node is first node). There may be different ways to update the head pointer in these functions. Let us discuss these ways using following simple problem:

Given a linked list, write a function deleteFirst() that deletes the first node of a given linked list. For example, if the list is 1->2->3->4, then it should be modified to 2->3->4?

Algorithm to solve the problem is a simple 3 step process: (a) Store the head pointer (b) change the head pointer to point to next node (c) delete the previous head node.

Following are different ways to update head pointer in deleteFirst() so that the list is updated everywhere.

- 2.1) Make head pointer global:** We can make the head pointer global so that it can be accessed and updated in our function. Following is C code that uses global head pointer.

```
// global head pointer
struct node *head = NULL;

// function to delete first node: uses approach 2.1
// See http://ideone.com/ClfQB for complete program and output
void deleteFirst()
{
    if(head != NULL)
    {
        // store the old value of head pointer
        struct node *temp = head;

        // Change head pointer to point to next node
        head = head->next;

        // delete memory allocated for the previous head node
        free(temp);
    }
}
```

See [this](#) for complete running program that uses above function.

This is not a recommended way as it has many problems like following:

- a) head is globally accessible, so it can be modified anywhere in your project and may lead to unpredictable results.
- b) If there are multiple linked lists, then multiple global head pointers with different names are needed.

See [this](#) to know all reasons why should we avoid global variables in our projects.

- 2.2) Return head pointer:** We can write deleteFirst() in such a way that it returns the modified head pointer. Whoever is using this function, have to use the returned value to update the head node.

```
// function to delete first node: uses approach 2.2
// See http://ideone.com/P5oLe for complete program and output
struct node *deleteFirst(struct node *head)
{
```

```

if(head != NULL)
{
    // store the old value of head pointer
    struct node *temp = head;

    // Change head pointer to point to next node
    head = head->next;

    // delete memory allocated for the previous head node
    free(temp);
}

return head;
}

```

See [this](#) for complete program and output.

This approach is much better than the previous 1. There is only one issue with this, if user misses to assign the returned value to head, then things become messy. C/C++ compilers allows to call a function without assigning the returned value.

```

head = deleteFirst(head); // proper use of deleteFirst()
deleteFirst(head); // improper use of deleteFirst(), allowed by compiler

```

2.3) Use Double Pointer: This approach follows the simple C rule: *if you want to modify local variable of one function inside another function, pass pointer to that variable.* So we can pass pointer to the head pointer to modify the head pointer in our deleteFirst() function.

```

// function to delete first node: uses approach 2.3
// See http://ideone.com/9GwTb for complete program and output
void deleteFirst(struct node **head_ref)
{
    if(*head_ref != NULL)
    {
        // store the old value of pointer to head pointer
        struct node *temp = *head_ref;

        // Change head pointer to point to next node
        *head_ref = (*head_ref)->next;

        // delete memory allocated for the previous head node
        free(temp);
    }
}

```

See [this](#) for complete program and output.

This approach seems to be the best among all three as there are less chances of having problems.

Swap nodes in a linked list without swapping data

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields.

It may be assumed that all keys in linked list are distinct.

Examples:

Input: 10->15->12->13->20->14, x = 12, y = 20
Output: 10->15->20->13->12->14

Input: 10->15->12->13->20->14, x = 10, y = 20
Output: 20->15->12->13->10->14

Input: 10->15->12->13->20->14, x = 12, y = 13
Output: 10->15->13->12->20->14

This may look a simple problem, but is interesting question as it has following cases to be handled.

- 1) x and y may or may not be adjacent.
- 2) Either x or y may be a head node.
- 3) Either x or y may be last node.
- 4) x and/or y may not be present in linked list.

How to write a clean working code that handles all of the above possibilities.

The idea is to first search x and y in given linked list. If any of them is not present, then return. While searching for x and y, keep track of current and previous pointers. First change next of previous pointers, then change next of current pointers. Following is C implementation of this approach.

```
/* This program swaps the nodes of linked list rather
   than swapping the field from the nodes.*/

#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
    int data;
    struct node *next;
};

/* Function to swap nodes x and y in linked list by
   changing links */
void swapNodes(struct node **head_ref, int x, int y)
{
    // Nothing to do if x and y are same
    if (x == y) return;

    // Search for x (keep track of prevX and currX)
    struct node *prevX = NULL, *currX = *head_ref;
    while (currX && currX->data != x)
    {
        prevX = currX;
        currX = currX->next;
    }

    // Search for y (keep track of prevY and currY)
    struct node *prevY = NULL, *currY = *head_ref;
    while (currY && currY->data != y)
    {
        prevY = currY;
        currY = currY->next;
    }

    // If either x or y is not present, nothing to do
    if (currX == NULL || currY == NULL)
        return;

    // If x is not head of linked list
    if (prevX != NULL)
        prevX->next = currY;
    else // Else make y as new head
        *head_ref = currY;

    // If y is not head of linked list
    if (prevY != NULL)
```

```

    prevY->next = currX;
else // Else make x as new head
    *head_ref = currX;

// Swap next pointers
struct node *temp = currY->next;
currY->next = currX->next;
currX->next = temp;
}

/* Function to add a node at the begining of List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function */
int main()
{
    struct node *start = NULL;

    /* The constructed linked list is:
     1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling swapNodes() ");
    printList(start);

    swapNodes(&start, 4, 3);

    printf("\n Linked list after calling swapNodes() ");
    printList(start);

    return 0;
}

```

Output:

```

Linked list before calling swapNodes() 1 2 3 4 5 6 7
Linked list after calling swapNodes() 1 2 4 3 5 6 7

```

Optimizations: The above code can be optimized to search x and y in single traversal. Two loops are used to keep program simple.

Write a function to reverse a linked list

Iterative Method

Iterate through the linked list. In loop, change next to prev, prev to current and current to next.

Implementation of Iterative Method

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

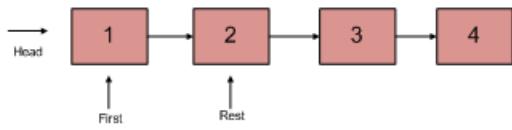
    printList(head);
    reverse(&head);
    printf("\n Reversed Linked list \n");
    printList(head);
    getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

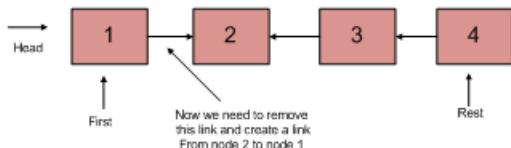
Recursive Method:

- 1) Divide the list in two parts - first node and rest of the linked list.
- 2) Call reverse for the rest of the linked list.
- 3) Link rest to first.
- 4) Fix head pointer

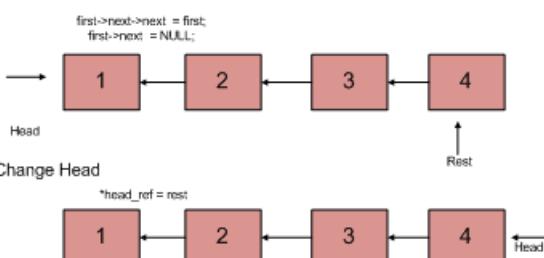
Divide the List in two parts



Reverse Rest



Link Rest to First



```
void recursiveReverse(struct node** head_ref)
{
    struct node* first;
    struct node* rest;

    /* empty list */
    if (*head_ref == NULL)
        return;

    /* suppose first = {1, 2, 3}, rest = {2, 3} */
    first = *head_ref;
    rest = first->next;

    /* List has only one node */
    if (rest == NULL)
        return;

    /* reverse the rest list and put the first element at the end */
    recursiveReverse(&rest);
    first->next->next = first;

    /* tricky step -- see the diagram */
    first->next = NULL;

    /* fix the head pointer */
    *head_ref = rest;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

A Simpler and Tail Recursive Method

Below is C++ implementation of this method.

```
// A simple and tail recursive C++ program to reverse
// a linked list
#include<bits/stdc++.h>
```

```

using namespace std;

struct node
{
    int data;
    struct node *next;
};

void reverseUtil(node *curr, node *prev, node **head);

// This function mainly calls reverseUtil()
// with prev as NULL
void reverse(node **head)
{
    if (!head)
        return;
    reverseUtil(*head, NULL, head);
}

// A simple and tail recursive function to reverse
// a linked list. prev is passed as NULL initially.
void reverseUtil(node *curr, node *prev, node **head)
{
    /* If last node mark it head*/
    if (!curr->next)
    {
        *head = curr;

        /* Update next to prev node */
        curr->next = prev;
        return;
    }

    /* Save curr->next node for recursive call */
    node *next = curr->next;

    /* and update next ..*/
    curr->next = prev;

    reverseUtil(next, curr, head);
}

// A utility function to create a new node
node *newNode(int key)
{
    node *temp = new node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printlist(node *head)
{
    while(head != NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// Driver program to test above functions
int main()
{
    node *head1 = newNode(1);
    head1->next = newNode(2);
    head1->next->next = newNode(2);
    head1->next->next->next = newNode(4);
    head1->next->next->next->next = newNode(5);
    head1->next->next->next->next->next = newNode(6);
    head1->next->next->next->next->next->next = newNode(7);
    head1->next->next->next->next->next->next = newNode(8);
    cout << "Given linked list\n";
    printlist(head1);
    reverse(&head1);
    cout << "\nReversed linked list\n";
    printlist(head1);
    return 0;
}

```

Output:

Given linked list
1 2 2 4 5 6 7 8

Reversed linked list
8 7 6 5 4 2 2 1

Thanks to Gaurav Ahirwar for suggesting this solution.

References:

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return a pointer to the head node of the merged list.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either a or b may be empty, during processing either a or b may run out first, and finally theres the problem of starting the result list empty, and building it up while going through a and b.

Method 1 (Using Dummy Nodes)

The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.

The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either a or b, and adding it to tail. When we are done, the result is in dummy.next.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef);

/* Takes two lists sorted in increasing order, and splices their nodes together to make one big sorted list which is returned. */
struct node* SortedMerge(struct node* a, struct node* b)
{
    /* a dummy first node to hang the result on */
    struct node dummy;

    /* tail points to the last result node */
    struct node* tail = &dummy;

    /* so tail->next is the place to add new nodes
       to the result. */
    dummy.next = NULL;
    while(1)
    {
        if(a == NULL)
        {
            /* if either list runs out, use the other list */
            tail->next = b;
            break;
        }
        else if (b == NULL)
        {
            tail->next = a;
            break;
        }
        if (a->data <= b->data)
        {
            MoveNode(&(tail->next), &a);
        }
        else
        {
            MoveNode(&(tail->next), &b);
        }
        tail = tail->next;
    }
    return(dummy.next);
}

/* UTILITY FUNCTIONS */
/*MoveNode() function takes the node from the front of the source, and move it to the front of the dest.
It is an error to call this with the source list empty.

Before calling MoveNode():
source == {1, 2, 3}
dest == {1, 2, 3}

After calling MoveNode():
source == {2, 3}
dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
    /* the front source node */
    struct node* newNode = *sourceRef;
    assert(newNode != NULL);
```

```

/* Advance the source pointer */
*sourceRef = newNode->next;

/* Link the old dest off the new node */
newNode->next = *destRef;

/* Move dest to point to the new node */
*destRef = newNode;
}

/* Function to insert a node at the beginning of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* res = NULL;
    struct node* a = NULL;
    struct node* b = NULL;

    /* Let us create two sorted linked lists to test the functions
     * Created lists shall be a: 5->10->15, b: 2->3->20 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);

    push(&b, 20);
    push(&b, 3);
    push(&b, 2);

    /* Remove duplicates from linked list */
    res = SortedMerge(a, b);

    printf("\n Merged Linked List is: \n");
    printList(res);

    getchar();
    return 0;
}

```

Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** reference strategy can be used (see Section 1 for details).

```

struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* point to the last result pointer */
    struct node** lastPtrRef = &result;

    while(1)
    {
        if (a == NULL)
        {
            *lastPtrRef = b;
            break;
        }
        else if (b==NULL)
        {
            *lastPtrRef = a;
            break;
        }
        if(a->data <= b->data)
        {

```

```

        MoveNode(lastPtrRef, &a);
    }
    else
    {
        MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
}
return(result);
}

```

Method 3 (Using Recursion)

Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```

struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

```

Source: <http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Merge Sort for Linked Lists

[Merge sort](#) is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
   then return.
2) Else divide the linked list into two halves.
   FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
   MergeSort(a);
   MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
   *headRef = SortedMerge(a, b);

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
    struct node* head = *headRef;
    struct node* a;
    struct node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
```

```

    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct node* source,
                     struct node** frontRef, struct node** backRef)
{
    struct node* fast;
    struct node* slow;
    if (source==NULL || source->next==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
    else
    {
        slow = source;
        fast = source->next;

        /* Advance 'fast' two nodes, and advance 'slow' one node */
        while (fast != NULL)
        {
            fast = fast->next;
            if (fast != NULL)
            {
                slow = slow->next;
                fast = fast->next;
            }
        }

        /* 'slow' is before the midpoint in the list, so split it in two
           at that point. */
        *frontRef = source;
        *backRef = slow->next;
        slow->next = NULL;
    }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Function to insert a node at the beginning of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* res = NULL;
    struct node* a = NULL;
}

```

```
/* Let us create a unsorted linked lists to test the functions
 Created lists shall be a: 2->3->20->5->10->15 */
push(&a, 15);
push(&a, 10);
push(&a, 5);
push(&a, 20);
push(&a, 3);
push(&a, 2);

/* Sort the above created Linked List */
MergeSort(&a);

printf("\n Sorted Linked List is: \n");
printList(a);

getchar();
return 0;
}
```

Time Complexity: O(nLogn)

Sources:

http://en.wikipedia.org/wiki/Merge_sort

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

Example:

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3
Output: 3->2->1->6->5->4->8->7->NULL.

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5
Output: 5->4->3->2->1->8->7->6->NULL.

Algorithm: *reverse(head, k)*

- 1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See [this post](#) for reversing a linked list.
- 2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */
- 3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of [this post](#)) */

C/C++

```
// C program to reverse a linked list in groups of given size
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the
   pointer to the new head node. */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next = NULL;
    struct node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if (next != NULL)
        head->next = reverse(next, k);

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
```

```

void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nGiven linked list \n");
    printList(head);
    head = reverse(head, 3);

    printf("\nReversed Linked list \n");
    printList(head);

    return(0);
}

```

Java

```

// Java program to reverse a linked list in groups of
// given size
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    Node reverse(Node head, int k)
    {
        Node current = head;
        Node next = null;
        Node prev = null;

        int count = 0;

        /* Reverse first k nodes of linked list */
        while (count < k && current != null)
        {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            count++;
        }

        /* next is now a pointer to (k+1)th node
           Recursively call for the list starting from current.
           And make rest of the list as next of first node */
        if (next != null)
            head.next = reverse(next, k);

        // prev is now head of input list
        return prev;
    }
}

```

```

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 1->2->3->4->5->6->
       7->8->9->null */
    llist.push(9);
    llist.push(8);
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List");
    llist.printList();

    llist.head = llist.reverse(llist.head, 3);

    System.out.println("Reversed list");
    llist.printList();
}
}

/* This code is contributed by Rajat Mishra */

```

```

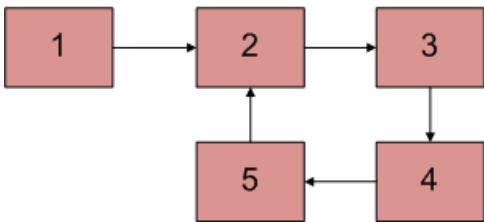
Given Linked List
1 2 3 4 5 6 7 8 9
Reversed list
3 2 1 6 5 4 9 8 7

```

Time Complexity: O(n) where n is the number of nodes in the given list.

Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We recommend to read following post as a prerequisite.

[Write a C function to detect loop in a linked list](#)

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say *ptr2*. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from *ptr2*. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}
```

```

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1;
    struct node *ptr2;

    /* Set a pointer to the begining of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = head;
    while (1)
    {
        /* Now start a pointer from loop_node and check if it ever
           reaches ptr2 */
        ptr2 = loop_node;
        while (ptr2->next != loop_node && ptr2->next != ptr1)
            ptr2 = ptr2->next;

        /* If ptr2 reached ptr1 then there is a loop. So break the
           loop */
        if (ptr2->next == ptr1)
            break;

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1->next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
       make next of ptr2 as NULL */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}

```

Output:

```

Linked List after removing loop
50 20 15 4 10

```

Method 2 (Better Solution)

This method is also dependent on Floyd's Cycle detection algorithm

- 1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
- 2) Count the number of nodes in loop. Let the count be k.

- 3) Fix one pointer to the head and another to kth node from head.
- 4) Move both pointers at the same pace, they will meet at loop starting node.
- 5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}

/* Function to remove loop.
   loop_node --> Pointer to one of the loop nodes
   head --> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
        ptr2 = ptr2->next;

    /* Move both pointers at the same pace,
       they will meet at loop starting node */
    while (ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
}
```

```

while (ptr2->next != ptr1)
    ptr2 = ptr2->next;

/* Set the next node of the loop ending node
   to fix the loop */
ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}

```

Output:

```

Linked List after removing loop
50 20 15 4 10

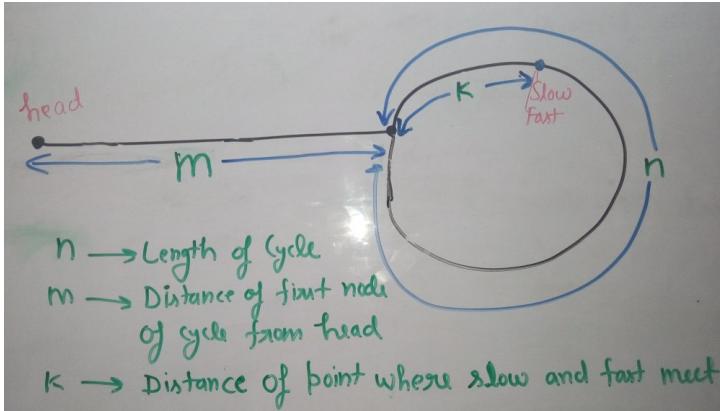
```

Method 3 (Optimized Method 2: Without Counting Nodes in Loop)

We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast dont meet, they would meet at the beginning of linked list.

How does this work?

Let slow and fast meet at some point after Floyds Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

$$\text{Distance traveled by fast pointer} = 2 * (\text{Distance traveled by slow pointer})$$

$$(m + n*x + k) = 2*(m + n*y + k)$$

Note that before meeting the point shown above, fast was moving at twice speed.

x --> Number of complete cyclic rounds made by fast pointer before they meet first time
y --> Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x - 2y) * n$$

Which means **m+k is a multiple of n**.

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of linked list (has made m steps). Fast pointer would have made also moved m steps as they are now moving same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

void detectAndRemoveLoop (Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;
        while (slow != fast->next)
        {
            slow = slow->next;
            fast = fast->next;
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}
```

```
/* Drier program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}
```

Output:

```
Linked List after removing loop
50 20 15 4 10
```

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

Add two numbers represented by linked lists | Set 1

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

Input:
First List: 5->6->3 // represents number 365
Second List: 8->4->2 // represents number 248
Output
Resultant list: 3->1->6 // represents number 613

Example 2

Input:
First List: 7->5->9->4->6 // represents number 64957
Second List: 8->4 // represents number 48
Output
Resultant list: 5->0->0->5->6 // represents number 65005

Solution

Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is C implementation of this approach.

```
#include<stdio.h>
#include<stdlib.h>

/* Linked list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to create a new node with given data */
struct node *newNode(int data)
{
    struct node *new_node = (struct node *) malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = newNode(new_data);

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Adds contents of two linked lists and return the head node of resultant list */
struct node* addTwoLists (struct node* first, struct node* second)
{
    struct node* res = NULL; // res is head node of the resultant list
    struct node *temp, *prev = NULL;
    int carry = 0, sum;

    while (first != NULL || second != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i) Carry
        // (ii) Next digit of first list (if there is a next digit)
        // (iii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);

        // update carry for next calculation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;
        temp = newNode(sum);
        if (res == NULL)
            res = temp;
        else
            prev->next = temp;
        prev = temp;
        if (first)
            first = first->next;
        if (second)
            second = second->next;
    }
    return res;
}
```

```

// Create a new node with sum as data
temp = newNode(sum);

// if this is the first node then set it as head of the resultant list
if(res == NULL)
    res = temp;
else // If this is not the first node then connect it to the rest.
    prev->next = temp;

// Set prev for next insertion
prev = temp;

// Move first and second pointers to next nodes
if (first) first = first->next;
if (second) second = second->next;
}

if (carry > 0)
    temp->next = newNode(carry);

// return head of the resultant list
return res;
}

// A utility function to print a linked list
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    struct node* res = NULL;
    struct node* first = NULL;
    struct node* second = NULL;

    // create first list 7->5->9->4->6
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 5);
    push(&first, 7);
    printf("First List is ");
    printList(first);

    // create second list 8->4
    push(&second, 4);
    push(&second, 8);
    printf("Second List is ");
    printList(second);

    // Add the two lists and see result
    res = addTwoLists(first, second);
    printf("Resultant list is ");
    printList(res);

    return 0;
}

```

Output:

```

First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6

```

Time Complexity: O(m + n) where m and n are number of nodes in first and second lists respectively.

Rotate a Linked List

Given a singly linked list, rotate the linked list counter-clockwise by k nodes. Where k is a given positive integer. For example, if the given linked list is 10->20->30->40->50->60 and k is 4, the list should be modified to 50->60->10->20->30->40. Assume that k is smaller than the count of nodes in linked list.

To rotate the linked list, we need to change next of kth node to NULL, next of last node to previous head node, and finally change head to (k+1)th node. So we need to get hold of three nodes: kth node, (k+1)th node and last node.

Traverse the list from beginning and stop at kth node. Store pointer to kth node. We can get (k+1)th node using kthNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above.

C/C++

```
// C/C++ program to rotate a linked list counter clock wise

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// This function rotates a linked list counter-clockwise and
// updates the head. The function assumes that k is smaller
// than size of linked list. It doesn't modify the list if
// k is greater than or equal to size
void rotate(struct node **head_ref, int k)
{
    if (k == 0)
        return;

    // Let us understand the below code for example k = 4 and
    // list = 10->20->30->40->50->60.
    struct node* current = *head_ref;

    // current will either point to kth or NULL after this loop.
    // current will point to node 40 in the above example
    int count = 1;
    while (count < k && current != NULL)
    {
        current = current->next;
        count++;
    }

    // If current is NULL, k is greater than or equal to count
    // of nodes in linked list. Don't change the list in this case
    if (current == NULL)
        return;

    // current points to kth node. Store it in a variable.
    // kthNode points to node 40 in the above example
    struct node *kthNode = current;

    // current will point to last node after this loop
    // current will point to node 60 in the above example
    while (current->next != NULL)
        current = current->next;

    // Change next of last node to previous head
    // Next of 60 is now changed to node 10
    current->next = *head_ref;

    // Change head to (k+1)th node
    // head is now changed to node 50
    *head_ref = kthNode->next;

    // change next of kth node to NULL
    // next of 40 is now NULL
    kthNode->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push (struct node** head_ref, int new_data)
{
```

```

/* allocate node */
struct node* new_node =
    (struct node*) malloc(sizeof(struct node));

/* put in the data */
new_node->data = new_data;

/* link the old list off the new node */
new_node->next = (*head_ref);

/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 10->20->30->40->50->60
    for (int i = 60; i > 0; i -= 10)
        push(&head, i);

    printf("Given linked list \n");
    printList(head);
    rotate(&head, 4);

    printf("\nRotated Linked list \n");
    printList(head);

    return (0);
}

```

Java

```

// Java program to rotate a linked list

class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    // This function rotates a linked list counter-clockwise
    // and updates the head. The function assumes that k is
    // smaller than size of linked list. It doesn't modify
    // the list if k is greater than or equal to size
    void rotate(int k)
    {
        if (k == 0) return;

        // Let us understand the below code for example k = 4
        // and list = 10->20->30->40->50->60.
        Node current = head;

        // current will either point to kth or NULL after this
        // loop. current will point to node 40 in the above example
        int count = 1;
        while (count < k && current != null)

```

```

{
    current = current.next;
    count++;
}

// If current is NULL, k is greater than or equal to count
// of nodes in linked list. Don't change the list in this case
if (current == null)
    return;

// current points to kth node. Store it in a variable.
// kthNode points to node 40 in the above example
Node kthNode = current;

// current will point to last node after this loop
// current will point to node 60 in the above example
while (current.next != null)
    current = current.next;

// Change next of last node to previous head
// Next of 60 is now changed to node 10

current.next = head;

// Change head to (k+1)th node
// head is now changed to node 50
head = kthNode.next;

// change next of kth node to null
kthNode.next = null;

}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

void printList()
{
    Node temp = head;
    while(temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    // create a list 10->20->30->40->50->60
    for (int i = 60; i >= 10; i -= 10)
        llist.push(i);

    System.out.println("Given list");
    llist.printList();

    llist.rotate(4);

    System.out.println("Rotated Linked List");
    llist.printList();
}
} /* This code is contributed by Rajat Mishra */

```

```
Given linked list  
10 20 30 40 50 60  
Rotated Linked list  
50 60 10 20 30 40
```

Time Complexity: O(n) where n is the number of nodes in Linked List. The code traverses the linked list only once.

Generic Linked List in C

Unlike [C++](#) and [Java](#), [C](#) doesn't support generics. How to create a linked list in C that can be used for any data type? In C, we can use [void pointer](#) and function pointer to implement the same functionality. The great thing about void pointer is it can be used to point to any data type. Also, size of all types of pointers is always the same, so we can always allocate a linked list node. Function pointer is needed to process actual content stored at address pointed by void pointer.

Following is a sample C code to demonstrate working of generic linked list.

```
// C program for generic linked list
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
    // Any data type can be stored in this node
    void *data;

    struct node *next;
};

/* Function to add a node at the beginning of Linked List.
   This function expects a pointer to the data to be added
   and size of the data type */
void push(struct node** head_ref, void *new_data, size_t data_size)
{
    // Allocate memory for node
    struct node* new_node = (struct node*)malloc(sizeof(struct node));

    new_node->data = malloc(data_size);
    new_node->next = (*head_ref);

    // Copy contents of new_data to newly allocated memory.
    // Assumption: char takes 1 byte.
    int i;
    for (i=0; i<data_size; i++)
        *(char*)(new_node->data + i) = *(char*)(new_data + i);

    // Change head pointer as new node is added at the beginning
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list. fpitr is used
   to access the function to be used for printing current node data.
   Note that different data types need different specifier in printf() */
void printList(struct node *node, void (*fpitr)(void *))
{
    while (node != NULL)
    {
        (*fpitr)(node->data);
        node = node->next;
    }
}

// Function to print an integer
void printInt(void *n)
{
    printf(" %d", *(int *)n);
}

// Function to print a float
void printFloat(void *f)
{
    printf(" %f", *(float *)f);
}

/* Driver program to test above function */
int main()
{
    struct node *start = NULL;

    // Create and print an int linked list
    unsigned int_size = sizeof(int);
    int arr[] = {10, 20, 30, 40, 50}, i;
    for (i=4; i>=0; i--)
        push(&start, &arr[i], int_size);
    printf("Created integer linked list is \n");
    printList(start, printInt);
```

```
// Create and print a float linked list
unsigned float_size = sizeof(float);
start = NULL;
float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};
for (i=4; i>=0; i--)
    push(&start, &arr2[i], float_size);
printf("\n\nCreated float linked list is \n");
printList(start, printFloat);

return 0;
}
```

Output:

```
Created integer linked list is
10 20 30 40 50

Created float linked list is
10.100000 20.200001 30.299999 40.400002 50.500000
```

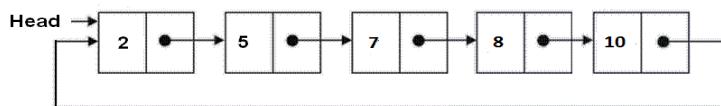
Circular Linked List | Set 1 (Introduction and Applications)

We have discussed singly and doubly linked lists in the following posts.

[Introduction to Linked List & Insertion](#)

[Doubly Linked List Introduction and Insertion](#)

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike [this](#) implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list. (Source <http://web.eecs.utk.edu/~bvz/cs140/notes/Dlists/>)

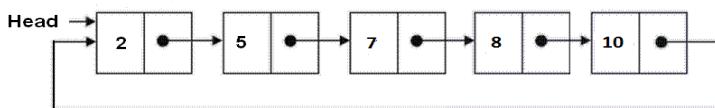
4) Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](#).

We will soon be discussing implementation of insert delete operations for circular linked lists.

[GATE Corner](#)[Quiz Corner](#)

Circular Linked List | Set 2 (Traversal)

We have discussed [Circular Linked List Introduction and Applications](#), in the previous post on Circular Linked List. In this post, traversal operation is discussed.



In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again. Following is C code for linked list traversal.

```
/* Function to traverse a given Circular linked list and print nodes */
void printList(struct node *first)
{
    struct node *temp = first;

    // If linked list is not empty
    if (first != NULL)
    {
        // Keep printing nodes till we reach the first node again
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != first);
    }
}
```

Complete C program to demonstrate traversal. Following is complete C program to demonstrate traversal of circular linked list.

```
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
    int data;
    struct node *next;
};

/* Function to insert a node at the begining of a Circular
linked list */
void push(struct node **head_ref, int data)
{
    struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
    struct node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the next of last node */
    if (*head_ref != NULL)
    {
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    if (head != NULL)
    {
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != head);
    }
}
```

```
}

/* Driver program to test above functions */
int main()
{
    /* Initialize lists as empty */
    struct node *head = NULL;

    /* Created linked list will be 12->56->2->11 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    printf("Contents of Circular Linked List\n ");
    printList(head);

    return 0;
}
```

Output:

```
Contents of Circular Linked List
11 2 56 12
```

You may like to see following posts on Circular Linked List

[Split a Circular Linked List into two halves](#)

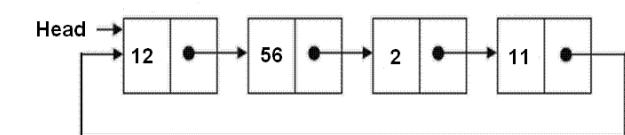
[Sorted insert for circular linked list](#)

We will soon be discussing implementation of insert delete operations for circular linked lists.

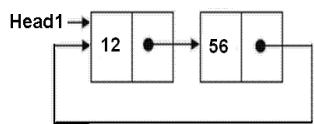
[GATE Corner](#)[Quiz Corner](#)

Split a Circular Linked List into two halves

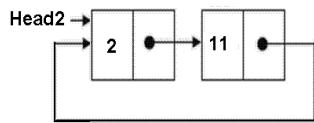
Asked by [Bharani](#)



Original Linked List



Result Linked List 1



Result Linked List 2

Thanks to [Geek4u](#) for suggesting the algorithm

- 1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
- 2) Make the second half circular.
- 3) Make the first half circular.
- 4) Set head (or start) pointers of the two linked lists.

In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

```
/* Program to split a circular linked list into two halves */
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
    int data;
    struct node *next;
};

/* Function to split a list (starting with head) into two lists.
   head1_ref and head2_ref are references to head nodes of
   the two resultant linked lists */
void splitList(struct node *head, struct node **head1_ref,
               struct node **head2_ref)
{
    struct node *slow_ptr = head;
    struct node *fast_ptr = head;

    if(head == NULL)
        return;

    /* If there are odd nodes in the circular list then
       fast_ptr->next becomes head and for even nodes
       fast_ptr->next->next becomes head */
    while(fast_ptr->next != head &&
          fast_ptr->next->next != head)
    {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }

    /* If there are even elements in list then move fast_ptr */
    if(fast_ptr->next->next == head)
        fast_ptr = fast_ptr->next;

    /* Set the head pointer of first half */
    *head1_ref = head;
```

```

/* Set the head pointer of second half */
if(head->next != head)
    *head2_ref = slow_ptr->next;

/* Make second half circular */
fast_ptr->next = slow_ptr->next;

/* Make first half circular */
slow_ptr->next = head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the begining of a Circular
linked lsit */
void push(struct node **head_ref, int data)
{
    struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
    struct node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the next of
    last node */
    if(*head_ref != NULL)
    {
        while(temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    if(head != NULL)
    {
        printf("\n");
        do {
            printf("%d ", temp->data);
            temp = temp->next;
        } while(temp != head);
    }
}

/* Driver program to test above functions */
int main()
{
    int list_size, i;

    /* Initialize lists as empty */
    struct node *head = NULL;
    struct node *head1 = NULL;
    struct node *head2 = NULL;

    /* Created linked list will be 12->56->2->11 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    printf("Original Circular Linked List");
    printList(head);

    /* Split the list */
    splitList(head, &head1, &head2);

    printf("\nFirst Circular Linked List");
    printList(head1);

    printf("\nSecond Circular Linked List");
    printList(head2);

    getchar();
    return 0;
}

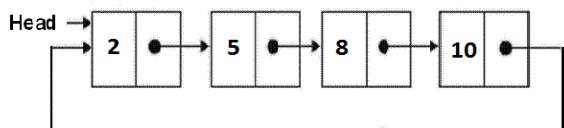
```

Time Complexity: $O(n)$

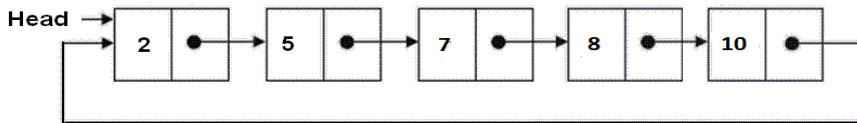
Sorted insert for circular linked list

Difficulty Level: Rookie

Write a C function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After insertion of 7, the above CLL should be changed to following



Algorithm:

Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be `new_node`. After memory allocation, following are the three cases that need to be handled.

- 1) *Linked List is empty:*
 - a) since `new_node` is the only node in CLL, make a self loop.
`new_node->next = new_node;`
 - b) change the head pointer to point to new node.
`*head_ref = new_node;`
- 2) *New node is to be inserted just before the head node:*
 - (a) Find out the last node using a loop.
`while(current->next != *head_ref)
 current = current->next;`
 - (b) Change the next of last node.
`current->next = new_node;`
 - (c) Change next of new node to point to head.
`new_node->next = *head_ref;`
 - (d) change the head pointer to point to new node.
`*head_ref = new_node;`
- 3) *New node is to be inserted somewhere after the head:*
 - (a) Locate the node after which new node is to be inserted.
`while (current->next != *head_ref &&
 current->next->data < new_node->data)
 {
 current = current->next;
 }`
 - (b) Make next of `new_node` as next of the located pointer
`new_node->next = current->next;`
 - (c) Change the next of the located pointer
`current->next = new_node;`

```
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
    int data;
    struct node *next;
};

/* function to insert a new_node in a list in sorted way.
Note that this function expects a pointer to head node
as this can modify the head of the input linked list */
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current = *head_ref;

    // Case 1 of the above algo
    if (current == NULL)
    {
        new_node->next = new_node;
        *head_ref = new_node;
    }

    // Case 2 of the above algo
    else if (current->data >= new_node->data)
    {
        /* If value is smaller than head's value then
        we need to change next of last node */
        new_node->next = current->next;
        current->next = new_node;
    }
    else
    {
        struct node* temp = current;
        while (temp->next != *head_ref &&
               temp->next->data < new_node->data)
            temp = temp->next;
        new_node->next = temp->next;
        temp->next = new_node;
    }
}
```

```

while(current->next != *head_ref)
    current = current->next;
current->next = new_node;
new_node->next = *head_ref;
*head_ref = new_node;
}

// Case 3 of the above algo
else
{
    /* Locate the node before the point of insertion */
    while (current->next!= *head_ref && current->next->data < new_node->data)
        current = current->next;

    new_node->next = current->next;
    current->next = new_node;
}
}

/* Function to print nodes in a given linked list */
void printList(struct node *start)
{
    struct node *temp;

    if(start != NULL)
    {
        temp = start;
        printf("\n");
        do {
            printf("%d ", temp->data);
            temp = temp->next;
        } while(temp != start);
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 56, 2, 11, 1, 90};
    int list_size, i;

    /* start with empty linked list */
    struct node *start = NULL;
    struct node *temp;

    /* Create linked list from the array arr[].
     * Created linked list will be 1->2->11->56->12 */
    for(i = 0; i< 6; i++)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->data = arr[i];
        sortedInsert(&start, temp);
    }

    printList(start);
    getchar();
    return 0;
}

```

Output:

1 2 11 12 56 90

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Case 2 of the above algorithm/code can be optimized. Please see [this](#) comment from Pavan. To implement the suggested change we need to modify the case 2 to following.

```

// Case 2 of the above algo
else if (current->data >= new_node->data)
{
    // swap the data part of head node and new node
    swap(&(current->data), &(new_node->data)); // assuming that we have a function swap(int *, int *)

    new_node->next = (*head_ref)->next;
    (*head_ref)->next = new_node;
}

```

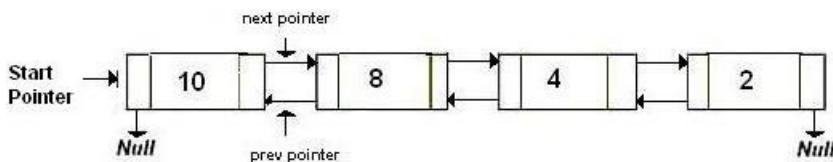

Doubly Linked List | Set 1 (Introduction and Insertion)

We strongly recommend to refer following post as a prerequisite of this post.

[Linked List Introduction](#)

[Inserting a node in Singly Linked List](#)

A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language.

```
/* Node of a doubly linked list */
struct node
{
    int data;
    struct node *next; // Pointer to next node in DLL
    struct node *prev; // Pointer to previous node in DLL
};
```

Following are advantages/disadvantages of doubly linked list over singly linked list.

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.

2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See [this](#) and [this](#)).

2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Insertion

A node can be added in four ways

- 1) At the front of the DLL
- 2) After a given node.
- 3) At the end of the DLL
- 4) Before a given node.

1) Add a node at the front: (A 5 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is 10<->15<->20<->25 and we add an item 5 at the front, then the Linked List becomes 5<->10<->15<->20<->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See [this](#))

Following are the 5 steps to add node at the front.

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node;
```

```

/* 5. move the head to point to the new node */
(*head_ref) = new_node;
}

```

Four steps of the above five steps are same as [the 4 steps used for inserting at the front in singly linked list](#). The only extra step is to change previous of head.

2) Add a node after a given node.: (A 7 steps process)

We are given pointer to a node as prev_node, and the new node is inserted after the given node.

```

/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}

```

Five of the above steps step process are same as [the 5 steps used for inserting after a given node in singly linked list](#). The two extra steps are needed to change previous pointer of new node and previous pointer of new nodes next node.

3) Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 5<->10<->15<->20<->25 and we add an item 30 at the end, then the DLL becomes 5<->10<->15<->20<->25<->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

Following are the 7 steps to add node at the end.

```

/* Given a reference (pointer to pointer) to the head
   of a DLL and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5 */

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
       make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
       node as head */
    if (*head_ref == NULL)
    {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;
}

```

```

/* 6. Change the next of last node */
last->next = new_node;

/* 7. Make last node as previous of new node */
new_node->prev = last;

return;
}

```

Six of the above 7 steps are same as [the 6 steps used for inserting after a given node in singly linked list](#). The one extra step is needed to change previous pointer of new node.

4) Add a node before a given node

This is left as an exercise for the readers.

A complete working program to test above functions.

Following is complete C program to test above functions.

```

// A complete working C program to demonstrate all insertion methods
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)

```

```

        new_node->next->prev = new_node;
    }

/* Given a reference (pointer to pointer) to the head
   of a DLL and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
       make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
       node as head */
    if (*head_ref == NULL)
    {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}

// This function prints contents of linked list starting from the given node
void printList(struct node *node)
{
    struct node *last;
    printf("\nTraversal in forward direction \n");
    while (node != NULL)
    {
        printf(" %d ", node->data);
        last = node;
        node = node->next;
    }

    printf("\nTraversal in reverse direction \n");
    while (last != NULL)
    {
        printf(" %d ", last->data);
        last = last->prev;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);
}

```

```
printf("\n Created DLL is: ");
printList(head);

getchar();
return 0;
}
```

Output:

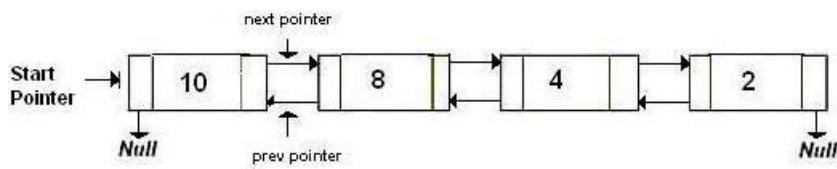
```
Created DLL is:
Traversal in forward direction
1 7 8 6 4
Traversal in reverse direction
4 6 8 7 1
```

[GATE Corner](#)[Quiz Corner](#)

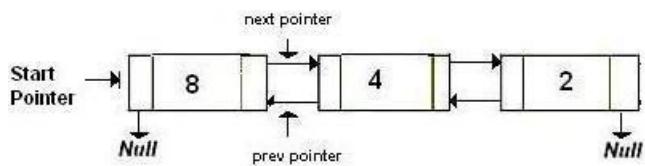
Delete a node in a Doubly Linked List

Write a function to delete a given node in a doubly linked list.

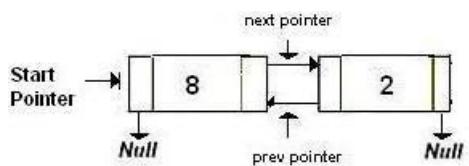
(a) Original Doubly Linked List



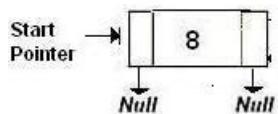
(a) After deletion of head node



(a) After deletion of middle node



(a) After deletion of last node



Algorithm

Let the node to be deleted is *del*.

- 1) If node to be deleted is head node, then change the head pointer to next current head.
- 2) Set *next* of previous to *del*, if previous to *del* exists.
- 3) Set *prev* of *next* to *del*, if next to *del* exists.

```
#include <stdio.h>
#include <stdlib.h>

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del --> pointer to node to be deleted. */
void deleteNode(struct node **head_ref, struct node *del)
{
    /* base case */
    if(*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if(*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted is NOT the last node */
}
```

```

if(del->next != NULL)
    del->next->prev = del->prev;

/* Change prev only if node to be deleted is NOT the first node */
if(del->prev != NULL)
    del->prev->next = del->next;

/* Finally, free the memory occupied by del*/
free(del);
return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the begining,
       prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Let us create the doubly linked list 10<->8<->4<->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    printf("\n Original Linked list ");
    printList(head);

    /* delete nodes from the doubly linked list */
    deleteNode(&head, head); /*delete first node*/
    deleteNode(&head, head->next); /*delete middle node*/
    deleteNode(&head, head->next); /*delete last node*/

    /* Modified linked list will be NULL<->8<->NULL */
    printf("\n Modified Linked list ");
    printList(head);

    getchar();
}

```

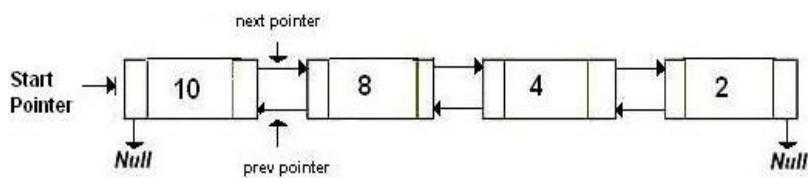
Time Complexity: O(1)
Time Complexity: O(1)

Reverse a Doubly Linked List

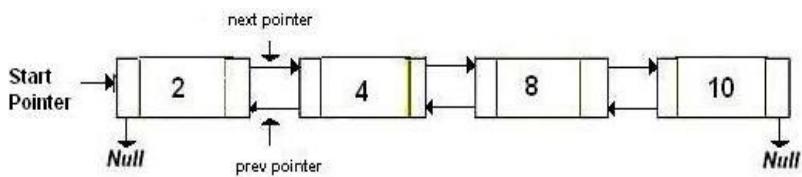
Write a C function to reverse a given Doubly Linked List

See below diagrams for example.

(a) Original Doubly Linked List



(b) Reversed Doubly Linked List



Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

```
/* Program to reverse a doubly linked list */
#include <stdio.h>
#include <stdlib.h>

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* Function to reverse a Doubly Linked List */
void reverse(struct node **head_ref)
{
    struct node *temp = NULL;
    struct node *current = *head_ref;

    /* swap next and prev for all nodes of
       doubly linked list */
    while (current != NULL)
    {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    /* Before changing head, check for the cases like empty
       list and list with only one node */
    if(temp != NULL)
        *head_ref = temp->prev;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* Since we are adding node at the front
       of linked list, previous pointer of
       new node should point to NULL */
    new_node->prev = NULL;

    /* Next pointer of new node should point to
       old head */
    new_node->next = *head_ref;

    /* change head pointer to new node */
    *head_ref = new_node;
}
```

```

/* since we are adding at the begining,
   prev is always NULL */
new_node->prev = NULL;

/* link the old list off the new node */
new_node->next = (*head_ref);

/* change prev of head node to new node */
if((*head_ref) != NULL)
    (*head_ref)->prev = new_node ;

/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 10->8->4->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    printf("\n Original Linked list ");
    printList(head);

    /* Reverse doubly linked list */
    reverse(&head);

    printf("\n Reversed Linked list ");
    printList(head);

    getchar();
}

```

Time Complexity: O(n)

We can also swap data instead of pointers to reverse the Doubly Linked List. [Method used for reversing array](#) can be used to swap data. Swapping data can be costly compared to pointers if size of data item(s) is more.

The Great Tree-List Recursion Problem.

Asked by Varun Bhatia.

Question:

Write a recursive function `treeToList(Node root)` that takes an ordered binary tree and rearranges the internal pointers to make a circular doubly linked list out of the tree nodes. The previous pointers should be stored in the small field and the next pointers should be stored in the large field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list.

This is very well explained and implemented at <http://cslibrary.stanford.edu/109/TreeListRecursion.html>

QuickSort on Doubly Linked List

Following is a typical recursive implementation of [QuickSort](#) for arrays. The implementation uses last element as pivot.

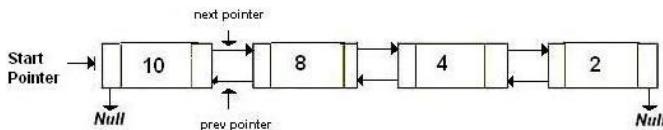
```
/* A typical recursive implementation of Quicksort for array*/
/* This function takes last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than
pivot) to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

Can we use same algorithm for Linked List?

Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is _quickSort() which is similar to quickSort() for array implementation.



```
// A C++ program to sort a linked list using Quicksort
#include <iostream>
#include <stdio.h>
using namespace std;

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{   int t = *a;      *a = *b;      *b = t; }

// A utility function to find last node of linked list
struct node *lastNode(node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than
pivot) to left of pivot and all greater elements to right of pivot */
```

```

node* partition(node *l, node *h)
{
    // set pivot as h element
    int x = h->data;

    // similar to i = l-1 for array implementation
    node *i = l->prev;

    // Similar to "for (int j = l; j <= h- 1; j++)"
    for (node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(struct node* l, struct node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        struct node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
void quickSort(struct node *head)
{
    // Find last node
    struct node *h = lastNode(head);

    // Call the recursive QuickSort
    _quickSort(head, h);
}

// A utility function to print contents of arr
void printList(struct node *head)
{
    while (head)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

/* Function to insert a node at the begining of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = new node; /* allocate node */
    new_node->data = new_data;

    /* since we are adding at the begining, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above function */
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
}

```

```

push(&a, 4);
push(&a, 3);
push(&a, 30);

cout << "Linked List before sorting \n";
printList(a);

quickSort(a);

cout << "Linked List after sorting \n";
printList(a);

return 0;
}

```

Output :

```

Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30

```

Time Complexity: Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes $O(n^2)$ time in worst case and $O(n\log n)$ in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?

Quicksort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

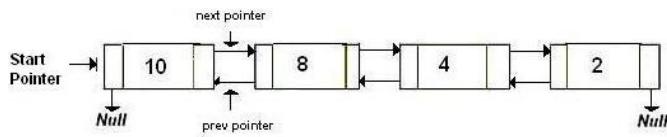
Exercise:

The above implementation is for doubly linked list. Modify it for singly linked list. Note that we dont have prev pointer in singly linked list. Refer [QuickSort on Singly Linked List](#) for solution.

Merge Sort for Doubly Linked List

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 2<->4<->8<->10



Below is C implementation of merge sort for doubly linked list.

```
// C program for merge sort on doubly linked list
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next, *prev;
};

struct node *split(struct node *head);

// Function to merge two linked lists
struct node *merge(struct node *first, struct node *second)
{
    // If first linked list is empty
    if (!first)
        return second;

    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data)
    {
        first->next = merge(first->next, second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else
    {
        second->next = merge(first, second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}

// Function to do merge sort
struct node *mergeSort(struct node *head)
{
    if (!head || !head->next)
        return head;
    struct node *second = split(head);

    // Recur for left and right halves
    head = mergeSort(head);
    second = mergeSort(second);

    // Merge the two sorted halves
    return merge(head, second);
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct node **head, int data)
{
    struct node *temp =
        (struct node *)malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
```

```

    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}

// A utility function to print a doubly linked list in
// both forward and backward directions
void print(struct node *head)
{
    struct node *temp = head;
    printf("Forward Traversal using next pointer\n");
    while (head)
    {
        printf("%d ", head->data);
        temp = head;
        head = head->next;
    }
    printf("\nBackward Traversal using prev pointer\n");
    while (temp)
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
}

// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
struct node *split(struct node *head)
{
    struct node *fast, *slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    struct node *temp = slow->next;
    slow->next = NULL;
    return temp;
}

// Driver program
int main(void)
{
    struct node *head = NULL;
    insert(&head, 5);
    insert(&head, 20);
    insert(&head, 4);
    insert(&head, 3);
    insert(&head, 30);
    insert(&head, 10);
    printf("Linked List before sorting\n");
    print(head);
    head = mergeSort(head);
    printf("\n\nLinked List after sorting\n");
    print(head);
    return 0;
}

```

Output:

```

Linked List before sorting
Forward Traversal using next pointer
10 30 3 4 20 5
Backward Traversal using prev pointer
5 20 4 3 30 10

Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3

```

Thanks to Goku for providing above implementation in a comment [here](#).

Time Complexity: Time complexity of the above implementation is same as time complexity of [MergeSort for arrays](#). It takes $\Theta(n \log n)$ time.

You may also like to see [QuickSort for doubly linked list](#)

Stack | Set 1 (Introduction)

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek: Get the topmost item

How to understand a stack practically?

There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Implementation:

There are two ways to implement a stack:

Using array

Using linked list

Using array:

C

```
// C program for array implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{   return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{   return stack->top == -1; }

// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[+stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}
```

```

// Function to get top item from stack
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    printf("Top item is %d\n", peek(stack));

    return 0;
}

```

Python

```

# Python program for implementation of stack

# import maxsize from sys module
# Used to return -infinite when stack is empty
from sys import maxsize

# Function to create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print("pushed to stack " + item)

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        return str(-maxsize -1) #return minus infinite

    return stack.pop()

# Function to get top item from stack
def peek(stack):
    if (isEmpty(stack)):
        return str(-maxsize -1)

    return stack[len(stack) - 1]

# Driver program to test above functions
stack = createStack()
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")
print("Top item is " + peek(stack))

```

Pros: Easy to implement. Memory is saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

```

10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top item is 20

```

Linked List Implementation:

```

// C program for linked list implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct StackNode
{
    int data;
    struct StackNode* next;
};

struct StackNode* newNode(int data)
{
    struct StackNode* stackNode =
        (struct StackNode*) malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(struct StackNode *root)
{
    return !root;
}

void push(struct StackNode** root, int data)
{
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}

int pop(struct StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    struct StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}

int peek(struct StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

int main()
{
    struct StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    printf("%d popped from stack\n", pop(&root));

    printf("Top element is %d\n", peek(root));

    return 0;
}

```

Output:

```

10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20

```

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

Applications of stack:

Balancing of symbols:

Infix to Postfix/Prefix conversion

Redo-undo features at many places like editors, photoshop.

Forward and backward feature in web browsers

Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem

Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver

We will cover the implementation of applications of stack in separate posts.

Quiz: [Stack Questions](#)

References:

http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29#Problem_Description

[GATE Corner](#)[Quiz Corner](#)

Stack | Set 2 (Infix to Postfix)

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is: abc*d++. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
 - ..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.
 - ..3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4. If the scanned character is an (, push it to the stack.
5. If the scanned character is an), pop and output from the stack until an (is encountered.
6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and output from the stack until it is not empty.

Following is C implementation of the above algorithm

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;

    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array)
        return NULL;
    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
```

```

        return '$';
    }
void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

// A utility function to check if the given character is operand
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// A utility function to return precedence of a given operator
// Higher returned value means higher precedence
int Prec(char ch)
{
    switch (ch)
    {
    case '+':
    case '-':
        return 1;

    case '*':
    case '/':
        return 2;

    case '^':
        return 3;
    }
    return -1;
}

// The main function that converts given infix expression
// to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    if(!stack) // See if stack was created successfully
        return -1;

    for (i = 0, k = -1; exp[i]; ++i)
    {
        // If the scanned character is an operand, add it to output.
        if (isOperand(exp[i]))
            exp[++k] = exp[i];

        // If the scanned character is an (, push it to the stack.
        else if (exp[i] == '(')
            push(stack, exp[i]);

        // If the scanned character is an ), pop and output from the stack
        // until an ( is encountered.
        else if (exp[i] == ')')
        {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1; // invalid expression
            else
                pop(stack);
        }
        else // an operator is encountered
        {
            while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }
    }

    // pop all the operators from the stack
    while (!isEmpty(stack))
        exp[++k] = pop(stack);

    exp[++k] = '\0';
}

```

```
    printf( "%s\n", exp );
}

// Driver program to test above functions
int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

Output:

abcd^e-fgh^{*}+^*+i-

[GATE Corner](#)[Quiz Corner](#)

Stack | Set 4 (Evaluation of Postfix Expression)

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed [infix to postfix conversion](#). In this post, evaluation of postfix expressions is discussed.

Following is algorithm for evaluation postfix expressions.

- 1) Create a stack to store operands (or values).
- 2) Scan the given expression and do following for every scanned element.
 - ..a) If the element is a number, push it into the stack
 - ..b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
- 3) When the expression is ended, the number in the stack is the final answer

Example:

Let the given expression be $2\ 3\ 1\ *\ +\ 9\ -$. We scan all elements one by one.

- 1) Scan 2, its a number, so push it to stack. Stack contains 2
- 2) Scan 3, again a number, push it to stack, stack now contains 2 3? (from bottom to top)
- 3) Scan 1, again a number, push it to stack, stack now contains 2 3 1?
- 4) Scan *, its an operator, pop two operands from stack, apply the * operator on operands, we get $3*1$ which results in 3. We push the result 3 to stack. Stack now becomes 2 3?
- 5) Scan +, its an operator, pop two operands from stack, apply the + operator on operands, we get $3 + 2$ which results in 5. We push the result 5 to stack. Stack now becomes 5.
- 6) Scan 9, its a number, we push it to the stack. Stack now becomes 5 9?.
- 7) Scan -, its an operator, pop two operands from stack, apply the operator on operands, we get $5 - 9$ which results in -4. We push the result -4? to stack. Stack now becomes -4?.
- 8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Following is C implementation of above algorithm.

```
// C program to evaluate value of a postfix expression
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}
```

```

void push(struct Stack* stack, char op)
{
    stack->array[+stack->top] = op;
}

// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand or number,
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
                case '+': push(stack, val2 + val1); break;
                case '-': push(stack, val2 - val1); break;
                case '*': push(stack, val2 * val1); break;
                case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "231*+9-";
    printf ("Value of %s is %d", exp, evaluatePostfix(exp));
    return 0;
}

```

Output:

Value of 231*+9- is -4

Time complexity of evaluation algorithm is O(n) where n is number of characters in input expression.

There are following limitations of above implementation.

- 1) It supports only 4 binary operators +, *, - and /. It can be extended for more operators by adding more switch cases.
- 2) The allowed operands are only single digit operands. The program can be extended for multiple digits by adding a separator like space between all elements (operators and operands) of given expression.

References:

<http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial2.pdf>

[GATE Corner](#)[Quiz Corner](#)

Stack | Set 3 (Reverse a string using stack)

Given a string, reverse it using stack. For example GeeksQuiz should be converted to ziuQskeeG.

Following is simple algorithm to reverse a string using stack.

- 1) Create an empty stack.
- 2) One by one push all characters of string to stack.
- 3) One by one pop all characters from stack and put them back to string.

Following are C and Python programs that implements above algorithm

C

```
// C program to reverse a string using stack
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    char* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (char*) malloc(stack->capacity * sizeof(char));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{   return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{   return stack->top == -1; }

// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, char item)
{
    if (isFull(stack))
        return;
    stack->array[stack->top] = item;
}

// Function to remove an item from stack. It decreases top by 1
char pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// A stack based function to reverese a string
void reverse(char str[])
{
    // Create a stack of capacity equal to length of string
    int n = strlen(str);
    struct Stack* stack = createStack(n);

    // Push all characters of string to stack
    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);

    // Pop all characters of string and put them back to str
    for (i = 0; i < n; i++)
        str[i] = pop(stack);
```

```

}

// Driver program to test above functions
int main()
{
    char str[] = "GeeksQuiz";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}

```

Python

```

# Python program to reverse a string using stack

# Function to create an empty stack. It initializes size of stack as 0
def createStack():
    stack=[]
    return stack

# Function to determine the size of the stack
def size(stack):
    return len(stack)

# Stack is empty if the size is 0
def isEmpty(stack):
    if size(stack) == 0:
        return true

# Function to add an item to stack . It increases size by 1
def push(stack,item):
    stack.append(item)

#Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if isEmpty(stack): return
    return stack.pop()

# A stack based function to reverse a string
def reverse(string):
    n = len(string)

    # Create a empty stack
    stack = createStack()

    # Push all characters of string to stack
    for i in range(0,n,1):
        push(stack,string[i])

    # Making the string empty since all characters are saved in stack
    string=""

    # Pop all characters of string and put them back to string
    for i in range(0,n,1):
        string+=pop(stack)

    return string

# Driver program to test above functions
string="GeeksQuiz"
string = reverse(string)
print("Reversed string is " + string)

# This code is contributed by Sunny Karira

```

Reversed string is ziQskeeG

Time Complexity: O(n) where n is number of characters in stack.

Auxiliary Space: O(n) for stack.

A string can also be reversed without using any auxiliary space. Following C and Python programs to implement reverse without using stack.

C

```

// C program to reverse a string without using stack
#include <stdio.h>
#include <string.h>

// A utility function to swap two characters
void swap(char *a, char *b)
{
    char temp = *a;
    *a = *b;
    *b = temp;
}

// A stack based function to reverese a string
void reverse(char str[])
{
    // get size of string
    int n = strlen(str), i;

    for (i = 0; i < n/2; i++)
        swap(&str[i], &str[n-i-1]);
}

// Driver program to test above functions
int main()
{
    char str[] = "abc";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}

```

Python

```

# Python program to reverse a string without stack

# Function to reverse a string
def reverse(string):
    string = string[::-1]
    return string

# Driver program to test above functions
string = "abc"
string = reverse(string)
print("Reversed string is " + string)

# This code is contributed by Sunny Karira

```

Reversed string is cba

[GATE Corner](#)[Quiz Corner](#)

Implement two stacks in an array

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

push1(int x) > pushes x to first stack

push2(int x) > pushes x to second stack

pop1() > pops an element from first stack and return the popped element

pop2() > pops an element from second stack and return the popped element

Implementation of *twoStack* should be space efficient.

Method 1 (Divide the space in two halves)

A simple way to implement two stacks is to divide the array in two halves and assign the half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[n/2+1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

C++

```
#include<iostream>
#include<stdlib.h>

using namespace std;

class twoStacks
{
    int *arr;
    int size;
    int top1, top2;
public:
    twoStacks(int n) // constructor
    {
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top1++;
            arr[top1] = x;
        }
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top2--;
            arr[top2] = x;
        }
        else
        {
```

```

        cout << "Stack Overflow";
        exit(1);
    }

// Method to pop an element from first stack
int pop1()
{
    if (top1 >= 0 )
    {
        int x = arr[top1];
        top1--;
        return x;
    }
    else
    {
        cout << "Stack UnderFlow";
        exit(1);
    }
}

// Method to pop an element from second stack
int pop2()
{
    if (top2 < size)
    {
        int x = arr[top2];
        top2++;
        return x;
    }
    else
    {
        cout << "Stack UnderFlow";
        exit(1);
    }
}
};

/* Driver program to test twStacks class */
int main()
{
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
    return 0;
}

```

Python

```

# Python Script to Implement two stacks in a list
class twoStacks:

    def __init__(self, n):      #constructor
        self.size = n
        self.arr = [None] * n
        self.top1 = -1
        self.top2 = self.size

    # Method to push an element x to stack1
    def push1(self, x):

        # There is at least one empty space for new element
        if self.top1 < self.top2 - 1 :
            self.top1 = self.top1 + 1
            self.arr[self.top1] = x

        else:
            print("Stack Overflow ")
            exit(1)

    # Method to push an element x to stack2
    def push2(self, x):

```

```

# There is at least one empty space for new element
if self.top1 < self.top2 - 1:
    self.top2 = self.top2 - 1
    self.arr[self.top2] = x

else :
    print("Stack Overflow ")
    exit(1)

# Method to pop an element from first stack
def pop1(self):
    if self.top1 >= 0:
        x = self.arr[self.top1]
        self.top1 = self.top1 -1
        return x
    else:
        print("Stack Underflow ")
        exit(1)

# Method to pop an element from second stack
def pop2(self):
    if self.top2 < self.size:
        x = self.arr[self.top2]
        self.top2 = self.top2 + 1
        return x
    else:
        print("Stack Underflow ")
        exit()

# Driver program to test twoStacks class
ts = twoStacks(5)
ts.push1(5)
ts.push2(10)
ts.push2(15)
ts.push1(11)
ts.push2(7)

print("Popped element from stack1 is " + str(ts.pop1()))
ts.push2(40)
print("Popped element from stack2 is " + str(ts.pop2()))

# This code is contributed by Sunny Karira

```

Popped element from stack1 is 11
Popped element from stack2 is 40

Time complexity of all 4 operations of *twoStack* is O(1).
We will extend to 3 stacks in an array in a separate post.

Check for balanced parentheses in an expression

Given an expression string exp, write a program to examine whether the pairs and the orders of {},(),[],[] are correct in exp. For example, the program should print true for exp = [0]{} {[00]0} and false for exp = [()

Algorithm:

- 1) Declare a character stack S.
- 2) Now traverse the expression string exp.
 - a) If the current character is a starting bracket ((or { or [) then push it to stack.
 - b) If the current character is a closing bracket () or } or]) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
- 3) After complete traversal, if there is some starting bracket left in stack then not balanced

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
    char data;
    struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Returns 1 if character1 and character2 are matching left
   and right Parenthesis */
bool isMatchingPair(char character1, char character2)
{
    if (character1 == '(' && character2 == ')')
        return 1;
    else if (character1 == '{' && character2 == '}')
        return 1;
    else if (character1 == '[' && character2 == ']')
        return 1;
    else
        return 0;
}

/*Return 1 if expression has balanced Parenthesis */
bool areParenthesisBalanced(char exp[])
{
    int i = 0;

    /* Declare an empty character stack */
    struct sNode *stack = NULL;

    /* Traverse the given expression to check matching parenthesis */
    while (exp[i])
    {
        /*If the exp[i] is a starting parenthesis then push it*/
        if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
            push(&stack, exp[i]);

        /* If exp[i] is a ending parenthesis then pop from stack and
           check if the popped parenthesis is a matching pair*/
        if (exp[i] == '}' || exp[i] == ')' || exp[i] == ']')
        {
            /*If we see an ending parenthesis without a pair then return false*/
            if (stack == NULL)
                return 0;

            /* Pop the top element from stack, if it is not a pair
               parenthesis of character then there is a mismatch.
               This happens for expressions like {{}} */
            else if ( !isMatchingPair(pop(&stack), exp[i]) )
                return 0;
        }
        i++;
    }
}
```

```

/* If there is something left in expression then there is a starting
 parenthesis without a closing parenthesis */
if (stack == NULL)
    return 1; /*balanced*/
else
    return 0; /*not balanced*/
}

/* UTILITY FUNCTIONS */
/*driver program to test above functions*/
int main()
{
    char exp[100] = "{()}{[]}";
    if (areParenthesisBalanced(exp))
        printf("\n Balanced ");
    else
        printf("\n Not Balanced ");
    return 0;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if (new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode *top;

    /*If stack is empty then error */
    if (*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

```

Time Complexity: O(n)

Auxiliary Space: O(n) for stack.

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

- a) For any array, rightmost element always has next greater element as -1.
- b) For an array which is sorted in decreasing order, all elements have next greater element as -1.
- c) For the input array {4, 5, 2, 25}, the next greater elements for each element are as follows.

| Element | NGE |
|---------|--------|
| 4 | --> 5 |
| 5 | --> 25 |
| 2 | --> 25 |
| 25 | --> -1 |

- d) For the input array {13, 7, 6, 12}, the next greater elements for each element are as follows.

| Element | NGE |
|---------|--------|
| 13 | --> -1 |
| 7 | --> 12 |
| 6 | --> 12 |
| 12 | --> -1 |

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Thanks to [Sachin](#) for providing following code.

C/C++

```
// Simple C program to print next greater elements
// in a given array
#include<stdio.h>

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -- %d\n", arr[i], next);
    }
}

int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Python

```
# Function to print element and NGE pair for all elements of list
def printNGE(arr):

    for i in range(0, len(arr), 1):

        next = -1
        for j in range(i+1, len(arr), 1):
```

```

if arr[i] < arr[j]:
    next = arr[j]
    break

print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function
arr = [11,13,21,3]
printNGE(arr)

# This code is contributed by Sunny Karira

```

11 -- 13
 13 -- 21
 21 -- -1
 3 -- -1

Time Complexity: O(n^2). The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

Thanks to [pcchild](#) for suggesting following approach.

- 1) Push the first element to stack.
- 2) Pick rest of the elements one by one and follow following steps in loop.
 - .a) Mark the current element as *next*.
 - .b) If stack is not empty, then pop an element from stack and compare it with *next*.
 - .c) If *next* is greater than the popped element, then *next* is the next greater element for the popped element.
 - .d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements
 - .g) If *next* is smaller than the popped element, then push the popped element back.
- 3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

Sort a stack using recursion

Given a stack, sort it using recursion. Use of any loop constructs like while, for..etc is not allowed. We can only use the following ADT functions on Stack S:

```
is_empty(S) : Tests whether stack is empty or not.  
push(S) : Adds new element to the stack.  
pop(S) : Removes top element from the stack.  
top(S) : Returns value of the top element. Note that this  
function does not remove element from the stack.
```

Example:

Input: -3 <-- Top
14
18
-5
30

Output: 30 <-- Top
18
14
-3
-5

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one in sorted order. Here sorted order is important.

Algorithm

We can use below algorithm to sort stack elements:

```
sortStack(stack S)  
if stack is not empty:  
    temp = pop(S);  
    sortStack(S);  
    sortedInsert(S, temp);
```

Below algorithm is to insert element in sorted order:

```
sortedInsert(Stack S, element)  
if stack is empty OR element > top element  
    push(S, elem)  
else  
    temp = pop(S)  
    sortedInsert(S, element)  
    push(S, temp)
```

Illustration:

Let given stack be
-3 <-- top of the stack
14
18
-5
30

Let us illustrate sorting of stack using above example:

First pop all the elements from the stack and store popped element in variable 'temp'. After popping all the elements function's stack frame will look like:

```
temp = -3 --> stack frame #1  
temp = 14 --> stack frame #2  
temp = 18 --> stack frame #3  
temp = -5 --> stack frame #4  
temp = 30 --> stack frame #5
```

Now stack is empty and 'insert_in_sorted_order()' function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

30 <-- top of the stack

Now next element i.e. -5 (from stack frame #4) is picked. Since $-5 < 30$, -5 is inserted at the bottom of stack. Now stack becomes:

```
30 <-- top of the stack  
-5
```

Next 18 (from stack frame #3) is picked. Since $18 < 30$, 18 is inserted below 30. Now stack becomes:

```
30 <-- top of the stack  
18  
14  
-5
```

Next 14 (from stack frame #2) is picked. Since $14 < 30$ and $14 < 18$, it is inserted below 18. Now stack becomes:

```
30 <-- top of the stack  
18  
14  
-3  
-5
```

Now -3 (from stack frame #1) is picked, as $-3 < 30$ and $-3 < 18$ and $-3 < 14$, it is inserted below 14. Now stack becomes:

```
30 <-- top of the stack  
18  
14  
-3  
-5
```

Implementation:

Below is C implementation of above algorithm.

```
// C program to sort a stack using recursion  
#include <stdio.h>  
#include <stdlib.h>  
  
// Stack is represented using linked list  
struct stack  
{  
    int data;  
    struct stack *next;  
};  
  
// Utility function to initialize stack  
void initStack(struct stack **s)  
{  
    *s = NULL;  
}  
  
// Utility function to check if stack is empty  
int isEmpty(struct stack *s)  
{  
    if (s == NULL)  
        return 1;  
    return 0;  
}  
  
// Utility function to push an item to stack  
void push(struct stack **s, int x)  
{  
    struct stack *p = (struct stack *)malloc(sizeof(*p));  
  
    if (p == NULL)  
    {  
        fprintf(stderr, "Memory allocation failed.\n");  
        return;  
    }  
  
    p->data = x;  
    p->next = *s;  
    *s = p;  
}  
  
// Utility function to remove an item from stack  
int pop(struct stack **s)  
{  
    int x;  
    struct stack *temp;  
  
    x = (*s)->data;  
    temp = *s;  
    (*s) = (*s)->next;  
    free(temp);
```

```

        return x;
    }

// Function to find top item
int top(struct stack *s)
{
    return (s->data);
}

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack **s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) || x > top(*s))
    {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
    sortedInsert(s, x);

    // Put back the top item removed earlier
    push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)
{
    // If stack is not empty
    if (!isEmpty(*s))
    {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility function to print contents of stack
void printStack(struct stack *s)
{
    while (s)
    {
        printf("%d ", s->data);
        s = s->next;
    }
    printf("\n");
}

// Driver Program
int main(void)
{
    struct stack *top;

    initStack(&top);
    push(&top, 30);
    push(&top, -5);
    push(&top, 18);
    push(&top, 14);
    push(&top, -3);

    printf("Stack elements before sorting:\n");
    printStack(top);

    sortStack(&top);
    printf("\n\n");

    printf("Stack elements after sorting:\n");
    printStack(top);

    return 0;
}

```

Output:

Stack elements before sorting:
-3 14 18 -5 30

Stack elements after sorting:
30 18 14 -3 -5

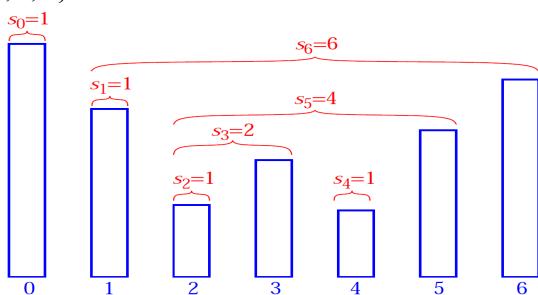
Exercise: Modify above code to reverse stack in descending order.

The Stock Span Problem

[The stock span problem](#) is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stocks price for all n days.

The span S_i of the stocks price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}



A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

C

```
// C program for brute force method to calculate stock span values
#include <stdio.h>

// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1;

    // Calculate span value of remaining days by linearly checking
    // previous days
    for (int i = 1; i < n; i++)
    {
        S[i] = 1; // Initialize span value

        // Traverse left while the next element on left is smaller
        // than price[i]
        for (int j = i-1; (j>=0)&&(price[i]>=price[j]); j--)
            S[i]++;
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}
```

Python

```

# Python program for brute force method to calculate stock span values

# Fills list S[] with span values
def calculateSpan(price, n, S):

    # Span value of first day is always 1
    S[0] = 1

    # Calculate span value of remaining days by linearly
    # checking previous days
    for i in range(1, n, 1):
        S[i] = 1    # Initialize span value

        # Traverse left while the next element on left is
        # smaller than price[i]
        j = i - 1
        while (j >= 0) and (price[i] >= price[j]) :
            S[i] += 1
            j -= 1

    # A utility function to print elements of array
    def printArray(arr, n):

        for i in range(n):
            print(arr[i], end = " ")

    # Driver program to test above function
    price = [10, 4, 5, 90, 120, 80]
    n = len(price)
    S = [None] * n

    # Fill the span values in list S[]
    calculateSpan(price, n, S)

    # print the calculated span values
    printArray(S, n)

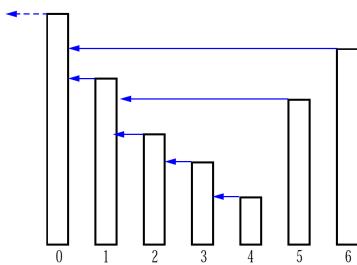
# This code is contributed by Sunny Karira

```

A Linear Time Complexity Method

We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on day i . If such a day exists, lets call it $h(i)$, otherwise, we define $h(i) = -1$.

The span is now computed as $S[i] = i - h(i)$. See the following diagram.



To implement this logic, we use a stack as an abstract data type to store the days i , $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to i , we pop the days when the price of the stock was less than or equal to $price[i]$ and then push the value of day i back into the stack.

Following is C++ implementation of this method.

```

// a linear time solution for stock span problem
#include <iostream>
#include <stack>
using namespace std;

// A stack based efficient method to calculate stock span values
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first element to it
    stack<int> st;
    st.push(0);

    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements

```

```

for (int i = 1; i < n; i++)
{
    // Pop elements from stack while stack is not empty and top of
    // stack is smaller than price[i]
    while (!st.empty() && price[st.top()] <= price[i])
        st.pop();

    // If stack becomes empty, then price[i] is greater than all elements
    // on left of it, i.e., price[0], price[1],..price[i-1]. Else price[i]
    // is greater than elements after top of stack
    S[i] = (st.empty())? (i + 1) : (i - st.top());

    // Push this element to stack
    st.push(i);
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}

```

Output:

1 1 2 4 5 1

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once. So there are total $2n$ operations at most. Assuming that a stack operation takes O(1) time, we can say that the time complexity is O(n).

Auxiliary Space: O(n) in worst case when all elements are sorted in decreasing order.

References:

[http://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)#The_Stock_Span_Problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#The_Stock_Span_Problem)
<http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/Stack-I.pdf>

Design and Implement Special Stack Data Structure | Added Space Optimized Version

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

```
Consider the following SpecialStack
16 --> TOP
15
29
19
18
```

When getMin() is called it should return 15, which is the minimum element in the current stack.

If we do pop two times on stack, the stack becomes
29 --> TOP
19
18

When getMin() is called, it should return 18 which is the minimum in the current stack.

Solution: Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

Push(int x) // inserts an element x to Special Stack

- 1) push x to the first stack (the stack with actual elements)
- 2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
 - ..a) If x is smaller than y then push x to the auxiliary stack.
 - ..b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

- 1) Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.
Actual Stack
18 <--- top
Auxiliary Stack
18 <---- top

When 19 is inserted, both stacks change to following.
Actual Stack
19 <--- top
18
Auxiliary Stack
18 <---- top
18

When 29 is inserted, both stacks change to following.
Actual Stack
29 <--- top
19
18
Auxiliary Stack
18 <---- top
18
18

```

When 15 is inserted, both stacks change to following.
Actual Stack
15 <--- top
29
19
18
Auxiliary Stack
15 <---- top
18
18
18

```

```

When 16 is inserted, both stacks change to following.
Actual Stack
16 <--- top
15
29
19
18
Auxiliary Stack
15 <---- top
15
18
18
18

```

Following is C++ implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object *min* which work as auxiliary stack.

```

#include<iostream>
#include<stdlib.h>

using namespace std;

/* A simple stack class with basic stack functionalities */
class Stack
{
private:
    static const int max = 100;
    int arr[max];
    int top;
public:
    Stack() { top = -1; }
    bool isEmpty();
    bool isFull();
    int pop();
    void push(int x);
};

/* Stack's member method to check if the stack is iempty */
bool Stack::isEmpty()
{
    if(top == -1)
        return true;
    return false;
}

/* Stack's member method to check if the stack is full */
bool Stack::isFull()
{
    if(top == max - 1)
        return true;
    return false;
}

/* Stack's member method to remove an element from it */
int Stack::pop()
{
    if(isEmpty())
    {
        cout<<"Stack Underflow";
        abort();
    }
    int x = arr[top];
    top--;
    return x;
}

/* Stack's member method to insert an element to it */
void Stack::push(int x)
{

```

```

if(isFull())
{
    cout<<"Stack Overflow";
    abort();
}
top++;
arr[top] = x;
}

/* A class that supports all the stack operations and one additional
operation getMin() that returns the minimum element from stack at
any time. This class inherits from the stack class and uses an
auxiliarry stack that holds minimum elements */
class SpecialStack: public Stack
{
    Stack min;
public:
    int pop();
    void push(int x);
    int getMin();
};

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);
        if( x < y )
            min.push(x);
        else
            min.push(y);
    }
}

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    min.pop();
    return x;
}

/* SpecialStack's member method to get minimum element from it. */
int SpecialStack::getMin()
{
    int x = min.pop();
    min.push(x);
    return x;
}

/* Driver program to test SpecialStack methods */
int main()
{
    SpecialStack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.getMin()<<endl;
    s.push(5);
    cout<<s.getMin();
    return 0;
}

```

Output:

10
5

Space Optimized Version

The above approach can be optimized. We can limit the number of elements in auxiliary stack. We can push only when the incoming element of

main stack is smaller than or equal to top of auxiliary stack. Similarly during pop, if the pop off element equal to top of auxiliary stack, remove the top element of auxiliary stack. Following is modified implementation of push() and pop().

```
/* SpecialStack's member method to insert an element to it. This method
   makes sure that the min stack is also updated with appropriate minimum
   values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);

        /* push only when the incoming element of main stack is smaller
           than or equal to top of auxiliary stack */
        if( x <= y )
            min.push(x);
    }
}

/* SpecialStack's member method to remove an element from it. This method
   removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    int y = min.pop();

    /* Push the popped element y back only if it is not equal to x */
    if ( y != x )
        min.push(y);

    return x;
}
```

Thanks to [@Venki](#), [@swarup](#) and [@Jing Huang](#) for their inputs.

Implement Stack using Queues

The problem is opposite of [this](#) post. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue.

A stack can be implemented using two queues. Let stack to be implemented be s and queues used to implement be q1? and q2?. Stack s can be implemented in two ways:

Method 1 (By making push operation costly)

This method makes sure that newly entered element is always at the front of q1?, so that pop operation just dequeues from q1?. q2? is used to put every new element at front of q1?.

```
push(s, x) // x is the element to be pushed and s is stack
1) Enqueue x to q2
2) One by one dequeue everything from q1 and enqueue to q2.
3) Swap the names of q1 and q2
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

pop(s)
1) Dequeue an item from q1 and return it.
```

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

```
push(s, x)
1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)
1) One by one dequeue everything except the last element from q1 and enqueue to q2.
2) Dequeue the last item of q1, the dequeued item is result, store it.
3) Swap the names of q1 and q2
4) Return the item stored in step 2.
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.
```

References:

[Implement Stack using Two Queues](#)

Design a stack with operations on middle element

How to implement a stack which will support following operations in **O(1) time complexity**?

- 1) push() which adds an element to the top of stack.
 - 2) pop() which removes an element from top of stack.
 - 3) findMiddle() which will return middle element of the stack.
 - 4) deleteMiddle() which will delete the middle element.
- Push and pop are standard stack operations.

The important question is, whether to use a linked list or array for implementation of stack?

Please note that, we need to find and delete middle element. Deleting an element from middle is not O(1) for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.

The idea is to use Doubly Linked List (DLL). We can delete middle element in O(1) time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.

Following is C implementation of push(), pop() and findMiddle() operations. Implementation of deleteMiddle() is left as an exercise. If there are even elements in stack, findMiddle() returns the first middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 2.

```
/* Program to implement a stack that supports findMiddle() and deleteMiddle
   in O(1) time */
#include <stdio.h>
#include <stdlib.h>

/* A Doubly Linked List Node */
struct DLLNode
{
    struct DLLNode *prev;
    int data;
    struct DLLNode *next;
};

/* Representation of the stack data structure that supports findMiddle()
   in O(1) time. The Stack is implemented using Doubly Linked List. It
   maintains pointer to head node, pointer to middle node and count of
   nodes */
struct myStack
{
    struct DLLNode *head;
    struct DLLNode *mid;
    int count;
};

/* Function to create the stack data structure */
struct myStack *createMyStack()
{
    struct myStack *ms =
        (struct myStack*) malloc(sizeof(struct myStack));
    ms->count = 0;
    return ms;
};

/* Function to push an element to the stack */
void push(struct myStack *ms, int new_data)
{
    /* allocate DLLNode and put in data */
    struct DLLNode* new_DLLNode =
        (struct DLLNode*) malloc(sizeof(struct DLLNode));
    new_DLLNode->data = new_data;

    /* Since we are adding at the begining,
       prev is always NULL */
    new_DLLNode->prev = NULL;

    /* link the old list off the new DLLNode */
    new_DLLNode->next = ms->head;

    /* Increment count of items in stack */
    ms->count += 1;

    /* Change mid pointer in two cases
       1) Linked List is empty
       2) Number of nodes in linked list is odd */
    if (ms->count == 1)
    {
        ms->mid = new_DLLNode;
    }
    else if (ms->count % 2 == 0)
    {
        ms->mid = ms->mid->next;
    }
}
```

```

        ms->mid = new_DLLNode;
    }
else
{
    ms->head->prev = new_DLLNode;

    if (ms->count & 1) // Update mid if ms->count is odd
        ms->mid = ms->mid->prev;
}

/* move head to point to the new DLLNode */
ms->head = new_DLLNode;
}

/* Function to pop an element from stack */
int pop(struct myStack *ms)
{
    /* Stack underflow */
    if (ms->count == 0)
    {
        printf("Stack is empty\n");
        return -1;
    }

    struct DLLNode *head = ms->head;
    int item = head->data;
    ms->head = head->next;

    // If linked list doesn't become empty, update prev
    // of new head as NULL
    if (ms->head != NULL)
        ms->head->prev = NULL;

    ms->count -= 1;

    // update the mid pointer when we have even number of
    // elements in the stack, i.e move down the mid pointer.
    if (!((ms->count) & 1))
        ms->mid = ms->mid->next;

    free(head);

    return item;
}

// Function for finding middle of the stack
int findMiddle(struct myStack *ms)
{
    if (ms->count == 0)
    {
        printf("Stack is empty now\n");
        return -1;
    }

    return ms->mid->data;
}

// Driver program to test functions of myStack
int main()
{
    /* Let us create a stack using push() operation*/
    struct myStack *ms = createMyStack();
    push(ms, 11);
    push(ms, 22);
    push(ms, 33);
    push(ms, 44);
    push(ms, 55);
    push(ms, 66);
    push(ms, 77);

    printf("Item popped is %d\n", pop(ms));
    printf("Item popped is %d\n", pop(ms));
    printf("Middle Element is %d\n", findMiddle(ms));
    return 0;
}

```

Output:

```

Item popped is 77
Item popped is 66
Middle Element is 33

```


How to efficiently implement k stacks in a single array?

We have discussed [space efficient implementation of 2 stacks in a single array](#). In this post, a general solution for k stacks is discussed. Following is the detailed problem statement.

Create a data structure `kStacks` that represents k stacks. Implementation of `kStacks` should use only one array, i.e., k stacks should use the same array for storing elements. Following functions must be supported by `kStacks`.

`push(int x, int sn) > pushes x to stack number sn where sn is from 0 to k-1`
`pop(int sn) > pops an element from stack number sn where sn is from 0 to k-1`

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use `arr[0]` to `arr[n/k-1]` for first stack, and `arr[n/k]` to `arr[2n/k-1]` for stack2 where `arr[]` is the array to be used to implement two stacks and size of array be n .

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in `arr[]`. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer* arrays as extra space.

Following are the two extra arrays are used:

- 1) `top[]`: This is of size k and stores indexes of top elements in all stacks.
- 2) `next[]`: This is of size n and stores indexes of next item for the items in array `arr[]`. Here `arr[]` is actual array that stores k stacks. Together with k stacks, a stack of free slots in `arr[]` is also maintained. The top of this stack is stored in a variable `free`.

All entries in `top[]` are initialized as -1 to indicate that all stacks are empty. All entries `next[i]` are initialized as $i+1$ because all slots are free initially and pointing to next slot. Top of free stack, `free` is initialized as 0.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate implementation of k stacks in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k stacks in a single array of size n
class kStacks
{
    int *arr; // Array of size n to store actual content to be stored in stacks
    int *top; // Array of size k to store indexes of top elements of stacks
    int *next; // Array of size n to store next entry in all stacks
               // and free list
    int n, k;
    int free; // To store beginning index of free list
public:
    //constructor to create k stacks in an array of size n
    kStacks(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To push an item in stack number 'sn' where sn is from 0 to k-1
    void push(int item, int sn);

    // To pop an from stack number 'sn' where sn is from 0 to k-1
    int pop(int sn);

    // To check whether stack number 'sn' is empty or not
    bool isEmpty(int sn) { return (top[sn] == -1); }
};

//constructor to create k stacks in an array of size n
kStacks::kStacks(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    top = new int[k];
    next = new int[n];
}
```

```

// Initialize all stacks as empty
for (int i = 0; i < k; i++)
    top[i] = -1;

// Initialize all spaces as free
free = 0;
for (int i=0; i<n-1; i++)
    next[i] = i+1;
next[n-1] = -1; // -1 is used to indicate end of free list
}

// To push an item in stack number 'sn' where sn is from 0 to k-1
void kStacks::push(int item, int sn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nStack Overflow\n";
        return;
    }

    int i = free;      // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    // Update next of top and then top for stack number 'sn'
    next[i] = top[sn];
    top[sn] = i;

    // Put the item in array
    arr[i] = item;
}

// To pop an from stack number 'sn' where sn is from 0 to k-1
int kStacks::pop(int sn)
{
    // Underflow check
    if (isEmpty(sn))
    {
        cout << "\nStack Underflow\n";
        return INT_MAX;
    }

    // Find index of top item in stack number 'sn'
    int i = top[sn];

    top[sn] = next[i]; // Change top to store next of previous top

    // Attach the previous top to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous top item
    return arr[i];
}

/* Driver program to test twStacks class */
int main()
{
    // Let us create 3 stacks in an array of size 10
    int k = 3, n = 10;
    kStacks ks(k, n);

    // Let us put some items in stack number 2
    ks.push(15, 2);
    ks.push(45, 2);

    // Let us put some items in stack number 1
    ks.push(17, 1);
    ks.push(49, 1);
    ks.push(39, 1);

    // Let us put some items in stack number 0
    ks.push(11, 0);
    ks.push(9, 0);
    ks.push(7, 0);

    cout << "Popped element from stack 2 is " << ks.pop(2) << endl;
    cout << "Popped element from stack 1 is " << ks.pop(1) << endl;
}

```

```
cout << "Popped element from stack 0 is " << ks.pop(0) << endl;
return 0;
}
```

Output:

```
Popped element from stack 2 is 45
Popped element from stack 1 is 39
Popped element from stack 0 is 7
```

Time complexities of operations push() and pop() is O(1).

The best part of above implementation is, if there is a slot available in stack, then an item can be pushed in any of the stacks, i.e., no wastage of space.

Sort a stack using recursion

Given a stack, sort it using recursion. Use of any loop constructs like while, for..etc is not allowed. We can only use the following ADT functions on Stack S:

```
is_empty(S) : Tests whether stack is empty or not.  
push(S) : Adds new element to the stack.  
pop(S) : Removes top element from the stack.  
top(S) : Returns value of the top element. Note that this  
function does not remove element from the stack.
```

Example:

Input: -3 <-- Top
14
18
-5
30

Output: 30 <-- Top
18
14
-3
-5

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one in sorted order. Here sorted order is important.

Algorithm

We can use below algorithm to sort stack elements:

```
sortStack(stack S)  
if stack is not empty:  
    temp = pop(S);  
    sortStack(S);  
    sortedInsert(S, temp);
```

Below algorithm is to insert element in sorted order:

```
sortedInsert(Stack S, element)  
if stack is empty OR element > top element  
    push(S, elem)  
else  
    temp = pop(S)  
    sortedInsert(S, element)  
    push(S, temp)
```

Illustration:

Let given stack be
-3 <-- top of the stack
14
18
-5
30

Let us illustrate sorting of stack using above example:

First pop all the elements from the stack and store popped element in variable 'temp'. After popping all the elements function's stack frame will look like:

```
temp = -3 --> stack frame #1  
temp = 14 --> stack frame #2  
temp = 18 --> stack frame #3  
temp = -5 --> stack frame #4  
temp = 30 --> stack frame #5
```

Now stack is empty and 'insert_in_sorted_order()' function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

30 <-- top of the stack

Now next element i.e. -5 (from stack frame #4) is picked. Since $-5 < 30$, -5 is inserted at the bottom of stack. Now stack becomes:

```
30 <-- top of the stack  
-5
```

Next 18 (from stack frame #3) is picked. Since $18 < 30$, 18 is inserted below 30. Now stack becomes:

```
30 <-- top of the stack  
18  
14  
-5
```

Next 14 (from stack frame #2) is picked. Since $14 < 30$ and $14 < 18$, it is inserted below 18. Now stack becomes:

```
30 <-- top of the stack  
18  
14  
-3  
-5
```

Now -3 (from stack frame #1) is picked, as $-3 < 30$ and $-3 < 18$ and $-3 < 14$, it is inserted below 14. Now stack becomes:

```
30 <-- top of the stack  
18  
14  
-3  
-5
```

Implementation:

Below is C implementation of above algorithm.

```
// C program to sort a stack using recursion  
#include <stdio.h>  
#include <stdlib.h>  
  
// Stack is represented using linked list  
struct stack  
{  
    int data;  
    struct stack *next;  
};  
  
// Utility function to initialize stack  
void initStack(struct stack **s)  
{  
    *s = NULL;  
}  
  
// Utility function to check if stack is empty  
int isEmpty(struct stack *s)  
{  
    if (s == NULL)  
        return 1;  
    return 0;  
}  
  
// Utility function to push an item to stack  
void push(struct stack **s, int x)  
{  
    struct stack *p = (struct stack *)malloc(sizeof(*p));  
  
    if (p == NULL)  
    {  
        fprintf(stderr, "Memory allocation failed.\n");  
        return;  
    }  
  
    p->data = x;  
    p->next = *s;  
    *s = p;  
}  
  
// Utility function to remove an item from stack  
int pop(struct stack **s)  
{  
    int x;  
    struct stack *temp;  
  
    x = (*s)->data;  
    temp = *s;  
    (*s) = (*s)->next;  
    free(temp);
```

```

        return x;
    }

// Function to find top item
int top(struct stack *s)
{
    return (s->data);
}

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack **s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) || x > top(*s))
    {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
    sortedInsert(s, x);

    // Put back the top item removed earlier
    push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)
{
    // If stack is not empty
    if (!isEmpty(*s))
    {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility function to print contents of stack
void printStack(struct stack *s)
{
    while (s)
    {
        printf("%d ", s->data);
        s = s->next;
    }
    printf("\n");
}

// Driver Program
int main(void)
{
    struct stack *top;

    initStack(&top);
    push(&top, 30);
    push(&top, -5);
    push(&top, 18);
    push(&top, 14);
    push(&top, -3);

    printf("Stack elements before sorting:\n");
    printStack(top);

    sortStack(&top);
    printf("\n\n");

    printf("Stack elements after sorting:\n");
    printStack(top);

    return 0;
}

```

Output:

Stack elements before sorting:
-3 14 18 -5 30

Stack elements after sorting:
30 18 14 -3 -5

Exercise: Modify above code to reverse stack in descending order.

Queue | Set 1 (Introduction and Array Implementation)

Like [Stack](#), [Queue](#) is a linear structure which follows a particular order in which the operations are performed. The order is **First In First Out** (FIFO).

A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:

Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.

Applications of Queue:

Queue is used when things don't have to be processed immediately, but have to be processed in **First In First Out** order like [Breadth First Search](#). This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See [this](#) for more detailed applications of Queue and Stack.

Array implementation Of Queue

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner (See [this](#) for details)

```
// C program for array implementation of queue
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a queue
struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int* array;
};

// function to create a queue of given capacity. It initializes size of
// queue as 0
struct Queue* createQueue(unsigned capacity)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1; // This is important, see the enqueue
    queue->array = (int*) malloc(queue->capacity * sizeof(int));
    return queue;
}

// Queue is full when size becomes equal to the capacity
int isFull(struct Queue* queue)
{ return (queue->size == queue->capacity); }

// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{ return (queue->size == 0); }

// Function to add an item to the queue. It changes rear and size
void enqueue(struct Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1)%queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}
```

```

// Function to remove an item from queue. It changes front and size
int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

// Function to get front of queue
int front(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}

// Driver program to test above functions./
int main()
{
    struct Queue* queue = createQueue(1000);

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    printf("%d dequeued from queue\n", dequeue(queue));

    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}

```

Output:

```

10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue
Front item is 20
Rear item is 40

```

Time Complexity: Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

Linked list implementation is easier, we will soon be discussing linked list implementation in next article.

Queue | Set 2 (Linked List Implementation)

In the [previous post](#), we introduced Queue and discussed array implementation. In this post, linked list implementation is discussed. The following two main operations must be implemented efficiently.

In a Queue data structure, we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.

enQueue() This operation adds a new node after *rear* and moves *rear* to the next node.

deQueue() This operation removes the front node and moves *front* to the next node.

```
// A C program to demonstrate linked list based implementation of queue
#include <stdlib.h>
#include <stdio.h>

// A linked list (LL) node to store a queue entry
struct QNode
{
    int key;
    struct QNode *next;
};

// The queue, front stores the front node of LL and rear stores the
// last node of LL
struct Queue
{
    struct QNode *front, *rear;
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
    struct QNode *temp = (struct QNode*)malloc(sizeof(struct QNode));
    temp->key = k;
    temp->next = NULL;
    return temp;
}

// A utility function to create an empty queue
struct Queue *createQueue()
{
    struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}

// The function to add a key k to q
void enQueue(struct Queue *q, int k)
{
    // Create a new LL node
    struct QNode *temp = newNode(k);

    // If queue is empty, then new node is front and rear both
    if (q->rear == NULL)
    {
        q->front = q->rear = temp;
        return;
    }

    // Add the new node at the end of queue and change rear
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove a key from given queue q
struct QNode *deQueue(struct Queue *q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
        return NULL;

    // Store previous front and move front one node ahead
    struct QNode *temp = q->front;
    q->front = q->front->next;

    // If front becomes NULL, then change rear also as NULL
    if (q->front == NULL)
        q->rear = NULL;
    return temp;
}
```

```
}
```

```
// Driver Program to test above functions
```

```
int main()
```

```
{
```

```
    struct Queue *q = createQueue();
```

```
    enQueue(q, 10);
```

```
    enQueue(q, 20);
```

```
    deQueue(q);
```

```
    deQueue(q);
```

```
    enQueue(q, 30);
```

```
    enQueue(q, 40);
```

```
    enQueue(q, 50);
```

```
    struct QNode *n = deQueue(q);
```

```
    if (n != NULL)
```

```
        printf("Dequeued item is %d", n->key);
```

```
    return 0;
```

```
}
```

Output:

```
Dequeued item is 30
```

Time Complexity: Time complexity of both operations enqueue() and dequeue() is O(1) as we only change few pointers in both operations. There is no loop in any of the operations.

[GATE Corner](#)[Quiz Corner](#)

Applications of Queue Data Structure

[Queue](#) is used when things don't have to be processed immediately, but have to be processed in **First In First Out** order like [Breadth First Search](#). This property of Queue makes it also useful in following kind of scenarios.

- 1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- 2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See [this](#) for more detailed applications of Queue and Stack.

References:

<http://introcs.cs.princeton.edu/43stack/>

Priority Queue | Set 1 (Introduction)

Priority Queue is an extension of [queue](#) with following properties.

- 1) Every item has a priority associated with it.
- 2) An element with high priority is dequeued before an element with low priority.
- 3) If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.

getHighestPriority(): Returns the highest priority item

deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?

Using Array: A simple implementation is to use array of following structure.

```
struct item {  
    int item;  
    int priority;  
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we dont have to move items.

Using Heaps:

Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, getHighestPriority() can be implemented in O(1) time, insert() can be implemented in O(Logn) time and deleteHighestPriority() can also be implemented in O(Logn) time.

With [Fibonacci heap](#), insert() and getHighestPriority() can be implemented in O(1) amortized time and deleteHighestPriority() can be implemented in O(Logn) amortized time.

Applications of Priority Queue:

- 1) CPU Scheduling
- 2) Graph algorithms like [Dijkstras shortest path algorithm](#), [Prims Minimum Spanning Tree](#), etc
- 3) All [queue applications](#) where priority is involved.

We will soon be discussing array and heap implementations of priority queue.

References:

http://en.wikipedia.org/wiki/Priority_queue

Deque | Set 1 (Introduction and Applications)

[Deque or Double Ended Queue](#) is a generalized version of [Queue data structure](#) that allows insert and delete at both ends.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.

insertLast(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.

isEmpty(): Checks whether Deque is empty or not.

isFull(): Checks whether Deque is full or not.

Applications of Deque:

Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque. For example see

[Maximum of all subarrays of size k problem](#).

See [wiki page](#) for another example of A-Steal job scheduling algorithm where Deque is used as deletions operation is required at both ends.

Language Support:

C++ STL provides implementation of Deque as [std::deque](#) and Java provides [Deque interface](#). See [this](#) for more details.

Implementation:

A Deque can be implemented either using a [doubly linked list](#) or circular array. In both implementation, we can implement all operations in O(1) time. We will soon be discussing C/C++ implementation of Deque Data structure.

Implement Queue using Stacks

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly)

This method makes sure that newly entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

```
enQueue(q, x)
1) While stack1 is not empty, push everything from stack1 to stack2.
2) Push x to stack1 (assuming size of stacks is unlimited).
3) Push everything back to stack1.

deQueue(q)
1) If stack1 is empty then error
2) Pop an item from stack1 and return it
```

Method 2 (By making deQueue operation costly)

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enQueue(q, x)
1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)
1) If both stacks are empty then error.
2) If stack2 is empty
   While stack1 is not empty, push everything from stack1 to stack2.
3) Pop the element from stack2 and return it.
```

Method 2 is definitely better than method 1. Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.

Implementation of method 2:

```
/* Program to implement a queue using two stacks */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
    int data;
    struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* structure of queue having two stacks */
struct queue
{
    struct sNode *stack1;
    struct sNode *stack2;
};

/* Function to enqueue an item to queue */
void enQueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int deQueue(struct queue *q)
{
    int x;

    /* If both stacks are empty then error */
    if(q->stack1 == NULL && q->stack2 == NULL)
    {
        printf("Q is empty");
        getchar();
        exit(0);
    }
}
```

```

/* Move elements from satck1 to stack 2 only if
   stack2 is empty */
if(q->stack2 == NULL)
{
    while(q->stack1 != NULL)
    {
        x = pop(&q->stack1);
        push(&q->stack2, x);
    }
}

x = pop(&q->stack2);
return x;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode *top;

    /*If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test anove functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;
    q->stack2 = NULL;
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    getchar();
}

```

Queue can also be implemented using one user stack and one Function Call Stack.

Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```
enQueue(x)
1) Push x to stack1.

deQueue:
1) If stack1 is empty then error.
2) If stack1 has only one element then return it.
3) Recursively pop everything from the stack1, store the popped item
   in a variable res, push the res back to stack1 and return res
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in *dequeue()* and all other items are pushed back in step 3.

Implementation of method 2 using Function Call Stack:

```
/* Program to implement a queue using one user defined stack and one Function Call Stack */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
    int data;
    struct sNode *next;
};

/* structure of queue having two stacks */
struct queue
{
    struct sNode *stack1;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Function to enqueue an item to queue */
void enQueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int deQueue(struct queue *q)
{
    int x, res;

    /* If both stacks are empty then error */
    if(q->stack1 == NULL)
    {
        printf("Q is empty");
        getchar();
        exit(0);
    }
    else if(q->stack1->next == NULL)
    {
        return pop(&q->stack1);
    }
    else
    {
        /* pop an item from the stack1 */
        x = pop(&q->stack1);

        /* store the last dequeued item */
        res = deQueue(q);

        /* push everything back to stack1 */
        push(&q->stack1, x);
    }
    return res;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
```

```

/* allocate node */
struct sNode* new_node =
    (struct sNode*) malloc(sizeof(struct sNode));

if(new_node == NULL)
{
    printf("Stack overflow \n");
    getchar();
    exit(0);
}

/* put in the data */
new_node->data = new_data;

/* link the old list off the new node */
new_node->next = (*top_ref);

/* move the head to point to the new node */
(*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode *top;

    /*If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;

    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    getchar();
}

```

Find the first circular tour that visits all petrol pumps

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be start = 1? (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).

We can **use a Queue** to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeuing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

```
// C program to find circular tour for a truck
#include <stdio.h>

// A petrol pump has petrol and distance to next petrol pump
struct petrolPump
{
    int petrol;
    int distance;
};

// The function returns starting point if there is a possible solution,
// otherwise returns -1
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end = 1;

    int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
     And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If current amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
            start = (start + 1)%n;

            // If 0 is being considered as start again, then there is no
            // possible solution
            if (start == 0)
                return -1;
        }

        // Add a petrol pump to current tour
        curr_petrol += arr[end].petrol - arr[end].distance;

        end = (end + 1)%n;
    }

    // Return starting point
    return start;
}

// Driver program to test above functions
int main()
{
    struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
int start = printTour(arr, n);

(start == -1)? printf("No solution"): printf("Start = %d", start);

return 0;
}
```

Output:

```
start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is $O(n)$.

Auxiliary Space: $O(1)$

Maximum of all subarrays of size k (Added a O(n) method)

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Examples:

Input :

arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}

k = 3

Output :

3 3 4 5 5 6

Input :

arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}

k = 4

Output :

10 10 10 15 15 90 90

Method 1 (Simple)

Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.

```
#include<stdio.h>

void printKMax(int arr[], int n, int k)
{
    int j, max;

    for (int i = 0; i <= n-k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i+j] > max)
                max = arr[i+j];
        }
        printf("%d ", max);
    }
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Time Complexity: The outer loop runs $n-k+1$ times and the inner loop runs k times for every iteration of outer loop. So time complexity is $O((n-k+1)*k)$ which can also be written as $O(nk)$.

Method 2 (Use Self-Balancing BST)

1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.

2) Run a loop for $i = 0$ to $n-k$

..a) Get the maximum element from the BST, and print it.

..b) Search for $arr[i]$ in the BST and delete it from the BST.

..c) Insert $arr[i+k]$ into the BST.

Time Complexity: Time Complexity of step 1 is $O(k\log k)$. Time Complexity of steps 2(a), 2(b) and 2(c) is $O(\log k)$. Since steps 2(a), 2(b) and 2(c) are in a loop that runs $n-k+1$ times, time complexity of the complete algorithm is $O(k\log k + (n-k+1)*\log k)$ which can also be written as $O(n\log k)$.

Method 3 (A O(n) method: use Dequeue)

We create a [Dequeue](#), Qi of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain Qi to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the Qi is the largest and

element at rear of Q_i is the smallest of current window. Thanks to [Aashish](#) for suggesting this method.

Following is C++ implementation of this method.

```
#include <iostream>
#include <deque>

using namespace std;

// A Dequeue (Double ended queue) based method for printing maximum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front()] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int> Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
    {
        // For every element, the previous smaller elements are useless so
        // remove them from Qi
        while (!Qi.empty() && arr[i] >= arr[Qi.back()])
            Qi.pop_back(); // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while (!Qi.empty() && Qi.front() <= i - k)
            Qi.pop_front(); // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while (!Qi.empty() && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

        // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, 1, 78, 90, 57, 89, 56};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Output:

```
78 90 90 90 89
```

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed at most once. So there are total $2n$ operations.

Auxiliary Space: $O(k)$

An Interesting Method to Generate Binary Numbers from 1 to n

Given a number n, write a function that generates and prints all binary numbers with decimal values from 1 to n.

Examples:

Input: n = 2
Output: 1, 10

Input: n = 5
Output: 1, 10, 11, 100, 101

A simple method is to run a loop from 1 to n, call decimal to binary inside the loop.

Following is an interesting method that uses [queue data structure](#) to print binary numbers. Thanks to [Vivek](#) for suggesting this approach.

- 1) Create an empty queue of strings
- 2) Enqueue the first binary number 1 to queue.
- 3) Now run a loop for generating and printing n binary numbers.
 - a) Dequeue and Print the front of queue.
 - b) Append 0 at the end of front item and enqueue it.
 - c) Append 1 at the end of front item and enqueue it.

Following is C++ implementation of above algorithm.

```
// C++ program to generate binary numbers from 1 to n
#include <iostream>
#include <queue>
using namespace std;

// This function uses queue data structure to print binary numbers
void generatePrintBinary(int n)
{
    // Create an empty queue of strings
    queue<string> q;

    // Enqueue the first binary number
    q.push("1");

    // This loops is like BFS of a tree with 1 as root
    // 0 as left child and 1 as right child and so on
    while (n--)
    {
        // print the front of queue
        string s1 = q.front();
        q.pop();
        cout << s1 << "\n";

        string s2 = s1; // Store s1 before changing it

        // Append "0" to s1 and enqueue it
        q.push(s1.append("0"));

        // Append "1" to s2 and enqueue it. Note that s2 contains
        // the previous front
        q.push(s2.append("1"));
    }
}

// Driver program to test above function
int main()
{
    int n = 10;
    generatePrintBinary(n);
    return 0;
}
```

Output:

```
1
10
11
100
101
110
111
1000
1001
```


How to efficiently implement k Queues in a single array?

We have discussed [efficient implementation of k stack in an array](#). In this post, same for queue is discussed. Following is the detailed problem statement.

Create a data structure kQueues that represents k queues. Implementation of kQueues should use only one array, i.e., k queues should use the same array for storing elements. Following functions must be supported by kQueues.

enqueue(int x, int qn) > adds x to queue number qn where qn is from 0 to k-1

dequeue(int qn) > deletes an element from queue number qn where qn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use $\text{arr}[0]$ to $\text{arr}[n/k-1]$ for first queue, and $\text{arr}[n/k]$ to $\text{arr}[2n/k-1]$ for queue2 where $\text{arr}[]$ is the array to be used to implement two queues and size of array be n .

The problem with this method is inefficient use of array space. An enqueue operation may result in overflow even if there is space available in $\text{arr}[]$. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to second queue. When we enqueue 4th element to first queue, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is similar to the [stack post](#), here we need to use three extra arrays. In stack post, we needed to extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.

Following are the three extra arrays are used:

- 1) **front[]**: This is of size k and stores indexes of front elements in all queues.
- 2) **rear[]**: This is of size k and stores indexes of rear elements in all queues.
- 3) **next[]**: This is of size n and stores indexes of next item for all items in array $\text{arr}[]$.

Here $\text{arr}[]$ is actual array that stores k stacks.

Together with k queues, a stack of free slots in $\text{arr}[]$ is also maintained. The top of this stack is stored in a variable free .

All entries in $\text{front}[]$ are initialized as -1 to indicate that all queues are empty. All entries $\text{next}[i]$ are initialized as $i+1$ because all slots are free initially and pointing to next slot. Top of free stack, free is initialized as 0.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate implementation of k queues in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k queues in a single array of size n
class kQueues
{
    int *arr; // Array of size n to store actual content to be stored in queue
    int *front; // Array of size k to store indexes of front elements of queue
    int *rear; // Array of size k to store indexes of rear elements of queue
    int *next; // Array of size n to store next entry in all queues
                // and free list
    int n, k;
    int free; // To store beginning index of free list
public:
    //constructor to create k queue in an array of size n
    kQueues(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To enqueue an item in queue number 'qn' where qn is from 0 to k-1
    void enqueue(int item, int qn);

    // To dequeue an from queue number 'qn' where qn is from 0 to k-1
    int dequeue(int qn);

    // To check whether queue number 'qn' is empty or not
    bool isEmpty(int qn) { return (front[qn] == -1); }
};

// Constructor to create k queues in an array of size n
kQueues::kQueues(int k1, int n1)
```

```

{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    front = new int[k];
    rear = new int[k];
    next = new int[n];

    // Initialize all queues as empty
    for (int i = 0; i < k; i++)
        front[i] = -1;

    // Initialize all spaces as free
    free = 0;
    for (int i=0; i<n-1; i++)
        next[i] = i+1;
    next[n-1] = -1; // -1 is used to indicate end of free list
}

// To enqueue an item in queue number 'qn' where qn is from 0 to k-1
void kQueues::enqueue(int item, int qn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nQueue Overflow\n";
        return;
    }

    int i = free; // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    if (isEmpty(qn))
        front[qn] = i;
    else
        next[rear[qn]] = i;

    next[i] = -1;

    // Update next of rear and then rear for queue number 'qn'
    rear[qn] = i;

    // Put the item in array
    arr[i] = item;
}

// To dequeue an from queue number 'qn' where qn is from 0 to k-1
int kQueues::dequeue(int qn)
{
    // Underflow checkSAS
    if (isEmpty(qn))
    {
        cout << "\nQueue Underflow\n";
        return INT_MAX;
    }

    // Find index of front item in queue number 'qn'
    int i = front[qn];

    front[qn] = next[i]; // Change top to store next of previous top

    // Attach the previous front to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous front item
    return arr[i];
}

/* Driver program to test kStacks class */
int main()
{
    // Let us create 3 queue in an array of size 10
    int k = 3, n = 10;
    kQueues ks(k, n);

    // Let us put some items in queue number 2
    ks.enqueue(15, 2);
    ks.enqueue(45, 2);
}

```

```

// Let us put some items in queue number 1
ks.enqueue(17, 1);
ks.enqueue(49, 1);
ks.enqueue(39, 1);

// Let us put some items in queue number 0
ks.enqueue(11, 0);
ks.enqueue(9, 0);
ks.enqueue(7, 0);

cout << "Dequeued element from queue 2 is " << ks.dequeue(2) << endl;
cout << "Dequeued element from queue 1 is " << ks.dequeue(1) << endl;
cout << "Dequeued element from queue 0 is " << ks.dequeue(0) << endl;

return 0;
}

```

Output:

```

Dequeued element from queue 2 is 15
Dequeued element from queue 1 is 17
Dequeued element from queue 0 is 11

```

Time complexities of enqueue() and dequeue() is O(1).

The best part of above implementation is, if there is a slot available in queue, then an item can be enqueued in any of the queues, i.e., no wastage of space. This method requires some extra space. Space may not be an issue because queue items are typically large, for example queues of employees, students, etc where every item is of hundreds of bytes. For such large queues, the extra space used is comparatively very less as we use three integer arrays as extra space.

Handshaking Lemma and Interesting Tree Properties

What is Handshaking Lemma?

[Handshaking lemma](#) is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)

$$\sum_{v \in V} \deg(v) = 2|E|$$

How is Handshaking Lemma useful in Tree Data structure?

Following are some interesting facts that can be proved using Handshaking lemma.

1) In a k-ary tree where every node has either 0 or k children, following property is always true.

$$L = (k - 1)*I + 1$$

Where L = Number of leaf nodes

I = Number of internal nodes

Proof:

Proof can be divided in two cases.

Case 1 (Root is Leaf): There is only one node in tree. The above formula is true for single node as L = 1, I = 0.

Case 2 (Root is Internal Node): For trees with more than 1 nodes, root is always internal node. The above formula can be proved using Handshaking Lemma for this case. A tree is an undirected acyclic graph.

Total number of edges in Tree is number of nodes minus 1, i.e., $|E| = L + I - 1$.

All internal nodes except root in the given type of tree have degree $k + 1$. Root has degree k . All leaves have degree 1. Applying the Handshaking lemma to such trees, we get following relation.

$$\text{Sum of all degrees} = 2 * (\text{Sum of Edges})$$

$$\text{Sum of degrees of leaves} +$$

$$\text{Sum of degrees for Internal Node except root} +$$

$$\text{Root's degree} = 2 * (\text{No. of nodes} - 1)$$

Putting values of above terms,

$$L + (I-1)*(k+1) + k = 2 * (L + I - 1)$$

$$L + k*I - k + I - 1 + k = 2*L + 2*I - 2$$

$$L + K*I + I - 1 = 2*L + 2*I - 2$$

$$K*I + 1 - I = L$$

$$(K-1)*I + 1 = L$$

So the above property is proved using Handshaking Lemma, let us discuss one more interesting property.

2) In Binary tree, number of leaf nodes is always one more than nodes with two children.

$$L = T + 1$$

Where L = Number of leaf nodes

T = Number of internal nodes with two children

Proof:

Let number of nodes with 2 children be T. Proof can be divided in three cases.

Case 1: There is only one node, the relationship holds as $T=0, L=1$.

Case 2: Root has two children, i.e., degree of root is 2.

$$\text{Sum of degrees of nodes with two children except root} +$$

$$\text{Sum of degrees of nodes with one child} +$$

$$\text{Sum of degrees of leaves} + \text{Root's degree} = 2 * (\text{No. of Nodes} - 1)$$

Putting values of above terms,

$$(T-1)*3 + S*2 + L + 2 = (S + T + L - 1)*2$$

Cancelling 2S from both sides.

$$(T-1)*3 + L + 2 = (S + L - 1)*2$$

$$T - 1 = L - 2$$

$$T = L - 1$$

Case 3: Root has one child, i.e., degree of root is 1.

Sum of degrees of nodes with two children +
 Sum of degrees of nodes with one child except root +
 Sum of degrees of leaves + Root's degree = $2 * (\text{No. of Nodes} - 1)$

Putting values of above terms,
 $T*3 + (S-1)*2 + L + 1 = (S + T + L - 1)*2$

Cancelling $2S$ from both sides.
 $3*T + L - 1 = 2*T + 2*L - 2$
 $T - 1 = L - 2$
 $T = L - 1$

Therefore, in all three cases, we get $T = L - 1$.

We have discussed proof of two important properties of Trees using Handshaking Lemma. Many GATE questions have been asked on these properties, following are few links.

[GATE-CS-2015 \(Set 3\) | Question 35](#)

[GATE-CS-2015 \(Set 2\) | Question 20](#)

[GATE-CS-2005 | Question 36](#)

[GATE-CS-2002 | Question 34](#)

[GATE-CS-2007 | Question 43](#)

Binary Tree | Set 2 (Properties)

We have discussed [Introduction to Binary Tree in set 1](#). In this post, properties of binary are discussed.

1) The maximum number of nodes at level l of a binary tree is 2^{l-1} .

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.

This can be proved by induction.

For root, l= 1, number of nodes = $2^{1-1} = 1$

Assume that maximum number of nodes on level l is 2^{l-1}

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height h is $2^h - 1$.

Here height of a tree is maximum number of nodes on root to leaf path. Height of a leaf node is considered as 1.

This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + \dots + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h - 1$.

In some books, height of a leaf is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is ? $\log_2(N+1)$?

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes $\log_2(N+1) + 1$

4) A Binary Tree with L leaves has at least $\log_2 L + 1$ levels

A Binary tree has maximum number of leaves when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

$$\begin{aligned} L &\leq 2^{l-1} \quad [\text{From Point 1}] \\ \log_2 L &\leq l-1 \\ l &\geq \log_2 L + 1 \end{aligned}$$

5) In Binary tree, number of leaf nodes is always one more than nodes with two children.

$L = T + 1$
Where L = Number of leaf nodes
T = Number of internal nodes with two children

See [Handshaking Lemma and Tree](#) for proof.

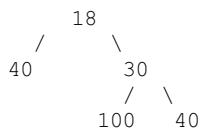
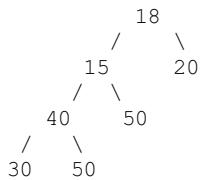
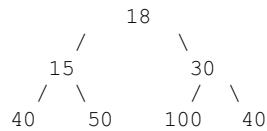
In the next article on tree series, we will be discussing [different types of Binary Trees and their properties](#).

Binary Tree | Set 3 (Types of Binary Tree)

We have discussed [Introduction to Binary Tree in set 1](#) and [Properties of Binary Tree in Set 2](#). In this post, common types of binary is discussed.

Following are common types of Binary Trees.

Full Binary Tree A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.



In a Full Binary, number of leaf nodes is number of internal nodes plus 1

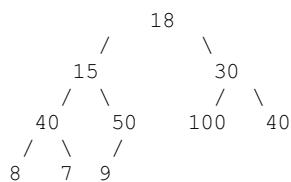
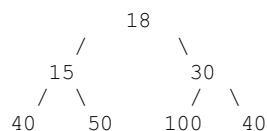
$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

See [Handshaking Lemma and Tree](#) for proof.

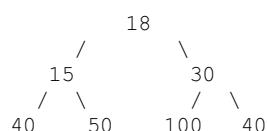
Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees



Practical example of Complete Binary Tree is [Binary Heap](#).

Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.
Following are examples of Perfect Binary Trees.



A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has $2^h - 1$ node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree

A binary tree is balanced if height of the tree is $O(\log n)$ where n is number of nodes. For Example, AVL tree maintains $O(\log n)$ height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain $O(\log n)$ height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide $O(\log n)$ time for search, insert and delete.

A degenerate (or pathological) tree A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

```
10
 /
20
 \
30
 \
40
```

Source:

https://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees

Applications of tree data structure

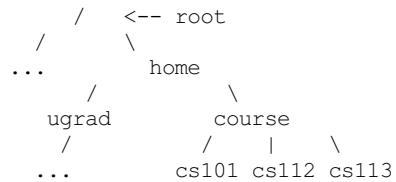
Difficulty Level: Rookie

Why Tree?

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

- 1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system



2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). [Self-balancing search trees](#) like [AVL](#) and [Red-Black trees](#) guarantee an upper bound of $O(\log n)$ for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). [Self-balancing search trees](#) like [AVL](#) and [Red-Black trees](#) guarantee an upper bound of $O(\log n)$ for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

[As per Wikipedia](#), following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

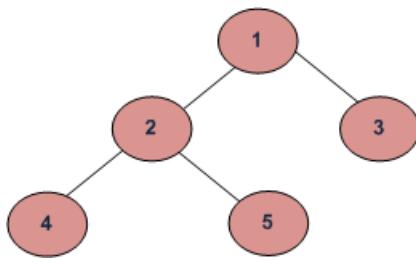
References:

<http://www.cs.bu.edu/teaching/c/tree/binary/>

http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Common_uses

Tree Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:

- (a) Inorder
- (b) Preorder
- (c) Postorder

Breadth First or Level Order Traversal

Please see [this](#) post for Breadth First Traversal.

Inorder Traversal:

```
Algorithm Inorder(tree)
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3.

Preorder Traversal:

```
Algorithm Preorder(tree)
1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

```
Algorithm Postorder(tree)
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.
```

Uses of Postorder

Postorder traversal is used to delete the tree. Please see [the question for deletion of tree](#) for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
```

```

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes according to the
   "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    printf("%d ", node->data);

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("\n Preorder traversal of binary tree is \n");
    printPreorder(root);

    printf("\n Inorder traversal of binary tree is \n");
    printInorder(root);

    printf("\n Postorder traversal of binary tree is \n");
    printPostorder(root);
}

```

```

getchar();
return 0;
}

```

Time Complexity: O(n)

Let us prove it:

Complexity function $T(n)$ for all problem where tree traversal is involved can be defined as:

$$T(n) = T(k) + T(n-k-1) + c$$

Where k is the number of nodes on one side of root and $n-k-1$ on the other side.

Lets do analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.

$$T(n) = T(0) + T(n-1) + c$$

$$T(n) = 2T(0) + T(n-2) + 2c$$

$$T(n) = 3T(0) + T(n-3) + 3c$$

$$T(n) = 4T(0) + T(n-4) + 4c$$

$$\dots$$

$$T(n) = (n-1)T(0) + T(1) + (n-1)c$$

$$T(n) = nT(0) + (n)c$$

Value of $T(0)$ will be some constant say d . (traversing a empty tree will take some constants time)

$$T(n) = n(c+d)$$

$$T(n) = \Theta(n)$$
 (Theta of n)

Case 2: Both left and right subtrees have equal number of nodes.

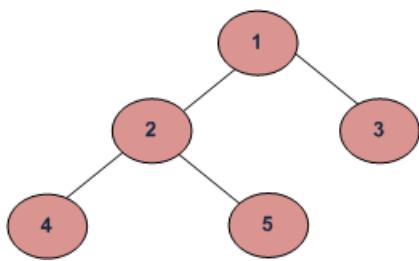
$$T(n) = 2T(\lfloor n/2 \rfloor) + c$$

This recursive function is in the standard form ($T(n) = aT(n/b) + \Theta(n)$) for master method http://en.wikipedia.org/wiki/Master_theorem. If we solve it by master method we get $\Theta(n)$

Auxiliary Space : If we dont consider size of stack for function calls then $O(1)$ otherwise $O(n)$.

Level Order Tree Traversal

Level order traversal of a tree is [breadth first traversal](#) for the tree.



Example Tree

Level order traversal of the above tree is 1 2 3 4 5

METHOD 1 (Use function to print a given level)

Algorithm:

There are basically two functions in this method. One is to print all nodes at a given level (`printGivenLevel`), and other is to print level order traversal of the tree (`printLevelorder`). `printLevelorder` makes use of `printGivenLevel` to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
    printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function prototypes*/
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for(i=1; i<=h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if(root == NULL)
        return;
    if(level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}
```

```

}

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);

    getchar();
    return 0;
}

```

Time Complexity: $O(n^2)$ in worst case. For a skewed tree, `printGivenLevel()` takes $O(n)$ time where n is the number of nodes in the skewed tree. So time complexity of `printLevelOrder()` is $O(n) + O(n-1) + O(n-2) + \dots + O(1)$ which is $O(n^2)$.

METHOD 2 (Use Queue)

Algorithm:

For each node, first the node is visited and then its child nodes are put in a FIFO queue.

```

printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
   a) print temp_node->data.
   b) Enqueue temp_nodes children (first left then right children) to q
   c) Dequeue a node from q and assign its value to temp_node

```

Implementation:

Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

```
#include <stdio.h>
#include <stdlib.h>
```

```

#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* frunction prototypes */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
   using array for implementing queue */
void printLevelOrder(struct node* root)
{
    int rear, front;
    struct node **queue = createQueue(&front, &rear);
    struct node *temp_node = root;

    while(temp_node)
    {
        printf("%d ", temp_node->data);

        /*Enqueue left child */
        if(temp_node->left)
            enQueue(queue, &rear, temp_node->left);

        /*Enqueue right child */
        if(temp_node->right)
            enQueue(queue, &rear, temp_node->right);

        /*Dequeue node and make it temp_node*/
        temp_node = deQueue(queue, &front);
    }
}

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
    struct node **queue =
        (struct node **)malloc(sizeof(struct node*) *MAX_Q_SIZE);

    *front = *rear = 0;
    return queue;
}

void enQueue(struct node **queue, int *rear, struct node *new_node)
{
    queue[*rear] = new_node;
    (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
    (*front)++;
    return queue[*front - 1];
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);

```

```
root->right      = newNode(3);
root->left->left  = newNode(4);
root->left->right = newNode(5);

printf("Level Order traversal of binary tree is \n");
printLevelOrder(root);

getchar();
return 0;
}
```

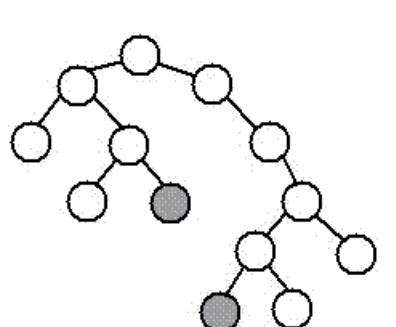
Time Complexity: O(n) where n is number of nodes in the binary tree

References:

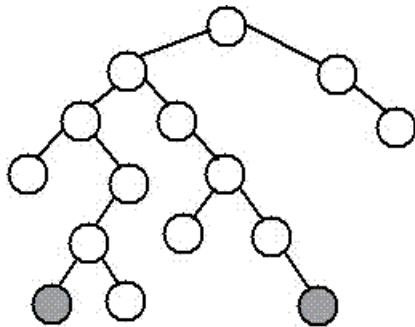
http://en.wikipedia.org/wiki/Breadth-first_traversal

Diameter of a Binary Tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



diameter, 9 nodes, through root



diameter, 9 nodes, NOT through root

The diameter of a tree T is the largest of the following quantities:

- * the diameter of Ts left subtree
- * the diameter of Ts right subtree
- * the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* function to Compute height of a tree. */
int height(struct node* node);

/* Function to get diameter of a binary tree */
int diameter(struct node * tree)
{
    /* base case where tree is empty */
    if (tree == 0)
        return 0;

    /* get the height of left and right sub-trees */
    int lheight = height(tree->left);
    int rheight = height(tree->right);

    /* get the diameter of left and right sub-trees */
    int ldiameter = diameter(tree->left);
    int rdiameter = diameter(tree->right);

    /* Return max of following three
       1) Diameter of left subtree
       2) Diameter of right subtree
       3) Height of left subtree + height of right subtree + 1 */
    return max(lheight + rheight + 1, max(ldiameter, rdiameter));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/* The function Compute the "height" of a tree. Height is the
   number f nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
```

```

{
    /* base case tree is empty */
    if(node == NULL)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b) ? a : b;
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
           1
         /   \
        2     3
       / \   /
      4   5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Diameter of the given binary tree is %d\n", diameter(root));

    getchar();
    return 0;
}

```

Time Complexity: O(n^2)

Optimized implementation: The above implementation can be optimized by calculating the height in the same recursion rather than calling a `height()` separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to O(n).

```

/*The second parameter is to store the height of tree.
Initially, we need to pass a pointer to a location with value
as 0. So, function should be used as follows:

int height = 0;
struct node *root = SomeFunctionToMakeTree();
int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
{
    /* lh --> Height of left subtree
       rh --> Height of right subtree */
    int lh = 0, rh = 0;

    /* ldiameter --> diameter of left subtree
       rdiameter --> Diameter of right subtree */
    int ldiameter = 0, rdiameter = 0;

    if(root == NULL)
    {
        *height = 0;
        return 0; /* diameter is also 0 */
    }

    /* Compute height of left and right subtrees */
    lh = diameterOpt(root->left, height);
    rh = diameterOpt(root->right, height);

    /* Compute diameter of current node */
    ldiameter = lh + rh + 1;
    rdiameter = max(lh, rh);

    /* Compute maximum of two diameters */
    diameter = max(ldiameter, rdiameter);

    /* Return height of current node */
    *height = max(lh, rh);
}
```

```
}

/* Get the heights of left and right subtrees in lh and rh
 And store the returned values in ldiameter and rdiameter */
ldiameter = diameterOpt(root->left, &lh);
rdiameter = diameterOpt(root->right, &rh);

/* Height of current node is max of heights of left and
 right subtrees plus 1*/
*height = max(lh, rh) + 1;

return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

Time Complexity: O(n)

References:

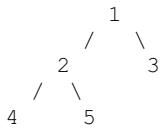
<http://www.cs.duke.edu/courses/spring00/cps100/assign/trees/diameter.html>

Inorder Tree Traversal without Recursion

Using [Stack](#) is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See [this](#) for step wise step execution of the algorithm.

- 1) Create an empty stack S.
- 2) Initialize current node as root
- 3) Push the current node to S and set current = current->left until current is NULL
- 4) If current is NULL and stack is not empty then
 - a) Pop the top item from stack.
 - b) Print the popped item, set current = popped_item->right
 - c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.

Let us consider the below tree for example



Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current is NULL

current -> 1

push 1: Stack S -> 1

current -> 2

push 2: Stack S -> 2, 1

current -> 4

push 4: Stack S -> 4, 2, 1

current = NULL

Step 4 pops from S

a) Pop 4: Stack S -> 2, 1

b) print "4"

c) current = NULL /*right of 4 */ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

a) Pop 2: Stack S -> 1

b) print "2"

c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

Stack S -> 5, 1

current = NULL

Step 4 pops from S

a) Pop 5: Stack S -> 1

b) print "5"

c) current = NULL /*right of 5 */ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

a) Pop 1: Stack S -> NULL

b) print "1"

c) current -> 3 /*right of 5 */

Step 3 pushes 3 to stack and makes current NULL

Stack S -> 3

current = NULL

Step 4 pops from S

a) Pop 3: Stack S -> NULL

b) print "3"

c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree tNode has data, pointer to left child
```

```

        and a pointer to right child */
struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Structure of a stack node. Linked List implementation is used for
stack. A stack node contains a pointer to tree node and a pointer to
next stack node */
struct sNode
{
    struct tNode *t;
    struct sNode *next;
};

/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inOrder(struct tNode *root)
{
    /* set current to root of binary tree */
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        /* Reach the left most tNode of the current tNode */
        if(current != NULL)
        {
            /* place pointer to a tree node on the stack before traversing
               the node's left subtree */
            push(&s, current);
            current = current->left;
        }

        /* backtrack from the empty subtree and visit the tNode
           at the top of the stack; however, if the stack is empty,
           you are done */
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);

                /* we have visited the node and its left subtree.
                   Now, it's right subtree's turn */
                current = current->right;
            }
            else
                done = 1;
        }
    } /* end of while */
}

/* UTILITY FUNCTIONS */
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)
{
    /* allocate tNode */
    struct sNode* new_tNode =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_tNode == NULL)
    {
        printf("Stack Overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_tNode->t = t;

    /* link the old list off the new tNode */
    new_tNode->next = (*top_ref);
}

```

```

/* move the head to point to the new tNode */
(*top_ref) = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
    struct tNode *res;
    struct sNode *top;

    /*If sNode is empty then error */
    if(isEmpty(*top_ref))
    {
        printf("Stack Underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->t;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
                           malloc(sizeof(struct tNode));
    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
       1
      / \
     2   3
    / \
   4   5
    */
    struct tNode *root = newtNode(1);
    root->left = newtNode(2);
    root->right = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    inOrder(root);

    getchar();
    return 0;
}

```

Time Complexity: O(n)

References:

<http://web.cs.wpi.edu/~cs2005/common/iterative.inorder>

<http://neural.cs.nthu.edu.tw/jang/courses/cs2351/slides/animation/Iterative%20Inorder%20Traversal.pps>

See [this post](#) for another approach of Inorder Tree Traversal without recursion and without stack!

Inorder Tree Traversal without recursion and without stack!

Using Morris Traversal, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on [Threaded Binary Tree](#). In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

1. Initialize current as root
2. While current is not NULL
 - If current does not have left child
 - a) Print currents data
 - b) Go to the right, i.e., current = current->right
 - Else
 - a) Make current as right child of the rightmost node in current's left subtree
 - b) Go to this left child, i.e., current = current->left

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike [Stack based traversal](#), no extra space is required for this traversal.

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Function to traverse binary tree without recursion and
   without stack */
void MorrisTraversal(struct tNode *root)
{
    struct tNode *current,*pre;

    if(root == NULL)
        return;

    current = root;
    while(current != NULL)
    {
        if(current->left == NULL)
        {
            printf(" %d ", current->data);
            current = current->right;
        }
        else
        {
            /* Find the inorder predecessor of current */
            pre = current->left;
            while(pre->right != NULL && pre->right != current)
                pre = pre->right;

            /* Make current as right child of its inorder predecessor */
            if(pre->right == NULL)
            {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in if part to restore the original
               tree i.e., fix the right child of predecessor */
            else
            {
                pre->right = NULL;
                printf(" %d ",current->data);
                current = current->right;
            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL*/
    } /* End of while */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
```

```

        malloc(sizeof(struct tNode));
tNode->data = data;
tNode->left = NULL;
tNode->right = NULL;

return(tNode);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
           1
          /   \
         2     3
        / \   /
       4   5 4
    */
    struct tNode *root = newtNode(1);
    root->left      = newtNode(2);
    root->right     = newtNode(3);
    root->left->left  = newtNode(4);
    root->left->right = newtNode(5);

    MorrisTraversal(root);

    getchar();
    return 0;
}

```

References:

- www.liacs.nl/~deutz/DS/september28.pdf
- <http://comsci.liu.edu/~murali/algo/Morris.htm>
- www.scss.tcd.ie/disciplines/software_systems//HughGibbonsSlides.pdf

Threaded Binary Tree

[Inorder traversal of a Binary tree](#) is either be done using recursion or [with the use of a auxiliary stack](#). The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

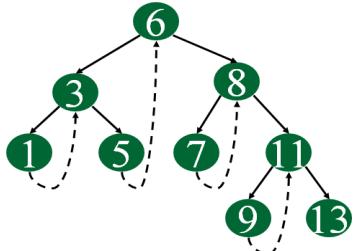
There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
    int data;
    Node *left, *right;
    bool rightThread;
}
```

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads

Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
        return NULL;

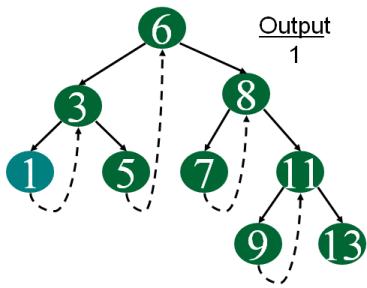
    while (n->left != NULL)
        n = n->left;

    return n;
}

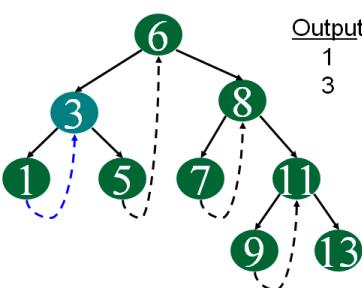
// C code to do inorder traversal in a threaded binary tree
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
}
```

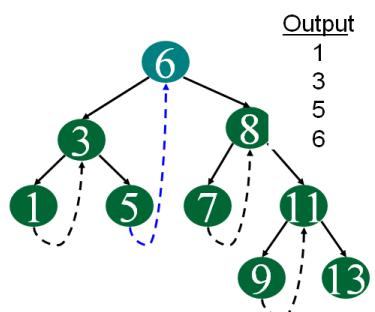
Following diagram demonstrates inorder order traversal using threads.



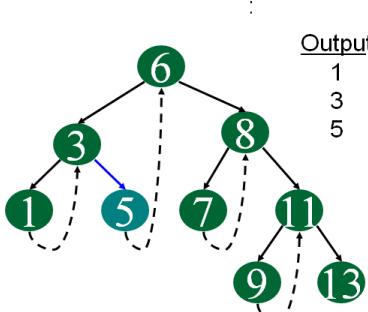
Start at leftmost node, print it



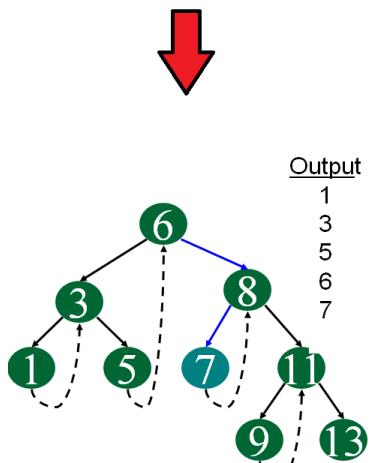
Follow thread to right, print node



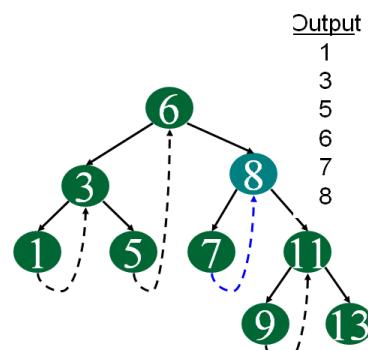
Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow link to right, go to leftmost node and print



Follow thread to right, print node

continue same way for remaining node.....

We will soon be discussing insertion and deletion in threaded binary trees.

Sources:

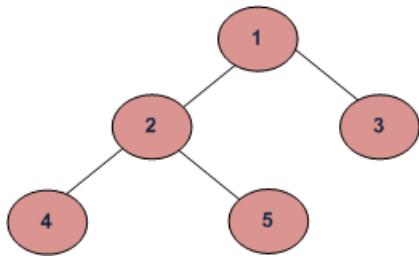
http://en.wikipedia.org/wiki/Threaded_binary_tree

www.cs.berkeley.edu/~kamil/teaching/su02/080802.ppt

[GATE Corner](#)[Quiz Corner](#)

Write a C Program to Find the Maximum Depth or Height of a Tree

Maximum depth or height of the below tree is 3.



Example Tree

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

Algorithm:

```

maxDepth()
1. If tree is empty then return 0
2. Else
    (a) Get the max depth of left subtree recursively i.e.,
        call maxDepth( tree->left-subtree)
    (a) Get the max depth of right subtree recursively i.e.,
        call maxDepth( tree->right-subtree)
    (c) Get the max of max depths of left and right
        subtrees and add 1 to it for the current node.
        max_depth = max(max dept of left subtree,
                         max depth of right subtree)
                     + 1
    (d) Return max_depth

```

See the below diagram for more clarity about execution of the recursive function `maxDepth()` for above example tree.

```

maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1
                = 2 + 1
                  /   \
                  /     \
                  /       \
                  /         \
maxDepth('1')           maxDepth('3') = 1
= max(maxDepth('4'), maxDepth('5')) + 1
= 1 + 1 = 2
                  /   \
                  /     \
                  /       \
                  /         \
maxDepth('4') = 1      maxDepth('5') = 1

```

Implementation:

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int maxDepth(struct node* node)
{
    if (node==NULL)
        return 0;
    else

```

```

{
    /* compute the depth of each subtree */
    int lDepth = maxDepth(node->left);
    int rDepth = maxDepth(node->right);

    /* use the larger one */
    if (lDepth > rDepth)
        return(lDepth+1);
    else return(rDepth+1);
}
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Height of tree is %d", maxDepth(root));

    getchar();
    return 0;
}

```

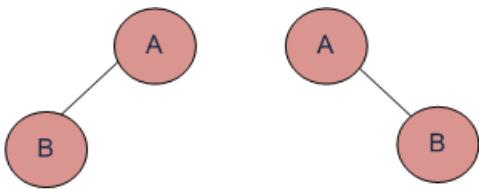
Time Complexity: O(n) (Please see our post [Tree Traversal](#) for details)

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

If you are given two traversal sequences, can you construct the binary tree?

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.



Trees having Preorder, Postorder and Level-Order and
traversals

Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

And following do not.

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

Preorder Traversal = AB

Postorder Traversal = BA

Level-Order Traversal = AB

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

Clone a Binary Tree with Random Pointers

Given a Binary Tree where every node has following structure.

```
struct node {
    int key;
    struct node *left, *right, *random;
}
```

The random pointer points to any random node of the binary tree and can even point to NULL, clone the given binary tree.

Method 1 (Use Hashing)

The idea is to store mapping from given tree nodes to clone tree node in hashtable. Following are detailed steps.

- 1) Recursively traverse the given Binary and copy key value, left pointer and right pointer to clone tree. While copying, store the mapping from given tree node to clone tree node in a hashtable. In the following pseudo code, cloneNode is currently visited node of clone tree and treeNode is currently visited node of given tree.

```
cloneNode->key = treeNode->key
cloneNode->left = treeNode->left
cloneNode->right = treeNode->right
map[treeNode] = cloneNode
```

- 2) Recursively traverse both trees and set random pointers using entries from hash table.

```
cloneNode->random = map[treeNode->random]
```

Following is C++ implementation of above idea. The following implementation uses [map](#) from C++ STL. Note that map doesn't implement hash table, it actually is based on self-balancing binary search tree.

```
// A hashmap based C++ program to clone a binary tree with random pointers
#include<iostream>
#include<map>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
   child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
   given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* then print data of Node and its random */
    cout << "[" << node->key << " ";
    if (node->random == NULL)
        cout << "NULL], ";
    else
        cout << node->random->key << "], ";

    /* now recur on right subtree */
    printInorder(node->right);
}

// This function creates clone by copying key and left and right pointers
// This function also stores mapping from given tree node to clone.
Node* copyLeftRightNode(Node* treeNode, map<Node *, Node *> *mymap)
{
    if (treeNode == NULL)
        return NULL;
```

```

        return NULL;
Node* cloneNode = newNode(treeNode->key);
(*mymap)[treeNode] = cloneNode;
cloneNode->left = copyLeftRightNode(treeNode->left, mymap);
cloneNode->right = copyLeftRightNode(treeNode->right, mymap);
return cloneNode;
}

// This function copies random node by using the hashmap built by
// copyLeftRightNode()
void copyRandom(Node* treeNode, Node* cloneNode, map<Node *, Node *> *mymap)
{
    if (cloneNode == NULL)
        return;
    cloneNode->random = (*mymap)[treeNode->random];
    copyRandom(treeNode->left, cloneNode->left, mymap);
    copyRandom(treeNode->right, cloneNode->right, mymap);
}

// This function makes the clone of given tree. It mainly uses
// copyLeftRightNode() and copyRandom()
Node* cloneTree(Node* tree)
{
    if (tree == NULL)
        return NULL;
    map<Node *, Node *> *mymap = new map<Node *, Node *>;
    Node* newTree = copyLeftRightNode(tree, mymap);
    copyRandom(tree, newTree, mymap);
    return newTree;
}

/* Driver program to test above functions*/
int main()
{
    //Test No 1
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->random = tree->left->right;
    tree->left->left->random = tree;
    tree->left->right->random = tree->right;

    // Test No 2
    // tree = NULL;

    // Test No 3
    // tree = newNode(1);

    // Test No 4
    /* tree = newNode(1);
       tree->left = newNode(2);
       tree->right = newNode(3);
       tree->random = tree->right;
       tree->left->random = tree;
    */
    cout << "Inorder traversal of original binary tree is: \n";
    printInorder(tree);

    Node *clone = cloneTree(tree);

    cout << "\n\nInorder traversal of cloned binary tree is: \n";
    printInorder(clone);

    return 0;
}

```

Output:

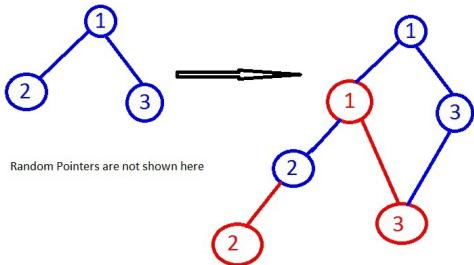
Inorder traversal of original binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],

Inorder traversal of cloned binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],

Method 2 (Temporarily Modify the Given Binary Tree)

1. Create new nodes in cloned tree and insert each new node in original tree between the left pointer edge of corresponding node in the original tree (See the below image).

i.e. if current node is A and its left child is B (A >> B), then new cloned node with key A will be created (say cA) and it will be put as A >> cA >> B (B can be a NULL or a non-NUL left child). Right child pointer will be set correctly i.e. if for current node A, right child is C in original tree (A >> C) then corresponding cloned nodes cA and cC will like cA ->> cC



2. Set random pointer in cloned tree as per original tree

i.e. if node A's random pointer points to node B, then in cloned tree, cA will point to cB (cA and cB are new node in cloned tree corresponding to node A and B in original tree)

3. Restore left pointers correctly in both original and cloned tree

Following is C++ implementation of above algorithm.

```
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
   child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
   given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* then print data of Node and its random */
    cout << "[" << node->key << " ";
    if (node->random == NULL)
        cout << "NULL], ";
    else
        cout << node->random->key << "], ";

    /* now recur on right subtree */
    printInorder(node->right);
}

// This function creates new nodes cloned tree and puts new cloned node
// in between current node and it's left child
// i.e. if current node is A and it's left child is B ( A --- >> B ),
//      then new cloned node with key A will be created (say cA) and
//      it will be put as
//      A --- >> cA --- >> B
// Here B can be a NULL or a non-NUL left child
// Right child pointer will be set correctly
// i.e. if for current node A, right child is C in original tree
//      (A --- >> C) then corresponding cloned nodes cA and cC will like
//      cA ----- >> cC
Node* copyLeftRightNode(Node* treeNode)
```

```

{
    if (treeNode == NULL)
        return NULL;

    Node* left = treeNode->left;
    treeNode->left = newNode(treeNode->key);
    treeNode->left->left = left;
    if(left != NULL)
        left->left = copyLeftRightNode(left);

    treeNode->left->right = copyLeftRightNode(treeNode->right);
    return treeNode->left;
}

// This function sets random pointer in cloned tree as per original tree
// i.e. if node A's random pointer points to node B, then
// in cloned tree, cA wil point to cB (cA and cB are new node in cloned
// tree corresponding to node A and B in original tree)
void copyRandomNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if(treeNode->random != NULL)
        cloneNode->random = treeNode->random->left;
    else
        cloneNode->random = NULL;

    if(treeNode->left != NULL && cloneNode->left != NULL)
        copyRandomNode(treeNode->left->left, cloneNode->left->left);
    copyRandomNode(treeNode->right, cloneNode->right);
}

// This function will restore left pointers correctly in
// both original and cloned tree
void restoreTreeLeftNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if (cloneNode->left != NULL)
    {
        Node* cloneLeft = cloneNode->left->left;
        treeNode->left = treeNode->left->left;
        cloneNode->left = cloneLeft;
    }
    else
        treeNode->left = NULL;

    restoreTreeLeftNode(treeNode->left, cloneNode->left);
    restoreTreeLeftNode(treeNode->right, cloneNode->right);
}

//This function makes the clone of given tree
Node* cloneTree(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = copyLeftRightNode(treeNode);
    copyRandomNode(treeNode, cloneNode);
    restoreTreeLeftNode(treeNode, cloneNode);
    return cloneNode;
}

/* Driver program to test above functions*/
int main()
{
/* //Test No 1
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->left = newNode(4);
tree->left->right = newNode(5);
tree->random = tree->left->right;
tree->left->left->random = tree;
tree->left->right->random = tree->right;

// Test No 2
// Node *tree = NULL;
/*
// Test No 3
Node *tree = newNode(1);
```

```

// Test No 4
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->random = tree->right;
tree->left->random = tree;

Test No 5
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->right->left = newNode(6);
    tree->right->right = newNode(7);
tree->random = tree->left;
*/
// Test No 6
Node *tree = newNode(10);
Node *n2 = newNode(6);
Node *n3 = newNode(12);
Node *n4 = newNode(5);
Node *n5 = newNode(8);
Node *n6 = newNode(11);
Node *n7 = newNode(13);
Node *n8 = newNode(7);
Node *n9 = newNode(9);
tree->left = n2;
tree->right = n3;
tree->random = n2;
n2->left = n4;
n2->right = n5;
n2->random = n8;
n3->left = n6;
n3->right = n7;
n3->random = n5;
n4->random = n9;
n5->left = n8;
n5->right = n9;
n5->random = tree;
n6->random = n9;
n9->random = n8;

/* Test No 7
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->random = tree;
tree->right->random = tree->left;
*/
    cout << "Inorder traversal of original binary tree is: \n";
    printInorder(tree);

    Node *clone = cloneTree(tree);

    cout << "\n\nInorder traversal of cloned binary tree is: \n";
    printInorder(clone);

    return 0;
}

```

Output:

```

Inorder traversal of original binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL], 

Inorder traversal of cloned binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL], 

```

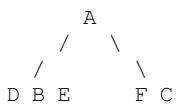
Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals:

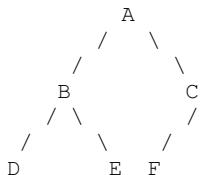
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element is the root of the tree. So we know A is root for given sequences. By searching A in Inorder sequence, we can find out all elements on left side of A are in left subtree and elements on right are in right subtree. So we know below structure now.



We recursively follow above steps and get the following tree.



Algorithm: buildTree()

- 1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.
- 2) Create a new tree node tNode with the data as picked element.
- 3) Find the picked elements index in Inorder. Let the index be inIndex.
- 4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
- 5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
- 6) return tNode.

Thanks to Rohini and [Tushar](#) for suggesting the code.

```
/* program to construct tree using inorder and preorder traversals */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    char data;
    struct node* left;
    struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[]. Initial values
   of inStrt and inEnd should be 0 and len -1. The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
    static int preIndex = 0;

    if(inStrt > inEnd)
        return NULL;

    /* Pick current node from Preorder traversal using preIndex
       and increment preIndex */
    struct node *tNode = newNode(pre[preIndex++]);

    /* If this node has no children then return */
    if(inStrt == inEnd)
        return tNode;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = search(in, inStrt, inEnd, tNode->data);

    /* Using index in Inorder traversal, construct left and
       right subtress */
    tNode->left = buildTree(in, pre, inStrt, inIndex - 1);
    tNode->right = buildTree(in, pre, inIndex + 1, inEnd);
}
```

```

tNode->left = buildTree(in, pre, inStrt, inIndex-1);
tNode->right = buildTree(in, pre, inIndex+1, inEnd);

return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
    int i;
    for(i = strt; i <= end; i++)
    {
        if(arr[i] == value)
            return i;
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(char data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

/* This function is here just to test buildTree() */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%c ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};
    char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};
    int len = sizeof(in)/sizeof(in[0]);
    struct node *root = buildTree(in, pre, 0, len - 1);

    /* Let us test the built tree by printing Inorder traversal */
    printf("\n Inorder traversal of the constructed tree is \n");
    printInorder(root);
    getchar();
}

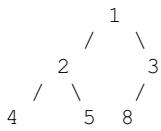
```

Time Complexity: O(n^2). Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

Print nodes at k distance from root

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.



The problem can be solved using recursion. Thanks to [eldho](#) for suggesting the solution.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

void printKDistant(node *root , int k)
{
    if(root == NULL)
        return;
    if( k == 0 )
    {
        printf( "%d ", root->data );
        return ;
    }
    else
    {
        printKDistant( root->left, k-1 ) ;
        printKDistant( root->right, k-1 ) ;
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
       1
      / \
     2   3
    / \ /
   4   5 8
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(8);

    printKDistant(root, 2);

    getchar();
    return 0;
}
```

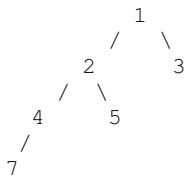
The above program prints 4, 5 and 8.

Time Complexity: $O(n)$ where n is number of nodes in the given binary tree.

Print Ancestors of a given node in Binary Tree

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.



Thanks to [Mike](#), [Sambasiva](#) and [wgpshashank](#) for their contribution.

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>

using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* If target is present in tree, then prints the ancestors
   and returns true, otherwise returns false. */
bool printAncestors(struct node *root, int target)
{
    /* base cases */
    if (root == NULL)
        return false;

    if (root->data == target)
        return true;

    /* If target is present in either left or right subtree of this node,
       then print this node */
    if (printAncestors(root->left, target) ||
        printAncestors(root->right, target) )
    {
        cout << root->data << " ";
        return true;
    }

    /* Else return false */
    return false;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{

    /* Construct the following binary tree
       1
      /   \
     2     3
    / \   / \
   4   5 7   8
    */
}
```

```
*/  
struct node *root = newnode(1);  
root->left = newnode(2);  
root->right = newnode(3);  
root->left->left = newnode(4);  
root->left->right = newnode(5);  
root->left->left->left = newnode(7);  
  
printAncestors(root, 7);  
  
getchar();  
return 0;  
}
```

Output:

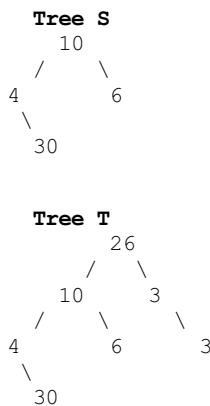
4 2 1

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

Check if a binary tree is subtree of another binary tree | Set 1

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T.



Solution: Traverse the tree T in preorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S.

Following is C implementation for this.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to check whether trees with roots as root1 and
   root2 are identical or not */
bool areIdentical(struct node * root1, struct node *root2)
{
    /* base cases */
    if (root1 == NULL && root2 == NULL)
        return true;

    if (root1 == NULL || root2 == NULL)
        return false;

    /* Check if the data of both roots is same and data of left and right
       subtrees are also same */
    return (root1->data == root2->data &&
            areIdentical(root1->left, root2->left) &&
            areIdentical(root1->right, root2->right) );
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(struct node *T, struct node *S)
{
    /* base cases */
    if (S == NULL)
        return true;

    if (T == NULL)
        return false;

    /* Check the tree with root as current node */
    if (areIdentical(T, S))
        return true;

    /* If the tree with root as current node doesn't match then
       try left and right subtrees one by one */
    return isSubtree(T->left, S) ||
           isSubtree(T->right, S);
}
```

```

/* Helper function that allocates a new node with the given data
   and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    /* Construct the following tree
       26
      /   \
     10   3
    / \   \
   4   6   3
  / \
 30
*/
    struct node *T = newNode(26);
    T->right = newNode(3);
    T->right->right = newNode(3);
    T->left = newNode(10);
    T->left->left = newNode(4);
    T->left->left->right = newNode(30);
    T->left->right = newNode(6);

    /* Construct the following tree
       10
      /   \
     4   6
    \   \
   30
*/
    struct node *S = newNode(10);
    S->right = newNode(6);
    S->left = newNode(4);
    S->left->right = newNode(30);

    if (isSubtree(T, S))
        printf("Tree S is subtree of tree T");
    else
        printf("Tree S is not a subtree of tree T");

    getchar();
    return 0;
}

```

Output:

Tree S is subtree of tree T

Time Complexity: Time worst case complexity of above solution is $O(mn)$ where m and n are number of nodes in given two trees.

We can solve the above problem in $O(n)$ time. Please refer [Check if a binary tree is subtree of another binary tree | Set 2](#) for $O(n)$ solution.

Connect nodes at same level

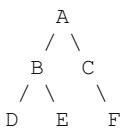
Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```
struct node{
    int data;
    struct node* left;
    struct node* right;
    struct node* nextRight;
}
```

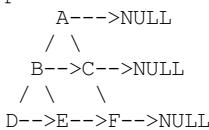
Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node.

Example

Input Tree



Output Tree



Method 1 (Extend Level Order Traversal or BFS)

Consider the method 2 of [Level Order Traversal](#). The method 2 can easily be extended to connect nodes of same level. We can augment queue entries to contain level of nodes also which is 0 for root, 1 for roots children and so on. So a queue node will now contain a pointer to a tree node and an integer level. When we enqueue a node, we make sure that correct level value for node is being set in queue. To set nextRight, for every node N, we dequeue the next node from queue, if the level number of next node is same, we set the nextRight of N as address of the dequeued node, otherwise we set nextRight of N as NULL.

Time Complexity: O(n)

Method 2 (Extend Pre Order Traversal)

This approach works only for [Complete Binary Trees](#). In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p, we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of ps left child (`p->left->nextRight`) will always be ps right child, and nextRight of ps right child (`p->right->nextRight`) will always be left child of ps nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of ps right child will be NULL.

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *nextRight;
};

void connectRecur(struct node* p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendants of p.
   Assumption: p is a compete binary tree */
void connectRecur(struct node* p)
{
    // Base case
    if (!p)
        return;

    // Set the nextRight pointer for p's left child
    if (p->left)
        p->left->nextRight = p->right;

    // Set the nextRight pointer for p's right child
    if (p->right)
        p->right->nextRight = p->right->right;

    connectRecur(p->right);
}
```

```

if (p->left)
    p->left->nextRight = p->right;

// Set the nextRight pointer for p's right child
// p->nextRight will be NULL if p is the right most child at its level
if (p->right)
    p->right->nextRight = (p->nextRight) ? p->nextRight->left: NULL;

// Set nextRight for other nodes in pre order fashion
connectRecur(p->left);
connectRecur(p->right);
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
       10
      /   \
     8     2
     /   \
    3     4
    */
    struct node *root = newnode(10);
    root->left = newnode(8);
    root->right = newnode(2);
    root->left->left = newnode(3);

    // Populates nextRight pointer in all nodes
    connect(root);

    // Let us check the values of nextRight pointers
    printf("Following are populated nextRight pointers in the tree "
          "(-1 is printed if there is no nextRight) \n");
    printf("nextRight of %d is %d \n", root->data,
           root->nextRight? root->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->data,
           root->left->nextRight? root->left->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->right->data,
           root->right->nextRight? root->right->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->left->data,
           root->left->left->nextRight? root->left->left->nextRight->data: -1);

    getchar();
    return 0;
}

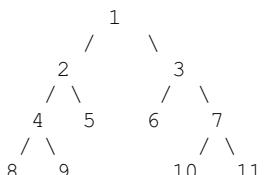
```

Thanks to Dhanya for suggesting this approach.

Time Complexity: O(n)

Why doesn't method 2 work for trees which are not Complete Binary Trees?

Let us consider following tree as an example. In Method 2, we set the nextRight pointer in pre order fashion. When we are at node 4, we set the nextRight of its children which are 8 and 9 (the nextRight of 4 is already set as node 5). nextRight of 8 will simply be set as 9, but nextRight of 9 will be set as NULL which is incorrect. We can't set the correct nextRight, because when we set nextRight of 9, we only have nextRight of node 4 and ancestors of node 4, we don't have nextRight of nodes in right subtree of root.



See [next post](#) for more solutions.

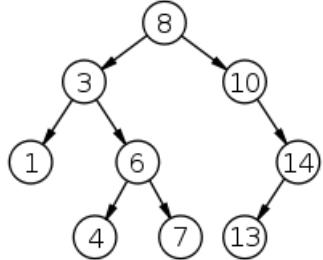
Binary Search Tree | Set 1 (Search and Insertion)

The following is definition of Binary Search Tree(BST) according to [Wikipedia](#)

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- The left and right subtree each must also be a binary search tree.

There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

C/C++

```
// C function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

Python

```
# A utility function to search a given key in BST
def search(root, key):

    # Base Cases: root is null or key is present at root
    if root is None or root.val == key:
        return root

    # Key is greater than root's key
    if root.val < key:
        return search(root.right, key)

    # Key is smaller than root's key
    return search(root.left, key)

# This code is contributed by Bhavya Jain
```

Java

```
// A utility function to search a given key in BST
public Node search(Node root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root==null || root.key==key)
        return root;

    // val is greater than root's key
```

```

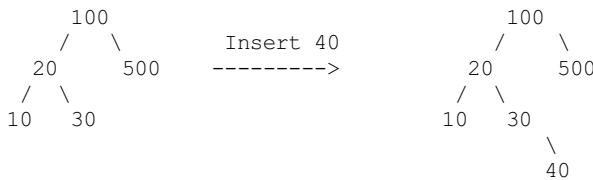
if (root.key > key)
    return search(root.left, key);

// val is less than root's key
return search(root.right, key);
}

```

Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.



C/C++

```

// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
       50
      /   \
     30   70
    / \   / \
   20 40 60 80 */
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
}

```

```

insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

// print inoder traversal of the BST
inorder(root);

return 0;
}

```

Python

```

# Python program to demonstrate insert operation in binary search tree

# A utility class that represents an individual node in a BST
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

    # A utility function to insert a new node with the given key
    def insert(self, node):
        if root is None:
            root = node
        else:
            if root.val < node.val:
                if root.right is None:
                    root.right = node
                else:
                    insert(root.right, node)
            else:
                if root.left is None:
                    root.left = node
                else:
                    insert(root.left, node)

    # A utility function to do inorder tree traversal
    def inorder(self):
        if root:
            inorder(root.left)
            print(root.val)
            inorder(root.right)

# Driver program to test the above functions
# Let us create the following BST
#      50
#     /   \
#    30   70
#   / \   / \
#  20 40 60 80
r = Node(50)
insert(r, Node(30))
insert(r, Node(20))
insert(r, Node(40))
insert(r, Node(70))
insert(r, Node(60))
insert(r, Node(80))

# Print inoder traversal of the BST
inorder(r)

# This code is contributed by Bhavya Jain

```

Java

```

// Java program to demonstrate insert operation in binary search tree
class BinarySearchTree {

    /* Class containing left and right child of current node and key value*/
    class Node {
        int key;
        Node left, right;

        public Node(int item) {
            key = item;
        }
    }

    Node root;
}
```

```

        left = right = null;
    }
}

// Root of BST
Node root;

// Constructor
BinarySearchTree() {
    root = null;
}

// This method mainly calls insertRec()
void insert(int key) {
    root = insertRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key) {

    /* If the tree is empty, return a new node */
    if (root == null) {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
}

// This method mainly calls InorderRec()
void inorder() {
    inorderRec(root);
}

// A utility function to do inorder traversal of BST
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.println(root.key);
        inorderRec(root.right);
    }
}

// Driver Program to test above functions
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
       50
      /   \
     30   70
    / \   / \
   20 40 60 80 */
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    // print inorder traversal of the BST
    tree.inorder();
}
}

// This code is contributed by Ankur Narain Verma

```

20
30
40
50
60

Time Complexity: The worst case time complexity of search and insert operations is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become $O(n)$.

[Binary Search Tree Delete Operation](#)

[Quiz on Binary Search Tree](#)

[GATE Corner](#)[Quiz Corner](#)

Binary Search Tree | Set 2 (Delete)

We have discussed [BST search and insert operations](#). In this post, delete operation is discussed. When we delete a node, there possibilities arise.

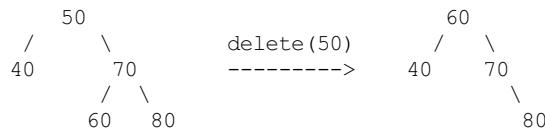
1) Node to be deleted is leaf: Simply remove from the tree.



2) Node to be deleted has only one child: Copy the child to the node and delete the child



3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

C/C++

```
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

```

}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
       50
      /     \
     30     70
    / \   / \
   20  40  60  80 */
    struct node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
}

```

```

printf("Inorder traversal of the given tree \n");
inorder(root);

printf("\nDelete 20\n");
root = deleteNode(root, 20);
printf("Inorder traversal of the modified tree \n");
inorder(root);

printf("\nDelete 30\n");
root = deleteNode(root, 30);
printf("Inorder traversal of the modified tree \n");
inorder(root);

printf("\nDelete 50\n");
root = deleteNode(root, 50);
printf("Inorder traversal of the modified tree \n");
inorder(root);

return 0;
}

```

Java

```

// Java program to demonstrate delete operation in binary search tree
class BinarySearchTree
{
    /* Class containing left and right child of current node and key value*/
    class Node
    {
        int key;
        Node left, right;

        public Node(int item)
        {
            key = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // This method mainly calls deleteRec()
    void deleteKey(int key)
    {
        root = deleteRec(root, key);
    }

    /* A recursive function to insert a new key in BST */
    Node deleteRec(Node root, int key)
    {
        /* Base Case: If the tree is empty */
        if (root == null) return root;

        /* Otherwise, recur down the tree */
        if (key < root.key)
            root.left = deleteRec(root.left, key);
        else if (key > root.key)
            root.right = deleteRec(root.right, key);

        // if key is same as root's key, then This is the node
        // to be deleted
        else
        {
            // node with only one child or no child
            if (root.left == null)
                return root.right;
            else if (root.right == null)
                return root.left;

            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            root.key = minValue(root.right);
        }
    }
}

```

```

        // Delete the inorder successor
        root.right = deleteRec(root.right, root.key);
    }

    return root;
}

int minValue(Node root)
{
    int minv = root.key;
    while (root.left != null)
    {
        minv = root.left.key;
        root = root.left;
    }
    return minv;
}

// This method mainly calls insertRec()
void insert(int key)
{
    root = insertRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key)
{
    /* If the tree is empty, return a new node */
    if (root == null)
    {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
}

// This method mainly calls InorderRec()
void inorder()
{
    inorderRec(root);
}

// A utility function to do inorder traversal of BST
void inorderRec(Node root)
{
    if (root != null)
    {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Driver Program to test above functions
public static void main(String[] args)
{
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
       50
      /   \
     30   70
    / \   / \
   20 40 60 80 */
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);
}

```

```

System.out.println("Inorder traversal of the given tree");
tree.inorder();

System.out.println("\nDelete 20");
tree.deleteKey(20);
System.out.println("Inorder traversal of the modified tree");
tree.inorder();

System.out.println("\nDelete 30");
tree.deleteKey(30);
System.out.println("Inorder traversal of the modified tree");
tree.inorder();

System.out.println("\nDelete 50");
tree.deleteKey(50);
System.out.println("Inorder traversal of the modified tree");
tree.inorder();
}
}

```

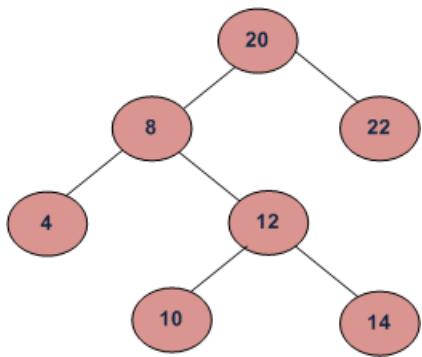
Inorder traversal of the given tree
 20 30 40 50 60 70 80
 Delete 20
 Inorder traversal of the modified tree
 30 40 50 60 70 80
 Delete 30
 Inorder traversal of the modified tree
 40 50 60 70 80
 Delete 50
 Inorder traversal of the modified tree
 40 60 70 80

Time Complexity: The worst case time complexity of delete operation is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become $O(n)$

[GATE Corner](#)[Quiz Corner](#)

Find the node with minimum value in a Binary Search Tree

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

```
#include <stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data  = data;
    node->left  = NULL;
    node->right = NULL;

    return(node);
}

/* Give a binary search tree and a number,
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). */
struct node* insert(struct node* node, int data)
{
    /* 1. If the tree is empty, return a new,
       single node */
    if (node == NULL)
        return(newNode(data));
    else
    {
        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
            node->left  = insert(node->left, data);
        else
            node->right = insert(node->right, data);

        /* return the (unchanged) node pointer */
        return node;
    }
}

/* Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
to be searched. */
int minValue(struct node* node) {
    struct node* current = node;
```

```

/* loop down to find the leftmost leaf */
while (current->left != NULL) {
    current = current->left;
}
return(current->data);
}

/* Driver program to test sameTree function*/
int main()
{
    struct node* root = NULL;
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 5);

    printf("\n Minimum value in BST is %d", minValue(root));
    getchar();
    return 0;
}

```

Time Complexity: O(n) Worst case happens for left skewed trees.

Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Inorder predecessor and successor for a given key in BST

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question:

There is BST given with root node with key part as integer only. The structure of each node is as follows:

```
struct Node
{
    int key;
    struct Node *left, *right ;
};
```

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

Following is the algorithm to reach the desired result. Its a recursive method:

Input: root node, key
output: predecessor node, successor node

1. If root is NULL
 then return
2. if key is found then
 - a. If its left subtree is not null
 Then predecessor will be the right most
 child of left subtree or left child itself.
 - b. If its right subtree is not null
 The successor will be the left most child
 of right subtree or right child itself.

```
    return
```
3. If key is smaller then root node
 set the successor as root
 search recursively into left subtree
else
 set the predecessor as root
 search recursively into right subtree

Following is C++ implementation of the above algorithm:

```
// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

// BST Node
struct Node
{
    int key;
    struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // Base case
    if (root == NULL)    return ;

    // If key is present at root
    if (root->key == key)
    {
        // the maximum value in left subtree is predecessor
        if (root->left != NULL)
        {
            Node* tmp = root->left;
            while (tmp->right)
                tmp = tmp->right;
            pre = tmp ;
        }

        // the minimum value in right subtree is successor
        if (root->right != NULL)
        {
            Node* tmp = root->right ;
            while (tmp->left)
                tmp = tmp->left ;
            suc = tmp ;
        }
    }
    return ;
}
```

```

// If key is smaller than root's key, go to left subtree
if (root->key > key)
{
    suc = root ;
    findPreSuc(root->left, pre, suc, key) ;
}
else // go to right subtree
{
    pre = root ;
    findPreSuc(root->right, pre, suc, key) ;
}

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65;      //Key to be searched in BST

    /* Let us create following BST
           50
         /   \
       30     70
      / \   / \
     20  40  60  80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    Node* pre = NULL, *suc = NULL;

    findPreSuc(root, pre, suc, key);
    if (pre != NULL)
        cout << "Predecessor is " << pre->key << endl;
    else
        cout << "No Predecessor";

    if (suc != NULL)
        cout << "Successor is " << suc->key;
    else
        cout << "No Successor";
    return 0;
}

```

Output:

Predecessor is 60
Successor is 70

A program to check if a binary tree is BST or not

A binary search tree (BST) is a node based binary tree data structure which has the following properties.

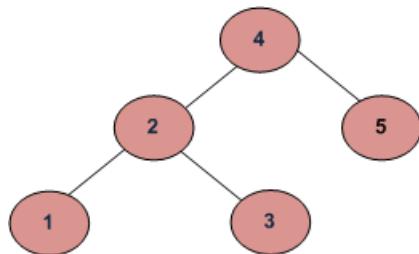
The left subtree of a node contains only nodes with keys less than the nodes key.

The right subtree of a node contains only nodes with keys greater than the nodes key.

Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

Each node (item in the tree) has a distinct key.



METHOD 1 (Simple but Wrong)

Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```
int isBST(struct node* node)
{
    if (node == NULL)
        return 1;

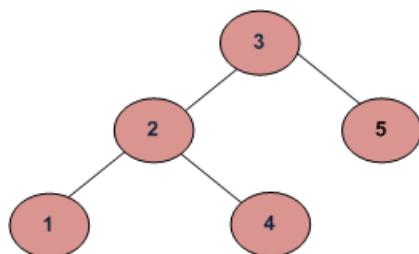
    /* false if left is > than node */
    if (node->left != NULL && node->left->data > node->data)
        return 0;

    /* false if right is < than node */
    if (node->right != NULL && node->right->data < node->data)
        return 0;

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return 0;

    /* passing all that, it's a BST */
    return 1;
}
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



METHOD 2 (Correct but not efficient)

For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
    if (node == NULL)
        return(true);

    /* false if the max of the left is > than us */
    if (node->left!=NULL && maxValue(node->left) > node->data)
```

```

    return(false);

/* false if the min of the right is <= than us */
if (node->right!=NULL && minValue(node->right) < node->data)
    return(false);

/* false if, recursively, the left or right is not a BST */
if (!isBST(node->left) || !isBST(node->right))
    return(false);

/* passing all that, it's a BST */
return(true);
}

```

It is assumed that you have helper functions `minValue()` and `maxValue()` that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)

Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function `isBSTUtil(struct node* node, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `INT_MIN` and `INT_MAX` they narrow from there.

```

/* Returns true if the given tree is a binary search tree
   (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)

```

Implementation:

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
   (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;

    /* false if this node violates the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
       tightening the min or max constraint */
    return
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values
        isBSTUtil(node->right, node->data+1, max); // Allow only distinct values
}

```

```

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(3);

    if(isBST(root))
        printf("Is BST");
    else
        printf("Not a BST");

    getchar();
    return 0;
}

```

Time Complexity: O(n)

Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

METHOD 4(Using In-Order Traversal)

Thanks to [LJW489](#) for suggesting this method.

- 1) Do In-Order Traversal of the given tree and store the result in a temp array.
- 3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to [ygos](#) for this space optimization.

```

bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;

        return isBST(root->right);
    }

    return true;
}

```

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to [this](#) post).

Sources:

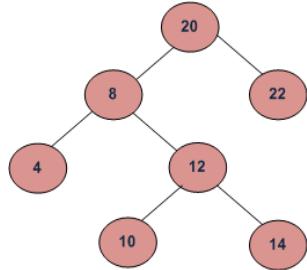
http://en.wikipedia.org/wiki/Binary_search_tree
<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Lowest Common Ancestor in a Binary Search Tree.

Given values of two nodes in a Binary Search Tree, write a c program to find the Lowest Common Ancestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

```
struct node *lca(node* root, int n1, int n2)
n1 and n2 are two given values in the tree with given root.
```



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Following is definition of LCA from [Wikipedia](#):

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))

Solutions:

If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., $n_1 < n < n_2$ or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that $n_1 < n_2$). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

```
// A recursive C program to find LCA of two nodes n1 and n2.
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
}
```

```

        return(node);
}

/* Driver program to test mirror() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->right          = newNode(22);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    getchar();
    return 0;
}

```

Output:

```

LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20

```

Time complexity of above solution is $O(h)$ where h is height of tree. Also, the above solution requires $O(h)$ extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
            root = root->right;

        else break;
    }
    return root;
}

```

See [this](#) for complete program.

You may like to see [Lowest Common Ancestor in a Binary Tree](#) also.

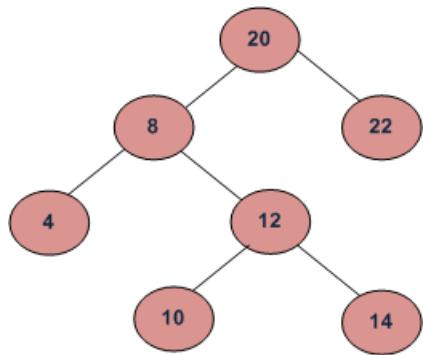
Exercise

The above functions assume that $n1$ and $n2$ both are in BST. If $n1$ and $n2$ are not present, then they may return incorrect result. Extend the above solutions to return NULL if $n1$ or $n2$ or both not present in BST.

Inorder Successor in Binary Search Tree

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inorder traversal.

In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

Method 1 (Uses Parent Pointer)

In this method, we assume that every node has parent pointer.

The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.

output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.

Go to right subtree and return the node with minimum key value in right subtree.

2) If right subtree of *node* is *NULL*, then *succ* is one of the ancestors. Do following.

Travel up using the parent pointer until you see a node which is left child of its parent. The parent of such a node is the *succ*.

Implementation

Note that the function to find InOrder Successor is highlighted (with gray background) in below code.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
};

struct node * minValue(struct node* node);

struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if( n->right != NULL )
        return minValue(n->right);

    // step 2 of the above algorithm
    struct node *p = n->parent;
    while(p != NULL && n == p->right)
    {
        n = p;
        p = p->parent;
    }
    return p;
}

/* Given a non-empty binary search tree, return the minimum data
   value found in that tree. Note that the entire tree does not need
   to be searched. */
struct node * minValue(struct node* node) {
    struct node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

```

/* loop down to find the leftmost leaf */
while (current->left != NULL) {
    current = current->left;
}
return current;
}

/* Helper function that allocates a new node with the given data and
   NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data    = data;
    node->left     = NULL;
    node->right    = NULL;
    node->parent   = NULL;

    return(node);
}

/* Give a binary search tree and a number, inserts a new node with
   the given number in the correct place in the tree. Returns the new
   root pointer which the caller should then use (the standard trick to
   avoid using reference parameters). */
struct node* insert(struct node* node, int data)
{
    /* 1. If the tree is empty, return a new,
       single node */
    if (node == NULL)
        return(newNode(data));
    else
    {
        struct node *temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
        {
            temp = insert(node->left, data);
            node->left    = temp;
            temp->parent= node;
        }
        else
        {
            temp = insert(node->right, data);
            node->right   = temp;
            temp->parent = node;
        }
    }

    /* return the (unchanged) node pointer */
    return node;
}
}

/* Driver program to test above functions*/
int main()
{
    struct node* root = NULL, *temp, *succ, *min;

    //creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    succ = inOrderSuccessor(root, temp);
    if(succ != NULL)
        printf("\n Inorder Successor of %d is %d ", temp->data, succ->data);
    else
        printf("\n Inorder Successor doesn't exist");

    getchar();
    return 0;
}

```

Output of the above program:
Inorder Successor of 14 is 20

Time Complexity: O(h) where h is height of tree.

Method 2 (Search from root)

Parent pointer is NOT needed in this algorithm. The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.

output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.

Go to right subtree and return the node with minimum key value in right subtree.

2) If right subtree of *node* is *NULL*, then start from root and us search like technique. Do following.

Travel down the tree, if a nodes data is greater than roots data then go right side, otherwise go to left side.

```
struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if( n->right != NULL )
        return minValue(n->right);

    struct node *succ = NULL;

    // Start from root and search for successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
            break;
    }

    return succ;
}
```

Thanks to [R.Srinivasan](#) for suggesting this method.

Time Complexity: O(h) where h is height of tree.

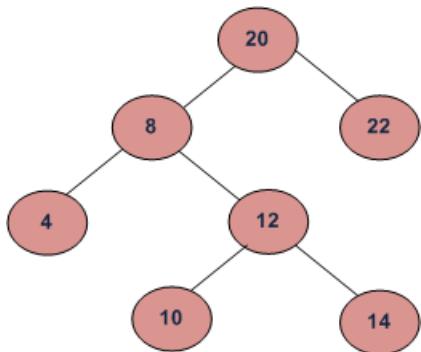
References:

<http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap13.htm>

Find k-th smallest element in BST (Order Statistics in BST)

Given root of binary search tree and K as input, find K-th smallest element in BST.

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



Method 1: Using Inorder Traversal.

Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes of tree (threaded tree avoids stack and recursion for traversal, see [this post](#)). The idea is to keep track of popped elements which participate in the order statistics. Hypothetical algorithm is provided below,

Time complexity: O(n) where n is total nodes in tree..

Algorithm:

```
/* initialization */
pCrawl = root
set initial stack element as NULL (sentinel)

/* traverse upto left extreme */
while(pCrawl is valid)
    stack.push(pCrawl)
    pCrawl = pCrawl.left

/* process other nodes */
while( pCrawl = stack.pop() is valid )
    stop if sufficient number of elements are popped.
    if( pCrawl.right is valid )
        pCrawl = pCrawl.right
        while( pCrawl is valid)
            stack.push(pCrawl)
            pCrawl = pCrawl.left
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

/* just add elements to test */
/* NOTE: A sorted array results in skewed tree */
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };

/* same alias */
typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;

    node_t* left;
    node_t* right;
};

/* simple stack that stores node addresses */
typedef struct stack_t stack_t;

/* initial element always NULL, uses as sentinel */
struct stack_t
{
    node_t* base[ARRAY_SIZE(ele) + 1];
```

```

        int      stackIndex;
};

/* pop operation of stack */
node_t *pop(stack_t *st)
{
    node_t *ret = NULL;

    if( st && st->stackIndex > 0 )
    {
        ret = st->base[st->stackIndex];
        st->stackIndex--;
    }

    return ret;
}

/* push operation of stack */
void push(stack_t *st, node_t *node)
{
    if( st )
    {
        st->stackIndex++;
        st->base[st->stackIndex] = node;
    }
}

/* Iterative insertion
   Recursion is least preferred unless we gain something */
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

```

```

/* initialize */
new_node->data    = keys[iterator];
new_node->left     = NULL;
new_node->right    = NULL;

/* insert into BST */
root = insert_node(root, new_node);
}

return root;
}

node_t *k_smallest_element_inorder(stack_t *stack, node_t *root, int k)
{
    stack_t *st = stack;
    node_t *pCrawl = root;

    /* move to left extremen (minimum) */
    while( pCrawl )
    {
        push(st, pCrawl);
        pCrawl = pCrawl->left;
    }

    /* pop off stack and process each node */
    while( pCrawl = pop(st) )
    {
        /* each pop operation emits one element
           in the order */
        if( !--k )
        {
            /* loop testing */
            st->stackIndex = 0;
            break;
        }

        /* there is right subtree */
        if( pCrawl->right )
        {
            /* push the left subtree of right subtree */
            pCrawl = pCrawl->right;
            while( pCrawl )
            {
                push(st, pCrawl);
                pCrawl = pCrawl->left;
            }
        }

        /* pop off stack and repeat */
    }
}

/* node having k-th element or NULL node */
return pCrawl;
}

/* Driver program to test above functions */
int main(void)
{
    node_t* root = NULL;
    stack_t stack = { {0}, 0 };
    node_t *kNode = NULL;

    int k = 5;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    kNode = k_smallest_element_inorder(&stack, root, k);

    if( kNode )
    {
        printf("kth smallest element for k = %d is %d", k, kNode->data);
    }
    else
    {
        printf("There is no such element");
    }

    getchar();
}

```

```

        return 0;
}

```

Method 2:Augmented Tree Data Structure.

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If $K = N + 1$, root is K-th node. If $K < N$, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If $K > N + 1$, we continue our search in the right subtree for the $(K - N - 1)$ -th smallest element. Note that we need the count of elements in left subtree only.

Time complexity: $O(h)$ where h is height of tree.

Algorithm:

```

start:
if K = root.leftElement + 1
    root node is the K th node.
    goto stop
else if K > root.leftElements
    K = K - (root.leftElements + 1)
    root = root.right
    goto start
else
    root = root.left
    goto start

stop:

```

Implementation:

```

#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;
    int lCount;

    node_t* left;
    node_t* right;
};

/* Iterative insertion
   Recursion is least preferred unless we gain something */
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* We are branching to left subtree
               increment node count */
            pTraverse->lCount++;
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */

```

```

if( !root )
{
    root = node;
}
else if( node->data < currentParent->data )
{
    /* Insert on left side */
    currentParent->left = node;
}
else
{
    /* Insert on right side */
    currentParent->right = node;
}

return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data    = keys[iterator];
        new_node->lCount  = 0;
        new_node->left    = NULL;
        new_node->right   = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

int k_smallest_element(node_t *root, int k)
{
    int ret = -1;

    if( root )
    {
        /* A crawling pointer */
        node_t *pTraverse = root;

        /* Go to k-th smallest */
        while(pTraverse)
        {
            if( (pTraverse->lCount + 1) == k )
            {
                ret = pTraverse->data;
                break;
            }
            else if( k > pTraverse->lCount )
            {
                /* There are less nodes on left subtree
                   Go to right subtree */
                k = k - (pTraverse->lCount + 1);
                pTraverse = pTraverse->right;
            }
            else
            {
                /* The node is on left subtree */
                pTraverse = pTraverse->left;
            }
        }
    }

    return ret;
}

/* Driver program to test above functions */
int main(void)
{
    /* just add elements to test */
    /* NOTE: A sorted array results in skewed tree */
}

```

```
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };
int i;
node_t* root = NULL;

/* Creating the tree given in the above diagram */
root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

/* It should print the sorted array */
for(i = 1; i <= ARRAY_SIZE(ele); i++)
{
    printf("\n kth smallest element for k = %d is %d",
           i, k_smallest_element(root, i));
}

getchar();
return 0;
}
```

Merge two BSTs with limited extra space

Given two Binary Search Trees(BST), print the elements of both BSTs in sorted form. The expected time complexity is $O(m+n)$ where m is the number of nodes in first tree and n is the number of nodes in second tree. Maximum allowed auxiliary space is $O(\text{height of the first tree} + \text{height of the second tree})$.

Examples:

```
First BST
      3
     /   \
    1     5
Second BST
      4
     /   \
    2     6
Output: 1 2 3 4 5 6
```

```
First BST
      8
     /   \
    2     10
   /
  1
Second BST
      5
     /
    3
   /
   0
Output: 0 1 2 3 5 8 10
```

Source: [Google interview question](#)

A similar question has been discussed earlier. Let us first discuss already discussed methods of the [previous post](#) which was for Balanced BSTs. The method 1 can be applied here also, but the time complexity will be $O(n^2)$ in worst case. The method 2 can also be applied here, but the extra space required will be $O(n)$ which violates the constraint given in this question. Method 3 can be applied here but the step 3 of method 3 cant be done in $O(n)$ for an unbalanced BST.

Thanks to [Kumar](#) for suggesting the following solution.

The idea is to use [iterative inorder traversal](#). We use two auxiliary stacks for two BSTs. Since we need to print the elements in sorted form, whenever we get a smaller element from any of the trees, we print it. If the element is greater, then we push it back to stack for the next iteration.

```
#include<stdio.h>
#include<stdlib.h>

// Structure of a BST Node
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//..... START OF STACK RELATED STUFF.....
// A stack node
struct snode
{
    struct node *t;
    struct snode *next;
};

// Function to add an elemnt k to stack
void push(struct snode **s, struct node *k)
{
    struct snode *tmp = (struct snode *) malloc(sizeof(struct snode));

    //perform memory check here
    tmp->t = k;
    tmp->next = *s;
    (*s) = tmp;
}

// Function to pop an element t from stack
struct node *pop(struct snode **s)
```

```

{
    struct node *t;
    struct snode *st;
    st=*s;
    (*s) = (*s)->next;
    t = st->t;
    free(st);
    return t;
}

// Function to check whether the stack is empty or not
int isEmpty(struct snode *s)
{
    if (s == NULL )
        return 1;

    return 0;
}
//..... END OF STACK RELATED STUFF......



/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* A utility function to print Inorder traversal of a Binary Tree */
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// The function to print data of two BSTs in sorted order
void merge(struct node *root1, struct node *root2)
{
    // s1 is stack to hold nodes of first BST
    struct snode *s1 = NULL;

    // Current node of first BST
    struct node *current1 = root1;

    // s2 is stack to hold nodes of second BST
    struct snode *s2 = NULL;

    // Current node of second BST
    struct node *current2 = root2;

    // If first BST is empty, then output is inorder
    // traversal of second BST
    if (root1 == NULL)
    {
        inorder(root2);
        return;
    }
    // If second BST is empty, then output is inorder
    // traversal of first BST
    if (root2 == NULL)
    {
        inorder(root1);
        return ;
    }

    // Run the loop while there are nodes not yet printed.
    // The nodes may be in stack(explored, but not printed)
    // or may be not yet explored
    while (current1 != NULL || !isEmpty(s1) ||
           current2 != NULL || !isEmpty(s2))
    {
        // Following steps follow iterative Inorder Traversal
        if (current1 != NULL || current2 != NULL )
        {

```

```

// Reach the leftmost node of both BSTs and push ancestors of
// leftmost nodes to stack s1 and s2 respectively
if (current1 != NULL)
{
    push(&s1, current1);
    current1 = current1->left;
}
if (current2 != NULL)
{
    push(&s2, current2);
    current2 = current2->left;
}

}
else
{
    // If we reach a NULL node and either of the stacks is empty,
    // then one tree is exhausted, print the other tree
    if (isEmpty(s1))
    {
        while (!isEmpty(s2))
        {
            current2 = pop (&s2);
            current2->left = NULL;
            inorder(current2);
        }
        return ;
    }
    if (isEmpty(s2))
    {
        while (!isEmpty(s1))
        {
            current1 = pop (&s1);
            current1->left = NULL;
            inorder(current1);
        }
        return ;
    }

    // Pop an element from both stacks and compare the
    // popped elements
    current1 = pop(&s1);
    current2 = pop(&s2);

    // If element of first tree is smaller, then print it
    // and push the right subtree. If the element is larger,
    // then we push it back to the corresponding stack.
    if (current1->data < current2->data)
    {
        printf("%d ", current1->data);
        current1 = current1->right;
        push(&s2, current2);
        current2 = NULL;
    }
    else
    {
        printf("%d ", current2->data);
        current2 = current2->right;
        push(&s1, current1);
        current1 = NULL;
    }
}
}

/*
 * Driver program to test above functions */
int main()
{
    struct node  *root1 = NULL, *root2 = NULL;

    /* Let us create the following tree as first tree
       3
      / \
     1   5
    */
    root1 = newNode(3);
    root1->left = newNode(1);
    root1->right = newNode(5);

    /* Let us create the following tree as second tree
       4
      / \
     2   5
    */

```

```
    /   \
   2     6
 */
root2 = newNode(4);
root2->left = newNode(2);
root2->right = newNode(6);

// Print sorted nodes of both trees
merge(root1, root2);

return 0;
}
```

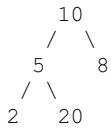
Time Complexity: O(m+n)

Auxiliary Space: O(height of the first tree + height of the second tree)

Two nodes of a BST are swapped, correct the BST

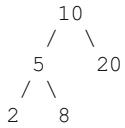
Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

Input Tree:



In the above tree, nodes 20 and 8 must be swapped to fix the tree.

Following is the output tree



The inorder traversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiliary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is $O(n\log n)$ and auxiliary space needed is $O(n)$.

We can solve this in $O(n)$ time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.

For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two nodes values.

2. The swapped nodes are adjacent in the inorder traversal of BST.

For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 5 8 7 10 15 20 25

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve? We will maintain three pointers, first, middle and last. When we find the first point where current node value is smaller than previous node value, we update the first with the previous node & middle with the current node. When we find the second point where current node value is smaller than previous node value, we update the last with the current node. In case #2, we will never find the second point. So, last pointer will not be updated. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.

Following is C implementation of the given code.

```
// Two nodes in the BST's swapped, correct the BST.
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
}
```

```

node->left = NULL;
node->right = NULL;
return(node);
}

// This function does inorder traversal to find out the two swapped nodes.
// It sets three pointers, first, middle and last. If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                     struct node** middle, struct node** last,
                     struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if ( *prev && root->data < (*prev)->data )
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

// A function to fix a given BST where two nodes are swapped. This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the pointers to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inorder traversal */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    /*      6
           /   \
          10   2
         / \   / \
}

```

```

1   3 7 12
10 and 2 are swapped
*/
struct node *root = newNode(6);
root->left      = newNode(10);
root->right     = newNode(2);
root->left->left = newNode(1);
root->left->right = newNode(3);
root->right->right = newNode(12);
root->right->left = newNode(7);

printf("Inorder Traversal of the original tree \n");
printInorder(root);

correctBST(root);

printf("\nInorder Traversal of the fixed tree \n");
printInorder(root);

return 0;
}

```

Output:

```

Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12

```

Time Complexity: O(n)

See [this](#) for different test cases of the above code.

Floor and Ceil from a BST

There are numerous applications we need to find floor (ceil) value of a key in a binary search tree or sorted array. For example, consider designing memory management system in which free nodes are arranged in BST. Find best fit for the input request.

Ceil Value Node: Node with smallest data larger than or equal to key value.

Imagine we are moving down the tree, and assume we are root node. The comparison yields three possibilities,

A) Root data is equal to key. We are done, root data is ceil value.

B) Root data < key value, certainly the ceil value can't be in left subtree. Proceed to search on right subtree as reduced problem instance.

C) Root data > key value, the ceil value *may be* in left subtree. We may find a node with larger data than key value in left subtree, if not the root itself will be ceil node.

Here is code in C for ceil value.

```
// Program to find ceil of a given value in BST
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers.*/
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(node *root, int input)
{
    // Base case
    if( root == NULL )
        return -1;

    // We found equal key
    if( root->key == input )
        return root->key;

    // If root's key is smaller, ceil must be in right subtree
    if( root->key < input )
        return Ceil(root->right, input);

    // Else, either left subtree or root has the ceil value
    int ceil = Ceil(root->left, input);
    return (ceil >= input) ? ceil : root->key;
}

// Driver program to test above function
int main()
{
    node *root = newNode(8);

    root->left = newNode(4);
    root->right = newNode(12);

    root->left->left = newNode(2);
    root->left->right = newNode(6);

    root->right->left = newNode(10);
    root->right->right = newNode(14);

    for(int i = 0; i < 16; i++)
        printf("%d %d\n", i, Ceil(root, i));
}
```

```
    return 0;  
}
```

Output:

```
0 2  
1 2  
2 2  
3 4  
4 4  
5 6  
6 6  
7 8  
8 8  
9 10  
10 10  
11 12  
12 12  
13 14  
14 14  
15 -1
```

Exercise:

1. Modify above code to find floor value of input key in a binary search tree.
2. Write neat algorithm to find floor and ceil values in a sorted array. Ensure to handle all possible boundary conditions.

In-place conversion of Sorted DLL to Balanced BST

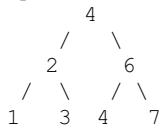
Given a Doubly Linked List which has data members sorted in ascending order. Construct a [Balanced Binary Search Tree](#) which has same data members as the given Doubly Linked List. The tree must be constructed in-place (No new node should be allocated for tree conversion)

Examples:

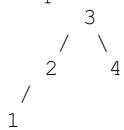
Input: Doubly Linked List 1 <--> 2 <--> 3
Output: A Balanced BST



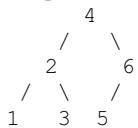
Input: Doubly Linked List 1 <--> 2 <--> 3 <--> 4 <--> 5 <--> 6 <--> 7
Output: A Balanced BST



Input: Doubly Linked List 1 <--> 2 <--> 3 <--> 4
Output: A Balanced BST



Input: Doubly Linked List 1 <--> 2 <--> 3 <--> 4 <--> 5 <--> 6
Output: A Balanced BST



The Doubly Linked List conversion is very much similar to [this Singly Linked List problem](#) and the method 1 is exactly same as the method 1 of [previous post](#). Method 2 is also almost same. The only difference in method 2 is, instead of allocating new nodes for BST, we reuse same DLL nodes. We use prev pointer as left and next pointer as right.

Method 1 (Simple)

Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

- 1) Get the Middle of the linked list and make it root.
- 2) Recursively do same for left half and right half.
 - a) Get the middle of left half and make it left child of the root created in step 1.
 - b) Get the middle of right half and make it right child of the root created in step 1.

Time complexity: O(nLogn) where n is the number of nodes in Linked List.

Method 2 (Tricky)

The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Doubly Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left $n/2$ nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.

While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call. Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

```
#include<stdio.h>
#include<stdlib.h>

/* A Doubly Linked List node that will also be used as a tree node */
struct Node
{
    int data;
    // For tree, next pointer can be used as right subtree pointer
    struct Node* next;
    // For tree, prev pointer can be used as left subtree pointer
};
```

```

    struct Node* prev;
};

// A utility function to count nodes in a Linked List
int countNodes(struct Node *head);

struct Node* sortedListToBSTRecur(struct Node **head_ref, int n);

/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct Node* sortedListToBST(struct Node *head)
{
    /*Count the number of nodes in Linked List */
    int n = countNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
   head_ref --> Pointer to pointer to head node of Doubly linked list
   n --> No. of nodes in the Doubly Linked List */
struct Node* sortedListToBSTRecur(struct Node **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct Node *left = sortedListToBSTRecur(head_ref, n/2);

    /* head_ref now refers to middle node, make middle node as root of BST*/
    struct Node *root = *head_ref;

    // Set pointer to left subtree
    root->prev = left;

    /*Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
       The number of nodes in right subtree is total nodes - nodes in
       left subtree - 1 (for root) */
    root->next = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a given Linked List */
int countNodes(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* Function to insert a node at the begining of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the begining,
       prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */

```

```

if((*head_ref) != NULL)
    (*head_ref)->prev = new_node ;

/* move the head to point to the new node */
(*head_ref)      = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->prev);
    preOrder(node->next);
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
     * Created linked list will be 7->6->5->4->3->2->1 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given Linked List ");
    printList(head);

    /* Convert List to BST */
    struct Node *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

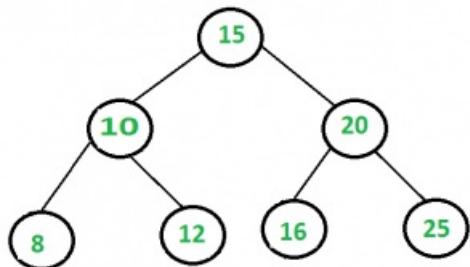
    return 0;
}

```

Time Complexity: O(n)

Find a pair with given sum in a Balanced BST

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is $O(n)$ and only $O(\log n)$ extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always $O(\log n)$.



This problem is mainly extension of the [previous post](#). Here we are not allowed to modify the BST.

The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X. The time complexity of this solution will be $O(n^2)$.

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in $O(n)$ time (See [this](#) for details). This solution works in $O(n)$ time, but requires $O(n)$ auxiliary space.

A **space optimized solution** is discussed in [previous post](#). The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in $O(n)$ time. This solution takes $O(n)$ time and $O(\log n)$ extra space, but it modifies the given BST.

The **solution discussed below takes $O(n)$ time, $O(\log n)$ space and doesn't modify BST**. The idea is same as finding the pair in sorted array (See method 1 of [this](#) for details). We traverse BST in Normal Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node. In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum. If the sum is same as target sum, we return true. If the sum is more than target sum, we move to next node in reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false. Following is C++ implementation of this approach.

```
/* In a balanced binary search tree isPairPresent two element which sums to
   a given value time O(n) space O(logn) */
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// A BST node
struct node
{
    int val;
    struct node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct node* *array;
};

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct node**) malloc(stack->size * sizeof(struct node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{   return stack->top - 1 == stack->size;  }

int isEmpty(struct Stack* stack)
```

```

    { return stack->top == -1; }

void push(struct Stack* stack, struct node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// Returns true if a pair with target sum exists in BST, otherwise false
bool isPairPresent(struct node *root, int target)
{
    // Create two stacks. s1 is used for normal inorder traversal
    // and s2 is used for reverse inorder traversal
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE, we can find the tree size and
    // fix stack size as O(Logn) for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

    // done1, val1 and curr1 are used for normal inorder traversal using s1
    // done2, val2 and curr2 are used for reverse inorder traversal using s2
    bool done1 = false, done2 = false;
    int val1 = 0, val2 = 0;
    struct node *curr1 = root, *curr2 = root;

    // The loop will break when we either find a pair or one of the two
    // traversals is complete
    while (1)
    {
        // Find next node in normal Inorder traversal. See following post
        // http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
        while (done1 == false)
        {
            if (curr1 != NULL)
            {
                push(s1, curr1);
                curr1 = curr1->left;
            }
            else
            {
                if (isEmpty(s1))
                    done1 = 1;
                else
                {
                    curr1 = pop(s1);
                    val1 = curr1->val;
                    curr1 = curr1->right;
                    done1 = 1;
                }
            }
        }

        // Find next node in REVERSE Inorder traversal. The only
        // difference between above and below loop is, in below loop
        // right subtree is traversed before left subtree
        while (done2 == false)
        {
            if (curr2 != NULL)
            {
                push(s2, curr2);
                curr2 = curr2->right;
            }
            else
            {
                if (isEmpty(s2))
                    done2 = 1;
                else
                {
                    curr2 = pop(s2);
                    val2 = curr2->val;
                    curr2 = curr2->left;
                    done2 = 1;
                }
            }
        }

        if (val1 + val2 == target)
            return 1;
    }
}

```

```

        }

    }

// If we find a pair, then print the pair and return. The first
// condition makes sure that two same values are not added
if ((val1 != val2) && (val1 + val2) == target)
{
    printf("\n Pair Found: %d + %d = %d\n", val1, val2, target);
    return true;
}

// If sum of current values is smaller, then move to next node in
// normal inorder traversal
else if ((val1 + val2) < target)
    done1 = false;

// If sum of current values is greater, then move to next node in
// reverse inorder traversal
else if ((val1 + val2) > target)
    done2 = false;

// If any of the inorder traversals is over, then there is no pair
// so return false
if (val1 >= val2)
    return false;
}

}

// A utility function to create BST node
struct node * NewNode(int val)
{
    struct node *tmp = (struct node *)malloc(sizeof(struct node));
    tmp->val = val;
    tmp->right = tmp->left =NULL;
    return tmp;
}

// Driver program to test above functions
int main()
{
    /*
        15
       /   \
      10   20
     / \   / \
    8  12 16  25 */
    struct node *root = NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        printf("\n No such values are found\n");

    getchar();
    return 0;
}

```

Output:

Pair Found: 8 + 25 = 33

Total number of possible Binary Search Trees with n keys

Total number of possible Binary Search Trees with n different keys = [Catalan number C_n](#) = $(2n)!/(n+1)!*n!$

See references for proof and examples.

References:

http://en.wikipedia.org/wiki/Catalan_number

Merge Two Balanced Binary Search Trees

You are given two balanced binary search trees e.g., AVL or Red Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take $O(m+n)$ time.

In the following solutions, it is assumed that sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree (See [this](#)).

Method 1 (Insert elements of first tree to second)

Take all elements of first BST one by one, and insert them  the second BST. Inserting an element to a self balancing BST takes $\log n$ time (See [this](#)) where n is size of the BST. So time complexity of this method is $\log(m) + \log(n+1) \log(m+n-1)$. The value of this expression will be between $m\log m$ and $m\log(m+n-1)$. As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals)

- 1) Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes $O(m)$ time.
- 2) Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes $O(n)$ time.
- 3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size $m + n$. This step takes $O(m+n)$ time.
- 4) Construct a balanced tree from the merged array using the technique discussed in [this](#) post. This step takes $O(m+n)$ time.

Time complexity of this method is $O(m+n)$ which is better than method 1. This method takes $O(m+n)$ time even if the input BSTs are not balanced. Following is C++ implementation of this method.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

// A utility function to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n);

// A helper function that stores inorder traversal of a tree in inorder array
void storeInorder(struct node* node, int inorder[], int *index_ptr);

/* A function that constructs Balanced Binary Search Tree from a sorted array
   See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end);

/* This function merges two balanced BSTs with roots as root1 and root2.
   m and n are the sizes of the trees respectively */
struct node* mergeTrees(struct node *root1, struct node *root2, int m, int n)
{
    // Store inorder traversal of first tree in an array arr1[]
    int *arr1 = new int[m];
    int i = 0;
    storeInorder(root1, arr1, &i);

    // Store inorder traversal of second tree in another array arr2[]
    int *arr2 = new int[n];
    int j = 0;
    storeInorder(root2, arr2, &j);

    // Merge the two sorted array into one
    int *mergedArr = merge(arr1, arr2, m, n);

    // Construct a tree from the merged array and return root of the tree
    return sortedArrayToBST(mergedArr, 0, m+n-1);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}
```

```

// A utility function to print inorder traversal of a given binary tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

// A utility function to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n)
{
    // mergedArr[] is going to contain result
    int *mergedArr = new int[m + n];
    int i = 0, j = 0, k = 0;

    // Traverse through both arrays
    while (i < m && j < n)
    {
        // Pick the smaller element and put it in mergedArr
        if (arr1[i] < arr2[j])
        {
            mergedArr[k] = arr1[i];
            i++;
        }
        else
        {
            mergedArr[k] = arr2[j];
            j++;
        }
        k++;
    }

    // If there are more elements in first array
    while (i < m)
    {
        mergedArr[k] = arr1[i];
        i++; k++;
    }

    // If there are more elements in second array
    while (j < n)
    {
        mergedArr[k] = arr2[j];
        j++; k++;
    }

    return mergedArr;
}

// A helper function that stores inorder traversal of a tree rooted with node
void storeInorder(struct node* node, int inorder[], int *index_ptr)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    storeInorder(node->left, inorder, index_ptr);

    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry

    /* now recur on right child */
    storeInorder(node->right, inorder, index_ptr);
}

/* A function that constructs Balanced Binary Search Tree from a sorted array
See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
        return NULL;
}

```

```

/* Get the middle element and make it root */
int mid = (start + end)/2;
struct node *root = newNode(arr[mid]);

/* Recursively construct the left subtree and make it
   left child of root */
root->left = sortedArrayToBST(arr, start, mid-1);

/* Recursively construct the right subtree and make it
   right child of root */
root->right = sortedArrayToBST(arr, mid+1, end);

return root;
}

/* Driver program to test above functions*/
int main()
{
    /* Create following tree as first balanced BST
       100
       / \
      50   300
      / \
     20   70
    */
    struct node *root1 = newNode(100);
    root1->left = newNode(50);
    root1->right = newNode(300);
    root1->left->left = newNode(20);
    root1->left->right = newNode(70);

    /* Create following tree as second balanced BST
       80
       / \
      40   120
    */
    struct node *root2 = newNode(80);
    root2->left = newNode(40);
    root2->right = newNode(120);

    struct node *mergedTree = mergeTrees(root1, root2, 5, 3);

    printf ("Following is Inorder traversal of the merged tree \n");
    printInorder(mergedTree);

    getchar();
    return 0;
}

```

Output:

Following is Inorder traversal of the merged tree
20 40 50 70 80 100 120 300

Method 3 (In-Place Merge using DLL)

We can use a Doubly Linked List to merge trees in place. Following are the steps.

- 1) Convert the given two Binary Search Trees into doubly linked list in place (Refer [this post](#) for this step).
- 2) Merge the two sorted Linked Lists (Refer [this post](#) for this step).
- 3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer [this post](#) for this step)

Time complexity of this method is also O(m+n) and this method does conversion in place.

Thanks to [Dheeraj](#) and [Ronzi](#) for suggesting this method.

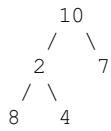
Binary Tree to Binary Search Tree Conversion

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

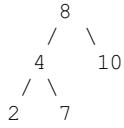
Examples.

Example 1

Input:

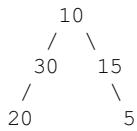


Output:

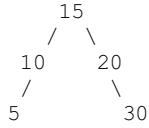


Example 2

Input:



Output:



Solution

Following is a 3 step solution for converting Binary tree to Binary Search Tree.

- 1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes O(n) time.
- 2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in $O(n \log n)$ time using Heap Sort or Merge Sort.
- 3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes O(n) time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

```
/* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* A helper function that stores inorder traversal of a tree rooted
   with node */
void storeInorder (struct node* node, int inorder[], int *index_ptr)
{
    // Base Case
    if (node == NULL)
        return;

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry

    /* finally store the right subtree */
    storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
```

```

{
    if (root == NULL)
        return 0;
    return countNodes (root->left) +
           countNodes (root->right) + 1;
}

// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
   This functon basically does Inorder traversal of Binary Tree and
   one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}

// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if(root == NULL)
        return;

    /* Count the number of nodes in Binary Tree so that
       we know the size of temporary array to be created */
    int n = countNodes (root);

    // Create a temp array arr[] and store inorder traversal of tree in arr[]
    int *arr = new int[n];
    int i = 0;
    storeInorder (root, arr, &i);

    // Sort the array using library function for quick sort
    qsort (arr, n, sizeof(arr[0]), compare);

    // Copy array elements back to Binary Tree
    i = 0;
    arrayToBST (arr, root, &i);

    // delete dynamically allocated memory to avoid meory leak
    delete [] arr;
}

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* Utility function to print inorder traversal of Binary Tree */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);
}

```

```

/* now recur on right child */
printInorder (node->right);
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure
       10
      / \
     30   15
    /   \
   20   5  */
    root = newNode(10);
    root->left = newNode(30);
    root->right = newNode(15);
    root->left->left = newNode(20);
    root->right->right = newNode(5);

    // convert Binary Tree to BST
    binaryTreeToBST (root);

    printf("Following is Inorder Traversal of the converted BST: \n");
    printInorder (root);

    return 0;
}

```

Output:

Following is Inorder Traversal of the converted BST:
5 10 15 20 30

We will be covering another method for this problem which converts the tree using $O(\text{height of tree})$ extra space.

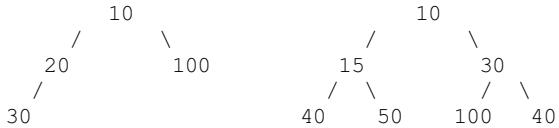
Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) Its a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

Examples of Min Heap:



Applications of Heaps:

1) [Heap Sort](#): Heap Sort uses Binary Heap to sort an array in $O(n \log n)$ time.

2) Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports `insert()`, `delete()` and `extractmax()`, `decreaseKey()` operations in $O(\log n)$ time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.

3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like [Dijkstras Shortest Path](#) and [Prims Minimum Spanning Tree](#).

4) Many problems can be efficiently solved using Heaps. See following for example.

- a) [Kth Largest Element in an array](#).
- b) [Sort an almost sorted array](#)/
- c) [Merge K Sorted Arrays](#).

Operations on Min Heap:

1) `getMin()`: It returns the root element of Min Heap. Time Complexity of this operation is $O(1)$.

2) `extractMin()`: Removes the minimum element from Min Heap. Time Complexity of this Operation is $O(\log n)$ as this operation needs to maintain the heap property (by calling `heapify()`) after removing root.

3) `decreaseKey()`: Decreases value of key. Time complexity of this operation is $O(\log n)$. If the decreases key value of a node is greater than parent of the node, then we dont need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

4) `insert()`: Inserting a new key takes $O(\log n)$ time. We add a new key at the end of the tree. IF new key is greater than its parent, then we dont need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

5) `delete()`: Deleting a key also takes $O(\log n)$ time. We replace the key to be deleted with minus infinite by calling `decreaseKey()`. After `decreaseKey()`, the minus infinite value must reach root, so we call `extractMin()` to remove key.

Following is C++ implementation of basic heap operations.

```
// A C++ program to demonstrate common Binary Heap Operations
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    // Constructor
    MinHeap(int capacity);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }
```

```

// to get index of right child of node at index i
int right(int i) { return (2*i + 2); }

// to extract the root which is the minimum element
int extractMin();

// Decreases key value of key at index i to new_val
void decreaseKey(int i, int new_val);

// Returns the minimum key (key at root) from min heap
int getMin() { return harr[0]; }

// Deletes a key stored at index i
void deleteKey(int i);

// Inserts a new key 'k'
void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// Decreases value of key at index 'i' to new_val. It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}

```

```

// This function deletes key at index i. It first reduces value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
    decreaseKey(i, INT_MIN);
    extractMin();
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Driver program to test above functions
int main()
{
    MinHeap h(11);
    h.insertKey(3);
    h.insertKey(2);
    h.deleteKey(1);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);
    cout << h.extractMin() << " ";
    cout << h.getMin() << " ";
    h.decreaseKey(2, 1);
    cout << h.getMin();
    return 0;
}

```

Output:

2 4 1

[GATE Corner](#)[Quiz Corner](#)

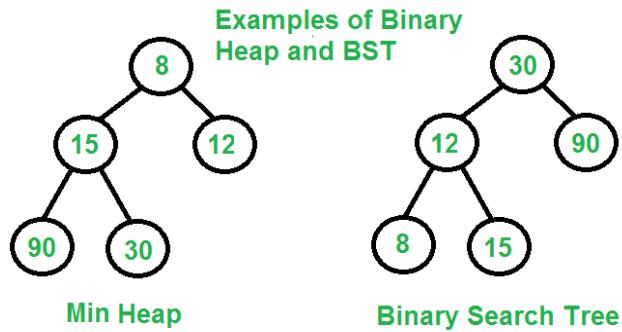
Why is Binary Heap Preferred over BST for Priority Queue?

A typical [Priority Queue](#) requires following operations to be efficient.

1. Get Top Priority Element (Get minimum or maximum)
2. Insert an element
3. Remove top priority element
4. Decrease Key

A [Binary Heap](#) supports above operations with following time complexities:

1. O(1)
2. O(Logn)
3. O(Logn)
4. O(Logn)



A Self Balancing Binary Search Tree like [AVL Tree](#), [Red-Black Tree](#), etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally O(1), but can be easily implemented in O(1) by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding inorder predecessor or successor.
2. Inserting an element is naturally O(Logn)
3. Removing maximum or minimum are also O(Logn)
4. Decrease key can be done in O(Logn) by doing a deletion followed by insertion. See [this](#) for details.

So why is Binary Heap Preferred for Priority Queue?

- Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.
- Although operations are of same time complexity, constants in Binary Search Tree are higher.
- We can build a Binary Heap in O(n) time. Self Balancing BSTs require O(nLogn) time to construct.
- Binary Heap doesn't require extra space for pointers.
- Binary Heap is easier to implement.
- There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in O(1) time

Is Binary Heap always better?

Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.

- Searching an element in self-balancing BST is O(Logn) which is O(n) in Binary Heap.
- We can print all elements of BST in sorted order in O(n) time, but Binary Heap requires O(nLogn) time.
- [Floor and ceil](#) can be found in O(Logn) time.
- [Kth largest/smallest element](#) be found in O(Logn) time by augmenting tree with an additional field.

Binomial Heap

The main application of [Binary Heap](#) is as implement priority queue. Binomial Heap is an extension of [Binary Heap](#) that provides faster union or merge operation together with other operations provided by Binary Heap.

A *Binomial Heap* is a collection of *Binomial Trees*

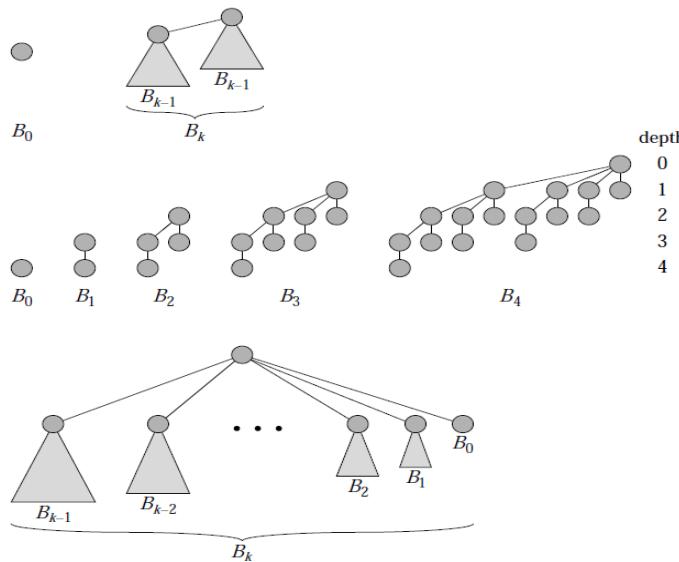
What is a Binomial Tree?

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1, and making one as leftmost child of other.

A Binomial Tree of order k has following properties.

- It has exactly 2^k nodes.
- It has depth as k.
- There are exactly kC_i nodes at depth i for $i = 0, 1, \dots, k$.
- The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

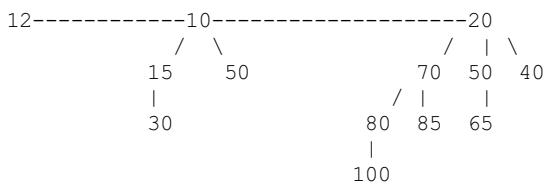
The following diagram is taken from 2nd Edition of [CLRS book](#).



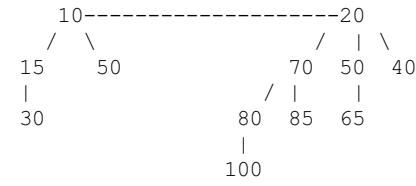
Binomial Heap:

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at-most one Binomial Tree of any degree.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2 and 3 from left to right.



A Binomial Heap with 12 nodes. It is a collection of 2 Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps

A Binomial Heap with n nodes has number of Binomial Trees equal to the number of set bits in Binary representation of n. For example let n be 13, there 3 set bits in binary representation of n (00001101), hence 3 Binomial Trees. We can also relate degree of these Binomial Trees with positions of set bits. With this relation we can conclude that there are $O(\log n)$ Binomial Trees in a Binomial Heap with n nodes.

Operations of Binomial Heap:

The main operation in Binomial Heap is union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

1) insert(H, k): Inserts a key k to Binomial Heap H. This operation first creates a Binomial Heap with single key k, then calls union on H and the new Binomial heap.

2) getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.

3) extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().

5) decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parents key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn).

Union operation in Binomial Heap:

Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.

1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

2) After the simple merge, we need to make sure that there is at-most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of same order. We traverse the list of merged roots, we keep track of three pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots.

Case 1: Orders of x and next-x are not same, we simply move ahead.

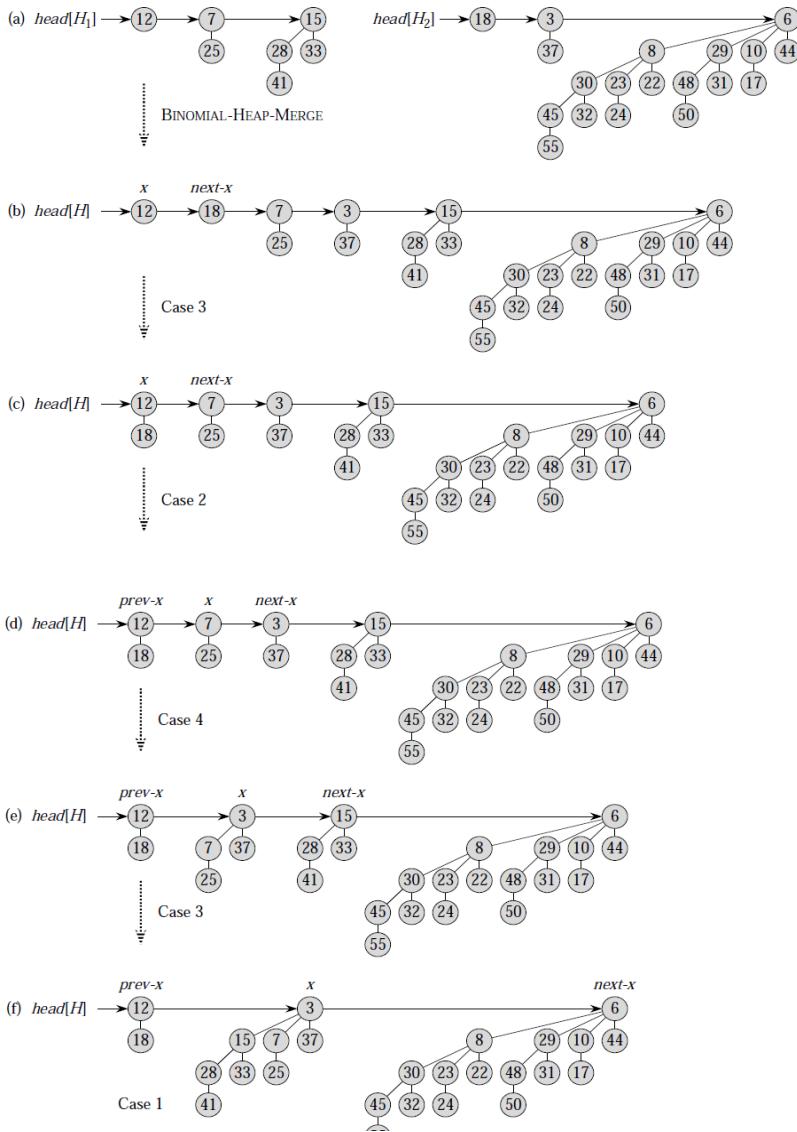
In following 3 cases orders of x and next-x are same.

Case 2: If order of next-next-x is also same, move ahead.

Case 3: If key of x is smaller than or equal to key of next-x, then make next-x as a child of x by linking it with x.

Case 4: If key of x is greater, then make x as child of next.

The following diagram is taken from 2nd Edition of [CLRS book](#).



How to represent Binomial Heap?

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in and extractMin() and delete()). The idea is to represent Binomial Trees as leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

We will soon be discussing implementation of Binomial Heap.

Sources:

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](#))

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by $2 * I + 1$ and right child by $2 * I + 2$.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1.
3. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
           4(0)
          /   \
        10(1)   3(2)
          /   \
        5(3)   1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
        4(0)
       /   \
     10(1)   3(2)
      /   \
    5(3)   1(4)
```

Applying heapify procedure to index 0:

```
        10(0)
       /   \
     5(1)   3(2)
      /   \
    4(3)   1(4)
```

The heapify procedure calls itself recursively to build heap in top down manner.

```
// C implementation of Heap Sort
#include <stdio.h>
#include <stdlib.h>

// A heap has current size and array of elements
struct MaxHeap
{
    int size;
    int* array;
};

// A utility function to swap to integers
void swap(int* a, int* b) { int t = *a; *a = *b; *b = t; }

// The main function to heapify a Max Heap. The function
// assumes that everything under given root (element at
// index idx) is already heapified
void maxHeapify(struct MaxHeap* maxHeap, int idx)
{
    int largest = idx; // Initialize largest as root
    int left = (idx << 1) + 1; // left = 2*idx + 1
```

```

int right = (idx + 1) << 1; // right = 2*idx + 2

// See if left child of root exists and is greater than
// root
if (left < maxHeap->size &&
    maxHeap->array[left] > maxHeap->array[largest])
    largest = left;

// See if right child of root exists and is greater than
// the largest so far
if (right < maxHeap->size &&
    maxHeap->array[right] > maxHeap->array[largest])
    largest = right;

// Change root, if needed
if (largest != idx)
{
    swap(&maxHeap->array[largest], &maxHeap->array[idx]);
    maxHeapify(maxHeap, largest);
}
}

// A utility function to create a max heap of given capacity
struct MaxHeap* createAndBuildHeap(int *array, int size)
{
    int i;
    struct MaxHeap* maxHeap =
        (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = size; // initialize size of heap
    maxHeap->array = array; // Assign address of first element of array

    // Start from bottommost and rightmost internal mode and heapify all
    // internal modes in bottom up way
    for (i = (maxHeap->size - 2) / 2; i >= 0; --i)
        maxHeapify(maxHeap, i);
    return maxHeap;
}

// The main function to sort an array of given size
void heapSort(int* array, int size)
{
    // Build a heap from the input data.
    struct MaxHeap* maxHeap = createAndBuildHeap(array, size);

    // Repeat following steps while heap size is greater than 1.
    // The last element in max heap will be the minimum element
    while (maxHeap->size > 1)
    {
        // The largest item in Heap is stored at the root. Replace
        // it with the last item of the heap followed by reducing the
        // size of heap by 1.
        swap(&maxHeap->array[0], &maxHeap->array[maxHeap->size - 1]);
        --maxHeap->size; // Reduce heap size

        // Finally, heapify the root of tree.
        maxHeapify(maxHeap, 0);
    }
}

// A utility function to print a given array of given size
void printArray(int* arr, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        printf("%d ", arr[i]);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, size);

    heapSort(arr, size);

    printf("\nSorted array is \n");
    printArray(arr, size);
    return 0;
}

```

}

Output:

```
Given array is  
12 11 13 5 6 7  
Sorted array is  
5 6 7 11 12 13
```

Notes:

Heap sort is an in-place algorithm.

Its typical implementation is not stable, but can be made stable (See [this](#))

Time Complexity: Time complexity of heapify is $O(\log n)$. Time complexity of createAndBuildHeap() is $O(n)$ and overall time complexity of Heap Sort is $O(n\log n)$.

Applications of HeapSort

1. [Sort a nearly sorted \(or K sorted\) array](#)
2. [k largest\(or smallest\) elements in an array](#)

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See [Applications of Heap Data Structure](#)

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- [Selection Sort](#)
- [Bubble Sort](#)
- [Insertion Sort](#)
- [Merge Sort](#)
- [QuickSort](#)
- [Radix Sort](#)
- [Counting Sort](#)
- [Bucket Sort](#)
- [ShellSort](#)

k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Method 1 (Use Bubble k times)

Thanks to Shailendra for suggesting this approach.

- 1) Modify [Bubble Sort](#) to run the outer loop at most k times.
- 2) Print the last k elements of the array obtained in step 1.

Time Complexity: $O(nk)$

Like Bubble sort, other sorting algorithms like [Selection Sort](#) can also be modified to get the k largest elements.

Method 2 (Use temporary array)

K largest elements from arr[0..n-1]

- 1) Store the first k elements in a temporary array temp[0..k-1].
- 2) Find the smallest element in temp[], let the smallest element be *min*.
- 3) For each element *x* in arr[k] to arr[n-1]
If *x* is greater than the *min* then remove *min* from temp[] and insert *x*.
- 4) Print final k elements of temp[]

Time Complexity: $O((n-k)*k)$. If we want the output sorted then $O((n-k)*k + k\log k)$

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)

- 1) Sort the elements in descending order in $O(n\log n)$
- 2) Print the first k numbers of the sorted array $O(k)$.

Time complexity: $O(n\log n)$

Method 4 (Use Max Heap)

- 1) Build a Max Heap tree in $O(n)$
- 2) Use [Extract Max](#) k times to get k maximum elements from the Max Heap $O(k\log n)$

Time complexity: $O(n + k\log n)$

Method 5(Use Order Statistics)

- 1) Use order statistic algorithm to find the kth largest element. Please [see the topic selection in worst-case linear time](#) $O(n)$
- 2) Use [QuickSort](#) Partition algorithm to partition around the kth largest number $O(n)$.
- 3) Sort the k-1 elements (elements greater than the kth largest element) $O(k\log k)$. This step is needed only if sorted output is required.

Time complexity: $O(n)$ if we dont need the sorted output, otherwise $O(n+k\log k)$

Thanks to [Shilpi](#) for suggesting the first two approaches.

Method 6 (Use Min Heap)

This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.

Thanks to [geek4u](#) for suggesting this method.

- 1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. $O(k)$
- 2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.
 - a) If the element is greater than the root then make it root and call [heapify](#) for MH
 - b) Else ignore it.

// The step 2 is $O((n-k)*\log k)$
- 3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: $O(k + (n-k)\log k)$ without sorted output. If sorted output is needed then $O(k + (n-k)\log k + k\log k)$

All of the above methods can also be used to find the kth largest (or smallest) element.

References:

http://en.wikipedia.org/wiki/Selection_algorithm

Asked by [geek4u](#)

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in $O(n \log k)$ time. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: [Nearly sorted algorithm](#)

We can use **Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int A[], int size)
{
    int i, key, j;
    for (i = 1; i < size; i++)
    {
        key = A[i];
        j = i-1;

        /* Move elements of A[0..i-1], that are greater than key, to one
           position ahead of their current position.
           This loop will run at most k times */
        while (j >= 0 && A[j] > key)
        {
            A[j+1] = A[j];
            j = j-1;
        }
        A[j+1] = key;
    }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall complexity will be $O(nk)$

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.

- 1) Create a Min Heap of size $k+1$ with first $k+1$ elements. This will take $O(k)$ time (See [this GFact](#))
- 2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Log k time. So overall complexity will be $O(k) + O((n-k)*\log k)$

```
#include<iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// Given an array of size n, where every element is k away from its target
// position, sorts the array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i<=k && i<n; i++) // i < n condition is needed when k > n
```

```

        harr[i] = arr[i];
MinHeap hp(harr, k+1);

// i is index for remaining elements in arr[] and ti
// is target index of for cuurent minimum element in
// Min Heapm 'hp'.
for(int i = k+1, ti = 0; ti < n; i++, ti++)
{
    // If there are remaining elements, then place
    // root of heap at target index and add arr[i]
    // to Min Heap
    if (i < n)
        arr[ti] = hp.replaceMin(arr[i]);

    // Otherwise place root at its target index and
    // reduce heap size
    else
        arr[ti] = hp.extractMin();
}
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

```

```

    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = {2, 6, 3, 12, 56, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array\n";
    printArray (arr, n);

    return 0;
}

```

Output:

```

Following is sorted array
2 3 6 8 12 56

```

The Min Heap based method takes $O(n \log k)$ time and uses $O(k)$ auxiliary space.

We can also **use a Balanced Binary Search Tree** instead of Heap to store $K+1$ elements. The [insert](#) and [delete](#) operations on Balanced BST also take $O(\log k)$ time. So Balanced BST based method will also take $O(n \log k)$ time, but the Heap based method seems to be more efficient as the **minimum element** will always be at root. Also, Heap doesn't need extra space for left and right pointers.

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

[Tournament tree](#) is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

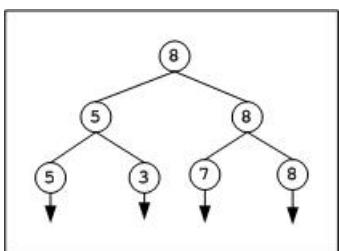
There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](#) (put $n = 2$ in equation given in the post).

It is obvious that to select the best player among N players, $(N - 1)$ players to be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons. For details read [this comment](#).

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

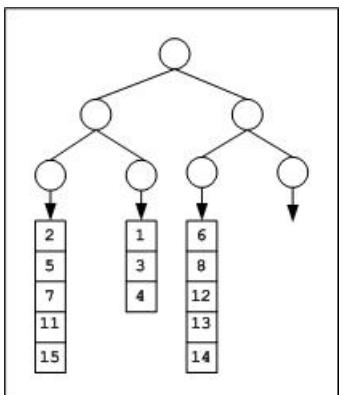
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2 M$)** to have atleast M external nodes.

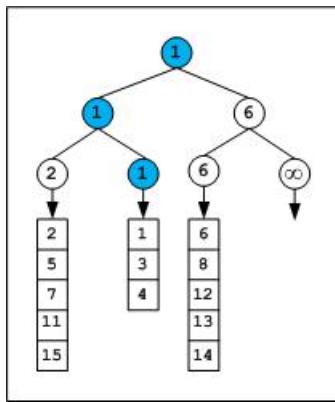
Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1  
{ 1, 3, 4 } ---- Array2  
{ 6, 8, 12, 13, 14 } ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 = 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



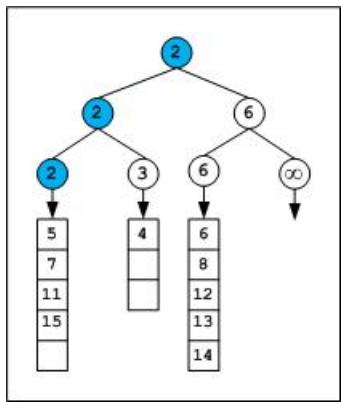
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being held in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2, \dots, L_m requires time complexity of $O(L_1 + L_2 + \dots + L_m) * \log M$ to merge all the arrays, and $O(m * \log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

Related Posts

[Link 1](#), [Link 2](#), [Link 3](#), [Link 4](#), [Link 5](#), [Link 6](#), [Link 7](#).

Hashing | Set 1 (Introduction)

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in $O(\log n)$ time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in $O(\log n)$ time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in $O(1)$ time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get $O(1)$ search time on average (under reasonable assumptions) and $O(n)$ in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties

- 1) Efficiently computable.
- 2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Next Posts:

[Separate Chaining for Collision Handling](#)
[Open Addressing for Collision Handling](#)

References:

[MIT Video Lecture](#)

[IITD Video Lecture](#)

[Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.](#)

<http://www.cs.princeton.edu/~rs/AlgsDS07/10Hashing.pdf>

<http://www.martinbroadhurst.com/articles/hash-table.html>

[GATE CornerQuiz Corner](#)

Hashing | Set 2 (Separate Chaining)

We strongly recommend to refer below post as a prerequisite of this.

<http://geeksquiz.com/hashing-set-1-introduction/>

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is [Birthday Paradox](#). With only 23 persons, the probability that two people have same birthday is 50%.

How to handle Collisions?

There are mainly two methods to handle collision:

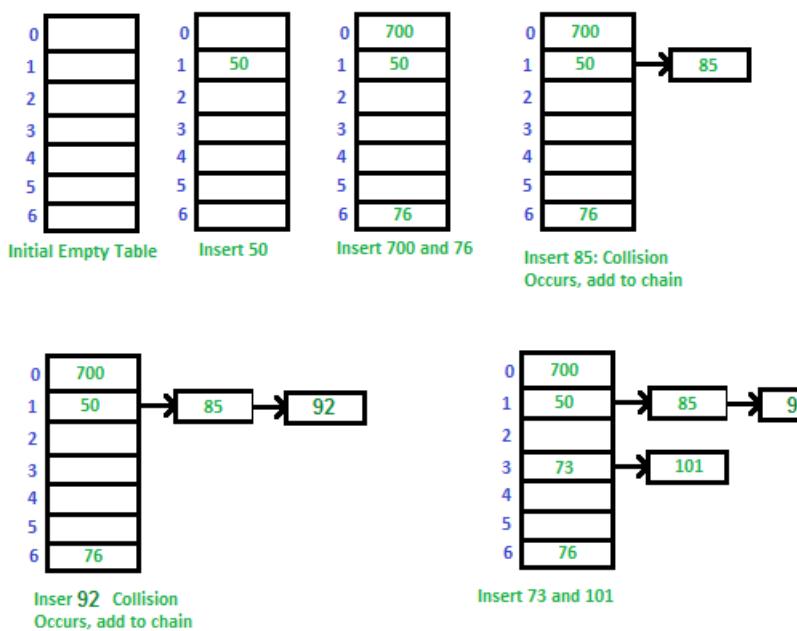
- 1) Separate Chaining
- 2) Open Addressing

In this article, only separate chaining is discussed. We will be discussing Open addressing in next post.

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as **key mod 7** and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table are never used)
- 3) If the chain becomes long, then search time can become O(n) in worst case.

4) Uses extra space for links.

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = Number of slots in hash table

n = Number of keys to be inserted in has table

Load factor $\alpha = n/m$

Expected time to search = $O(1 + \alpha)$

Expected time to insert/delete = $O(1 + \alpha)$

Time complexity of search insert and delete is
 $O(1)$ if α is $O(1)$

Next Post:

[Open Addressing for Collision Handling](#)

References:

http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture05.pdf

[GATE Corner](#)[Quiz Corner](#)

Hashing | Set 3 (Open Addressing)

We strongly recommend to refer below post as a prerequisite of this.

[Hashing | Set 1 \(Introduction\)](#)

[Hashing | Set 2 \(Separate Chaining\)](#)

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k .

Search(k): Keep probing until slots key doesn't become equal to k or an empty slot is reached.

Delete(k): **Delete operation is interesting.** If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as deleted.

Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

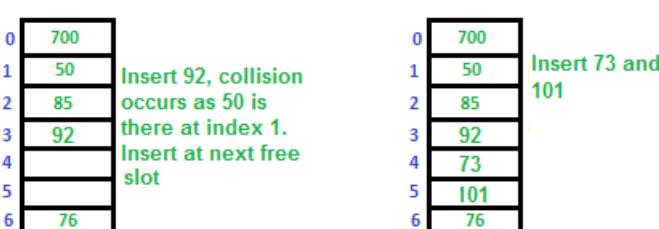
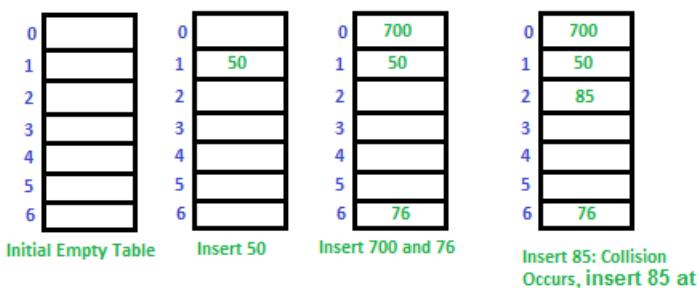
Open Addressing is done following ways:

a) **Linear Probing:** In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let $\text{hash}(x)$ be the slot index computed using hash function and S be the table size

```
If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1) \% S$ 
If  $(\text{hash}(x) + 1) \% S$  is also full, then we try  $(\text{hash}(x) + 2) \% S$ 
If  $(\text{hash}(x) + 2) \% S$  is also full, then we try  $(\text{hash}(x) + 3) \% S$ 
.....
....
```

Let us consider a simple hash function as key mod 7 and sequence of keys as 50, 700, 76, 85, 93, 73, 101.



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

b) **Quadratic Probing** We look for i^2 th slot in i th iteration.

```
let  $\text{hash}(x)$  be the slot index computed using hash function.
If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1*1) \% S$ 
If  $(\text{hash}(x) + 1*1) \% S$  is also full, then we try  $(\text{hash}(x) + 2*2) \% S$ 
If  $(\text{hash}(x) + 2*2) \% S$  is also full, then we try  $(\text{hash}(x) + 3*3) \% S$ 
.....
....
```

c) **Double Hashing** We use another hash function $\text{hash2}(x)$ and look for $i * \text{hash2}(x)$ slot in i th rotation.

```
let  $\text{hash}(x)$  be the slot index computed using hash function.
If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1*\text{hash2}(x)) \% S$ 
If  $(\text{hash}(x) + 1*\text{hash2}(x)) \% S$  is also full, then we try  $(\text{hash}(x) + 2*\text{hash2}(x)) \% S$ 
```

```
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S  
.....  
.....
```

See [this](#) for step by step diagrams.

Comparison of above three:

Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Open Addressing vs. Separate Chaining

Advantages of Chaining:

- 1) Chaining is Simpler to implement.
- 2) In chaining, Hash table never fills up, we can always add more elements to chain. In open addressing, table may become full.
- 3) Chaining is Less sensitive to the hash function or load factors.
- 4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- 5) Open addressing requires extra care for to avoid clustering and load factor.

Advantages of Open Addressing

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn't map to it.
- 3) Chaining uses extra space for links.

Performance of Open Addressing:

Like Chaining, performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing)

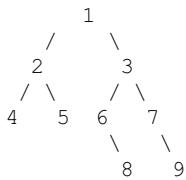
```
m = Number of slots in hash table  
n = Number of keys to be inserted in has table  
  
Load factor ? = n/m ( < 1 )  
  
Expected time to search/insert/delete < 1/(1 - ?)  
  
So Search, Insert and Delete take O(1/(1 + ?)) time
```

References:

<http://courses.csail.mit.edu/6.006/fall11/lectures/lecture10.pdf>
https://www.cse.cuhk.edu.hk/irwin.king/_media/teaching/csc2100b/tu6.pdf

Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method)

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.



The output of print this tree vertically will be:

```
4  
2  
1 5 6  
3 8  
7  
9
```

We have discussed a $O(n^2)$ solution in the [previous post](#). In this post, an efficient solution based on hash map is discussed. We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do inorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1. For every HD value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in map.

Following is C++ implementation of the above method. Thanks to Chirag for providing the below C++ implementation.

```
// C++ program for printing vertical order of a given binary tree
#include <iostream>
#include <vector>
#include <map>
using namespace std;

// Structure for a binary tree node
struct Node
{
    int key;
    Node *left, *right;
};

// A utility function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

// Utility function to store vertical order in map 'm'
// 'hd' is horizontal distance of current node from root.
// 'hd' is initially passed as 0
void getVerticalOrder(Node* root, int hd, map<int, vector<int>> &m)
{
    // Base case
    if (root == NULL)
        return;

    // Store current node in map 'm'
    m[hd].push_back(root->key);

    // Store nodes in left subtree
    getVerticalOrder(root->left, hd-1, m);

    // Store nodes in right subtree
    getVerticalOrder(root->right, hd+1, m);
}

// The main function to print vertical order of a binary tree
// with given root
void printVerticalOrder(Node* root)
```

```

{
    // Create a map and store vertical order in map using
    // function getVerticalOrder()
    map < int, vector<int> > m;
    int hd = 0;
    getVerticalOrder(root, hd, m);

    // Traverse the map and print nodes at every horizontal
    // distance (hd)
    map< int, vector<int> > :: iterator it;
    for (it=m.begin(); it!=m.end(); it++)
    {
        for (int i=0; i<it->second.size(); ++i)
            cout << it->second[i] << " ";
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);
    cout << "Vertical order traversal is \n";
    printVerticalOrder(root);
    return 0;
}

```

Output:

```

Vertical order traversal is
4
2
1 5 6
3 8
7
9

```

Time Complexity of hashing based solution can be considered as $O(n)$ under the assumption that we have good hashing function that allows insertion and retrieval operations in $O(1)$ time. In the above C++ implementation, [map of STL](#) is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take $O(\log n)$ time. Therefore time complexity of above implementation is $O(n \log n)$.

Find whether an array is subset of another array | Added Method 3

Given two arrays: arr1[0..m-1] and arr2[0..n-1]. Find whether arr2[] is a subset of arr1[] or not. Both the arrays are not in sorted order. It may be assumed that elements in both array are distinct.

Examples:

Input: arr1[] = {11, 1, 13, 21, 3, 7}, arr2[] = {11, 3, 7, 1}

Output: arr2[] is a subset of arr1[]

Input: arr1[] = {1, 2, 3, 4, 5, 6}, arr2[] = {1, 2, 4}

Output: arr2[] is a subset of arr1[]

Input: arr1[] = {10, 5, 2, 23, 19}, arr2[] = {19, 5, 3}

Output: arr2[] is not a subset of arr1[]

Method 1 (Simple)

Use two loops: The outer loop picks all the elements of arr2[] one by one. The inner loop linearly searches for the element picked by outer loop. If all elements are found then return 1, else return 0.

```
#include<stdio.h>

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;
    int j = 0;
    for (i=0; i<n; i++)
    {
        for (j = 0; j<m; j++)
        {
            if(arr2[i] == arr1[j])
                break;
        }

        /* If the above inner loop was not broken at all then
           arr2[i] is not present in arr1[] */
        if (j == m)
            return 0;
    }

    /* If we reach here then all elements of arr2[]
       are present in arr1[] */
    return 1;
}

int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[] ");

    getchar();
    return 0;
}
```

Time Complexity: O(m*n)

Method 2 (Use Sorting and Binary Search)

- 1) Sort arr1[] O(mLogm)
- 2) For each element of arr2[], do binary search for it in sorted arr1[].
 - a) If the element is not found then return 0.
- 3) If all elements are present then return 1.

```
#include<stdio.h>

/* Function prototypes */
void quickSort(int *arr, int si, int ei);
int binarySearch(int arr[], int low, int high, int x);
```

```

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;

    quickSort(arr1, 0, m-1);
    for (i=0; i<n; i++)
    {
        if (binarySearch(arr1, 0, m-1, arr2[i]) == -1)
            return 0;
    }

    /* If we reach here then all elements of arr2[]
       are present in arr1[] */
    return 1;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SEARCHING AND SORTING PURPOSE */
/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int x)
{
    if(high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/

        /* Check if arr[mid] is the first occurrence of x.
           arr[mid] is first occurrence if x is one of the following
           is true:
           (i)  mid == 0 and arr[mid] == x
           (ii) arr[mid-1] < x and arr[mid] == x
        */
        if(( mid == 0 || x > arr[mid-1]) && (arr[mid] == x))
            return mid;
        else if(x > arr[mid])
            return binarySearch(arr, (mid + 1), high, x);
        else
            return binarySearch(arr, low, (mid -1), x);
    }
    return -1;
}

void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a    = *b;
    *b    = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

```

```

}

/*Driver program to test above functions */
int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[] ");

    getchar();
    return 0;
}

```

Time Complexity: $O(m\log m + n\log n)$. Please note that this will be the complexity if an $m\log m$ algorithm is used for sorting which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is $O(n^2)$

Method 3 (Use Sorting and Merging)

- 1) Sort both arrays: $arr1[]$ and $arr2[]$ $O(m\log m + n\log n)$
- 2) Use Merge type of process to see if all elements of sorted $arr2[]$ are present in sorted $arr1[]$.

Thanks to [Parthsarthi](#) for suggesting this method.

```

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0, j = 0;

    if(m < n)
        return 0;

    quickSort(arr1, 0, m-1);
    quickSort(arr2, 0, n-1);
    while( i < n && j < m )
    {
        if( arr1[j] < arr2[i] )
            j++;
        else if( arr1[j] == arr2[i] )
        {
            j++;
            i++;
        }
        else if( arr1[j] > arr2[i] )
            return 0;
    }

    if( i < n )
        return 0;
    else
        return 1;
}

```

Time Complexity: $O(m\log m + n\log n)$ which is better than method 2. Please note that this will be the complexity if an $n\log n$ algorithm is used for sorting both arrays which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is $O(n^2)$

Method 4 (Use Hashing)

- 1) Create a Hash Table for all the elements of $arr1[]$.
- 2) Traverse $arr2[]$ and search for each element of $arr2[]$ in the Hash Table. If element is not found then return 0.
- 3) If all elements are found then return 1.

Note that method 1, method 2 and method 4 don't handle the cases when we have duplicates in $arr2[]$. For example, $\{1, 4, 4, 2\}$ is not a subset of $\{1, 4, 2\}$, but these methods will print it as a subset.

Source: <http://geeksforgeeks.org/forum/topic/if-an-array-is-subset-of-another>

Union and Intersection of two Linked Lists

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn't matter.

Example:

Input:
List1: 10->15->4->20
List2: 8->4->2->10

Output:
Intersection List: 4->10
Union List: 2->8->20->4->15->10

Method 1 (Simple)

Following are simple algorithms to get union and intersection lists respectively.

Intersection (list1, list2)

Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

Union (list1, list2):

Initialize result list as NULL. Traverse list1 and add all of its elements to the result.

Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to [Shekhu](#) for suggesting this method. Following is C implementation of this method.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the beginning of a linked list*/
void push (struct node** head_ref, int new_data);

/* A utility function to check if given data is present in a list */
bool isPresent (struct node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}

/* Function to get intersection of two linked lists head1 and head2 */
struct node *getIntersection (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in list2. If the element
    // is present in list 2, then insert the element to result
    while (t1 != NULL)
```

```

    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utility function that returns true if data is present in linked list
else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("\n First list is \n");
    printList (head1);

    printf ("\n Second list is \n");
    printList (head2);

    printf ("\n Intersection list is \n");
}

```

```

printList (intersecn);

printf ("\n Union list is \n");
printList (unin);

return 0;
}

```

Output:

```

First list is
10 15 4 20
Second list is
8 4 2 10
Intersection list is
4 10
Union list is
2 8 20 4 15 10

```

Time Complexity: $O(mn)$ for both union and intersection operations. Here m is the number of elements in first list and n is the number of elements in second list.

Method 2 (Use Merge Sort)

In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection.

Following are the steps to be followed to get union and intersection lists.

- 1) Sort the first Linked List using merge sort. This step takes $O(m\log m)$ time. Refer [this post](#) for details of this step.
- 2) Sort the second Linked List using merge sort. This step takes $O(n\log n)$ time. Refer [this post](#) for details of this step.
- 3) Linearly scan both sorted lists to get the union and intersection. This step takes $O(m + n)$ time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed [here](#).

Time complexity of this method is $O(m\log m + n\log n)$ which is better than method 1's time complexity.

Method 3 (Use Hashing)

Union (list1, list2)

Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

Intersection (list1, list2)

Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

Source: <http://geeksforgeeks.org/forum/topic/union-intersection-of-unsorted-lists>

Given an array A[] and a number x, check for pair in A[] with sum as x

Write a C program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

METHOD 1 (Use Sorting)

Algorithm:

```
hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
   (a) Initialize first to the leftmost index: l = 0
   (b) Initialize second the rightmost index: r = ar_size-1
3) Loop while l < r.
   (a) If (A[l] + A[r] == sum) then return 1
   (b) Else if( A[l] + A[r] < sum ) then l++
   (c) Else r--
4) No candidates in whole array - return 0
```

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then $O(n \log n)$ in worst case. If we use Quick Sort then $O(n^2)$ in worst case.

Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is $O(n)$ for merge sort and $O(1)$ for Heap Sort.

Example:

Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16

Sort the array

A = {-8, 1, 4, 6, 10, 45}

Initialize l=0, r=5

A[l] + A[r] (-8 + 45) > 16 => decrement r. Now r = 10

A[l] + A[r] (-8 + 10) < 2 => increment l. Now l = 1

A[l] + A[r] (1 + 10) < 16 => increment l. Now l = 2

A[l] + A[r] (4 + 10) < 14 => increment l. Now l = 3

A[l] + A[r] (6 + 10) == 16 => Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

Implementation:

C

```
# include <stdio.h>
# define bool int

void quickSort(int *, int, int);

bool hasArrayTwoCandidates(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

    /* Now look for the two candidates in the sorted
       array*/
    l = 0;
    r = arr_size-1;
    while (l < r)
    {
        if(A[l] + A[r] == sum)
            return 1;
        else if(A[l] + A[r] < sum)
            l++;
        else // A[i] + A[j] > sum
            r--;
    }
    return 0;
}

/* Driver program to test above function */
int main()
{
```

```

int A[] = {1, 4, 45, 6, 10, -8};
int n = 16;
int arr_size = 6;

if( hasArrayTwoCandidates(A, arr_size, n))
    printf("Array has two elements with sum 16");
else
    printf("Array doesn't have two elements with sum 16 ");

getchar();
return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
PURPOSE */

void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a    = *b;
    *b    = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

```

Python

```

# Python program to check for the sum condition to be satisfied
def hasArrayTwoCandidates(A,arr_size,sum):

    # sort the array
    quickSort(A,0,arr_size-1)
    l = 0
    r = arr_size-1

    # traverse the array for the two elements
    while l < r:
        if (A[l] + A[r] == sum):
            return 1
        elif (A[l] + A[r] < sum):
            l += 1
        else:
            r -= 1
    return 0

# Implementation of Quick Sort
# A[] --> Array to be sorted
# si --> Starting index

```

```

# ei --> Ending index
def quickSort(A, si, ei):
    if si < ei:
        pi=partition(A,si,ei)
        quickSort(A,si,pi-1)
        quickSort(A,pi+1,ei)

# Utility function for partitioning the array(used in quick sort)
def partition(A, si, ei):
    x = A[ei]
    i = (si-1)
    for j in range(si,ei):
        if A[j] <= x:
            i += 1

        # This operation is used to swap two variables in python
        A[i], A[j] = A[j], A[i]

    A[i+1], A[ei] = A[ei], A[i+1]

    return i+1

# Driver program to test the functions
A = [1,4,45,6,10,-8]
n = 16
if (hasArrayTwoCandidates(A, len(A), n)):
    print("Array has two elements with the given sum")
else:
    print("Array doesn't have two elements with the given sum")

## This code is contributed by __Devesh Agrawal__

```

Array has two elements with the given sum

METHOD 2 (Use Hash Map)

Thanks to Bindu for suggesting this method and thanks to [Shekhu](#) for providing code.

This method works in O(n) time if range of numbers is known.

Let sum be the given sum and A[] be the array in which we need to find pair.

- 1) Initialize Binary Hash Map M[] = {0, 0, }
- 2) Do following for each element A[i] in A[]
 - (a) If M[x - A[i]] is set then print the pair (A[i], x - A[i])
 - (b) Set M[A[i]]

Implementation:

C/C++

```

#include <stdio.h>
#define MAX 100000

void printPairs(int arr[], int arr_size, int sum)
{
    int i, temp;
    bool binMap[MAX] = {0}; /*initialize hash map as 0*/

    for (i = 0; i < arr_size; i++)
    {
        temp = sum - arr[i];
        if (temp >= 0 && binMap[temp] == 1)
            printf("Pair with given sum %d is (%d, %d) \n",
                   sum, arr[i], temp);
        binMap[arr[i]] = 1;
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int n = 16;
    int arr_size = sizeof(A)/sizeof(A[0]);
}

```

```

printPairs(A, arr_size, n);

getchar();
return 0;
}

```

Java

```

// Java implementation using Hashing
import java.io.*;

class PairSum
{
    private static final int MAX = 100000; // Max size of Hashmap

    static void printpairs(int arr[],int sum)
    {
        //Declares and initializes the whole array as false
        boolean[] binmap = new boolean[MAX];

        for (int i=0; i<arr.length; ++i)
        {
            int temp = sum-arr[i];

            //checking for condition
            if (temp>=0 && binmap[temp])
            {
                System.out.println("Pair with given sum " +
                    sum + " is (" + arr[i] +
                    ", " +temp+ ")");
            }
            binmap[arr[i]] = true;
        }
    }

    // Main to test the above function
    public static void main (String[] args)
    {
        int A[] = {1, 4, 45, 6, 10, 8};
        int n = 16;
        printpairs(A, n);
    }
}

// This article is contributed by Aakash Hasija

```

Python

```

# Python program to find if there are two elements wtih given sum
CONST_MAX = 100000

# function to check for the given sum in the array
def printPairs(arr, arr_size, sum):

    # initialize hash map as 0
    binmap = [0]*CONST_MAX

    for i in range(0,arr_size):
        temp = sum-arr[i]
        if (temp>=0 and binmap[temp]==1):
            print "Pair with the given sum is", arr[i], "and", temp
            binmap[arr[i]]=1

    # driver program to check the above function
A = [1,4,45,6,10,-8]
n = 16
printPairs(A, len(A), n)

# This code is contributed by __Devesh Agrawal__

```

Pair with given sum 16 is (10, 6)

Auxiliary Space: O(R) where R is range of integers.

If range of numbers include negative numbers then also it works. All we have to do for negative numbers is to make everything positive by adding

the absolute value of smallest negative integer to all numbers.

Check if a given array contains duplicate elements within k distance from each other

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than size of array. Write a function that returns true if array contains duplicates within k distance.

Examples:

Input: $k = 3$, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}
Output: false
All duplicates are more than k distance away.

Input: $k = 3$, arr[] = {1, 2, 3, 1, 4, 5}
Output: true
1 is repeated at distance 3.

Input: $k = 3$, arr[] = {1, 2, 3, 4, 5}
Output: false

Input: $k = 3$, arr[] = {1, 2, 3, 4, 4}
Output: true

A **Simple Solution** is to run two loops. The outer loop picks every element arr[i] as a starting element, the inner loop compares all elements which are within k distance of arr[i]. The time complexity of this solution is $O(kn)$.

We can solve this problem in $\Theta(n)$ time using Hashing. The idea is to one by add elements to hash. We also remove elements which are at more than k distance from current element. Following is detailed algorithm.

- 1) Create an empty hashtable.
- 2) Traverse all elements from left from right. Let the current element be arr[i]
 - a) If current element arr[i] is present in hashtable, then return true.
 - b) Else add arr[i] to hash and remove arr[i-k] from hash if i is greater than or equal to k

```
/* Java program to Check if a given array contains duplicate
elements within k distance from each other */
import java.util.*;

class Main
{
    static boolean checkDuplicatesWithinK(int arr[], int k)
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the input array
        for (int i=0; i<arr.length; i++)
        {
            // If already present n hash, then we found
            // a duplicate within k distance
            if (set.contains(arr[i]))
                return true;

            // Add this item to hashset
            set.add(arr[i]);

            // Remove the k+1 distant item
            if (i >= k)
                set.remove(arr[i-k]);
        }
        return false;
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 5, 3, 4, 3, 5, 6};
        if (checkDuplicatesWithinK(arr, 3))
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

Output:

Yes

Find Itinerary from a given list of tickets

Given a list of tickets, find itinerary in order using the given list.

Example:

Input:

```
"Chennai" -> "Banglore"
"Bombay" -> "Delhi"
"Goa"      -> "Chennai"
"Delhi"    -> "Goa"
```

Output:

```
Bombay->Delhi, Delhi->Goa, Goa->Chennai, Chennai->Banglore,
```

It may be assumed that the input list of tickets is not cyclic and there is one ticket from every city except final destination.

One Solution is to build a graph and do [Topological Sorting](#) of the graph. Time complexity of this solution is O(n).

We can also use [hashing](#) to avoid building a graph. The idea is to first find the starting point. A starting point would never be on to side of a ticket. Once we find the starting point, we can simply traverse the given map to print itinerary in order. Following are steps.

- 1) Create a HashMap of given pair of tickets. Let the created HashMap be 'dataset'. Every entry of 'dataset' is of the form "from->to" like "Chennai" -> "Banglore"
- 2) Find the starting point of itinerary.
 - a) Create a reverse HashMap. Let the reverse be 'reverseMap'. Entries of 'reverseMap' are of the form "to->form". Following is 'reverseMap' for above example.

```
"Banglore"-> "Chennai"
"Delhi"      -> "Bombay"
"Chennai"    -> "Goa"
"Goa"        -> "Delhi"
```
 - b) Traverse 'dataset'. For every key of dataset, check if it is there in 'reverseMap'. If a key is not present, then we found the starting point. In the above example, "Bombay" is starting point.
- 3) Start from above found starting point and traverse the 'dataset' to print itinerary.

All of the above steps require O(n) time so overall time complexity is O(n).

Below is Java implementation of above idea.

```
// Java program to print itinerary in order
import java.util.HashMap;
import java.util.Map;

public class printItinerary
{
    // Driver function
    public static void main(String[] args)
    {
        Map<String, String> dataSet = new HashMap<String, String>();
        dataSet.put("Chennai", "Banglore");
        dataSet.put("Bombay", "Delhi");
        dataSet.put("Goa", "Chennai");
        dataSet.put("Delhi", "Goa");

        printResult(dataSet);
    }

    // This function populates 'result' for given input 'dataset'
    private static void printResult(Map<String, String> dataSet)
    {
        // To store reverse of given map
        Map<String, String> reverseMap = new HashMap<String, String>();

        // To fill reverse map, iterate through the given map
        for (Map.Entry<String, String> entry: dataSet.entrySet())
            reverseMap.put(entry.getValue(), entry.getKey());

        // Find the starting point of itinerary
        String start = null;
        for (Map.Entry<String, String> entry: dataSet.entrySet())
            if (!reverseMap.containsKey(entry.getKey()))
```

```

{
    if (!reverseMap.containsKey(entry.getKey()))
    {
        start = entry.getKey();
        break;
    }
}

// If we could not find a starting point, then something wrong
// with input
if (start == null)
{
    System.out.println("Invalid Input");
    return;
}

// Once we have starting point, we simple need to go next, next
// of next using given hash map
String to = dataSet.get(start);
while (to != null)
{
    System.out.print(start + " -> " + to + ", ");
    start = to;
    to = dataSet.get(to);
}
}
}

```

Output:

Bombay->Delhi, Delhi->Goa, Goa->Chennai, Chennai->Banglore,

Find number of Employees Under every Employee

Given a dictionary that contains mapping of employee and his manager as a number of (employee, manager) pairs like below.

```
{ "A", "C" },
{ "B", "C" },
{ "C", "F" },
{ "D", "E" },
{ "E", "F" },
{ "F", "F" }
```

In this example C is manager of A,
C is also manager of B, F is manager
of C and so on.

Write a function to get no of employees under each manager in the hierarchy not just their direct reports. It may be assumed that an employee directly reports to only one manager. In the above dictionary the root node/ceo is listed as reporting to himself.

Output should be a Dictionary that contains following.

```
A - 0
B - 0
C - 2
D - 0
E - 1
F - 5
```

Source: Microsoft Interview

This question might be solved differently but i followed this and found interesting, so sharing:

1. Create a reverse map with Manager->DirectReportingEmployee combination. Off-course employee will be multiple so Value in Map is List of Strings.
"C" --> "A", "B",
"E" --> "D"
"F" --> "C", "E", "F"
2. Now use the given employee-manager map to iterate and at the same time use newly reverse map to find the count of employees under manager.

Let the map created in step 2 be 'mngrEmpMap'
Do following for every employee 'emp'.
a) If 'emp' is not present in 'mngrEmpMap'
 Count under 'emp' is 0 [Nobody reports to 'emp']
b) If 'emp' is present in 'mngrEmpMap'
 Use the list of direct reports from map 'mngrEmpMap'
 and recursively calculate number of total employees under 'emp'.

A trick in step 2.b is to use memorization(Dynamic programming) while finding number of employees under a manager so that we dont need to find number of employees again for any of the employees. In the below code populateResultUtil() is the recursive function that uses memoization to avoid re-computation of same results.

Below is Java implementation of above ideas

```
// Java program to find number of persons under every employee
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class NumberEmployeeUnderManager
{
    // A hashmap to store result. It stores count of employees
    // under every employee, the count may be 0 also
    static Map<String, Integer> result =
        new HashMap<String, Integer>();

    // Driver function
    public static void main(String[] args)
    {
        Map<String, String> dataSet = new HashMap<String, String>();
        dataSet.put("A", "C");
        dataSet.put("B", "C");
        dataSet.put("C", "F");
        dataSet.put("D", "E");
        dataSet.put("E", "F");
        dataSet.put("F", "F");
        populateResultUtil(dataSet);
    }

    private static int populateResultUtil(Map<String, String> dataSet, String emp, int count)
    {
        if (result.containsKey(emp))
            return result.get(emp);

        List<String> directReports = new ArrayList<String>();
        for (String mngr : dataSet.keySet())
            if (dataSet.get(mngr).equals(emp))
                directReports.add(mngr);

        int totalEmployees = 1;
        for (String report : directReports)
            totalEmployees += populateResultUtil(dataSet, report, 0);

        result.put(emp, totalEmployees);
        return totalEmployees;
    }
}
```

```

        dataSet.put("D", "E");
        dataSet.put("E", "F");
        dataSet.put("F", "F");

        populateResult(dataSet);
        System.out.println("result = " + result);
    }

    // This function populates 'result' for given input 'dataset'
    private static void populateResult(Map<String, String> dataSet)
    {
        // To store reverse of original map, each key will have 0
        // to multiple values
        Map<String, List<String>> mngrEmpMap =
            new HashMap<String, List<String>>();

        // To fill mngrEmpMap, iterate through the given map
        for (Map.Entry<String, String> entry: dataSet.entrySet())
        {
            String emp = entry.getKey();
            String mngr = entry.getValue();
            if (!emp.equals(mngr)) // excluding emp-emp entry
            {
                // Get the previous list of direct reports under
                // current 'mgr' and add the current 'emp' to the list
                List<String> directReportList = mngrEmpMap.get(mngr);

                // If 'emp' is the first employee under 'mgr'
                if (directReportList == null)
                    directReportList = new ArrayList<String>();

                directReportList.add(emp);

                // Replace old value for 'mgr' with new
                // directReportList
                mngrEmpMap.put(mngr, directReportList);
            }
        }

        // Now use manager-Emp map built above to populate result
        // with use of populateResultUtil()

        // note- we are iterating over original emp-manager map and
        // will use mngr-emp map in helper to get the count
        for (String mngr: dataSet.keySet())
            populateResultUtil(mngr, mngrEmpMap);
    }

    // This is a recursive function to fill count for 'mgr' using
    // mngrEmpMap. This function uses memoization to avoid re-
    // computations of subproblems.
    private static int populateResultUtil(String mngr,
                                         Map<String, List<String>> mngrEmpMap)
    {
        int count = 0;

        // means employee is not a manager of any other employee
        if (!mngrEmpMap.containsKey(mngr))
        {
            result.put(mngr, 0);
            return 0;
        }

        // this employee count has already been done by this
        // method, so avoid re-computation
        else if (result.containsKey(mngr))
            count = result.get(mngr);

        else
        {
            List<String> directReportEmpList = mngrEmpMap.get(mngr);
            count = directReportEmpList.size();
            for (String directReportEmp: directReportEmpList)
                count += populateResultUtil(directReportEmp, mngrEmpMap);

            result.put(mngr, count);
        }
        return count;
    }
}

```

Output:

```
result = {D=0, E=1, F=5, A=0, B=0, C=2}
```

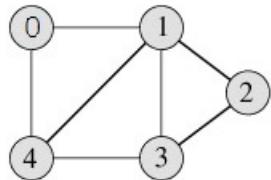
Graph and its representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. This can be easily viewed by <http://graph.facebook.com/barnwal.aashish> where barnwal.aashish is the profile name. See [this](#) for more applications of graph.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $\text{adj}[][],$ a slot $\text{adj}[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $\text{adj}[i][j] = w,$ then there is an edge from vertex i to vertex j with weight $w.$

The adjacency matrix for the above example graph is:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

Adjacency Matrix

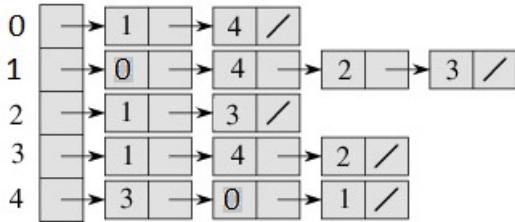
Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex u to vertex v are efficient and can be done $O(1).$

Cons: Consumes more space $O(V^2).$ Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be $\text{array}[].$ An entry $\text{array}[i]$ represents the linked list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

Below is C code for adjacency list representation of an undirected graph:

```

// A C Program to demonstrate adjacency list representation of graphs

#include <stdio.h>
#include <stdlib.h>

// A structure to represent an adjacency list node
struct AdjListNode
{
    int dest;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    int i;
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

```

```

}

// A utility function to print the adjacency list representation of graph
void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

// Driver program to test above functions
int main()
{
    // create the graph given in above figure
    int V = 5;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // print the adjacency list representation of the above graph
    printGraph(graph);

    return 0;
}

```

Output:

Adjacency list of vertex 0
head -> 4-> 1

Adjacency list of vertex 1
head -> 4-> 3-> 2-> 0

Adjacency list of vertex 2
head -> 3-> 1

Adjacency list of vertex 3
head -> 4-> 2-> 1

Adjacency list of vertex 4
head -> 3-> 1-> 0

Pros: Saves space $O(|V|+|E|)$. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

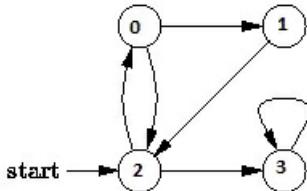
Reference:

http://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29

Breadth First Traversal for a Graph

[Breadth First Traversal \(or Search\)](#) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](#)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we dont mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are C++ and Java implementations of simple Breadth First Traversal from a given source.

The C++ implementation uses [adjacency list representation](#) of graphs. [STL's list container](#) is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given source vertex. BFS(int s)
// traverses vertices reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BFS(int s); // prints BFS traversal from a given source s
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        int u = queue.front();
        queue.pop_front();

        for(i = adj[u].begin(); i != adj[u].end(); ++i)
            if (!visited[*i])
                {
```

```

        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it visited
        // and enqueue it
        for(i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if(!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal (starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}

```

Java

```

// Java program to print BFS traversal from a given source vertex.
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)

```

```

{
    // Dequeue a vertex from queue and print it
    s = queue.poll();
    System.out.print(s+" ");

    // Get all adjacent vertices of the dequeued vertex s
    // If a adjacent has not been visited, then mark it
    // visited and enqueue it
    Iterator<Integer> i = adj[s].listIterator();
    while (i.hasNext())
    {
        int n = i.next();
        if (!visited[n])
        {
            visited[n] = true;
            queue.add(n);
        }
    }
}

// Driver method to
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Breadth First Traversal "+
                      "(starting from vertex 2)");

    g.BFS(2);
}
}

// This code is contributed by Aakash Hasija

```

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

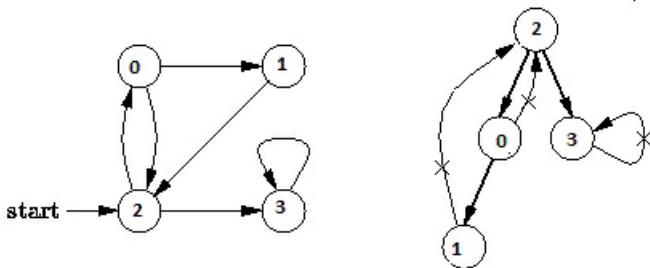
Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](#)).

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

Also see [Depth First Traversal](#)

Depth First Traversal for a Graph

[Depth First Traversal \(or Search\)](#) for a graph is similar to [Depth First Traversal of a tree](#). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See [this post](#) for all applications of Depth First Traversal.

Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](#) of graphs. [STL's list container](#) is used to store lists of adjacent nodes.

C++

```
// C++ program to print DFS traversal from a given vertex in a given graph
#include<iostream>
#include <list>

using namespace std;

// Graph class represents a directed graph using adjacency list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void DFS(int v);    // DFS traversal of the vertices reachable from v
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v. It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
}
```

```

    DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}

```

Java

```

// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;      // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);
    }

    public static void main(String args[])
    {

```

```

Graph g = new Graph(4);

g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

System.out.println("Following is Depth First Traversal "+
                    "(starting from vertex 2)");

g.DFS(2);
}

// This code is contributed by Aakash Hasija

```

Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call `DFSUtil()` for every vertex. Also, before calling `DFSUtil()`, we should check if it is already printed by some other call of `DFSUtil()`. Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

C++

```

// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include <list>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void DFS();    // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    // starting from all vertices one by one

```

```

for (int i = 0; i < V; i++)
    if (visited[i] == false)
        DFSUtil(i, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal\n";
    g.DFS();

    return 0;
}

```

Java

```

// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        // starting from all vertices one by one
        for (int i=0; i<V; ++i)
            if (visited[i] == false)

```

```

        DFSUtil(i, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal");
        g.DFS();
    }
}
// This code is contributed by Aakash Hasija

```

Following is Depth First Traversal
0 1 2 3

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

[Breadth First Traversal for a Graph](#)

Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.



1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](#) for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

- i) Call DFS(G, u) with u as the start vertex.
- ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
- iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

See [this](#) for details.

4) Topological Sorting

See [this](#) for details.

5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](#) for details.

6) Finding Strongly Connected Components of a graph

A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](#) for DFS based algo for finding Strongly Connected Components)

7) Solving puzzles with only one solution,

such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Sources:

<http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM>

http://en.wikipedia.org/wiki/Depth-first_search

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm>

<http://www.algorithmdesign.net/handouts/DFS.pdf>

Applications of Breadth First Traversal

We have earlier discussed [Breadth First Traversal Algorithm](#) for Graphs. We have also discussed [Applications of Depth First Traversal](#). In this article, applications of Breadth First Search are discussed.

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance k from a person using Breadth First Search till k levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using [Cheneys algorithm](#). Refer [this](#) and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) FordFulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Many algorithms like [Prims Minimum Spanning Tree](#) and [Dijkstras Single Source Shortest Path](#) use structure similar to Breadth First Search.

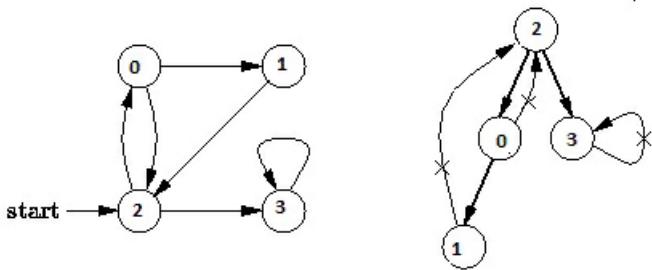
There can be many more applications as Breadth First Search is one of the core algorithm for Graphs.

Detect Cycle in a Directed Graph

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles $0 \rightarrow 2 \rightarrow 0$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ and $3 \rightarrow 3$, so your function must return true.

Solution

Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a [back edge](#) present in the graph. A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for cycle in individual trees by checking back edges.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is back edge. We have used `recStack[]` array to keep track of vertices in the recursion stack.

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], bool *rs); // used by isCyclic()
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

// This function is a variation of DFSUtil() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }
    }
}
```

```

    }
    recStack[v] = false; // remove the vertex from recursion stack
    return false;
}

// Returns true if the graph contains a cycle, else false.
// This function is a variation of DFS() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if(g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";
    return 0;
}

```

Output:

Graph contains cycle

Time Complexity of this method is same as time complexity of [DFS traversal](#) which is $O(V+E)$.

Union-Find Algorithm | Set 1 (Detect Cycle in a an Undirected Graph)

A [disjoint-set data structure](#) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [union-find algorithm](#) is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

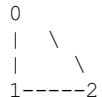
Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](#). This is another method based on *Union-Find*. This method assumes that graph doesn't contain any self-loops.

We can keep track of the subsets in a 1D array, let's call it `parent[]`.

Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

```
0  1  2
-1 -1 -1
```

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

```
0  1  2      <---- 1 is made parent of 0 (1 is now representative of subset {0, 1})
1  -1 -1
```

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.

```
0  1  2      <---- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})
1  2  -1
```

Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

0->1->2 // 1 is parent of 0 and 2 is parent of 1

Based on the above explanation, below are implementations:

C/C++

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
```

```

struct Graph* graph =
    (struct Graph*) malloc( sizeof(struct Graph) );
graph->V = V;
graph->E = E;

graph->edge =
    (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle( struct Graph* graph )
{
    // Allocate memory for creating V subsets
    int *parent = (int*) malloc( graph->V * sizeof(int) );

    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for(int i = 0; i < graph->E; ++i)
    {
        int x = find(parent, graph->edge[i].src);
        int y = find(parent, graph->edge[i].dest);

        if (x == y)
            return 1;

        Union(parent, x, y);
    }
    return 0;
}

// Driver program to test above functions
int main()
{
    /* Let us create following graph
       0
       |
       |
       1---2 */
    struct Graph* graph = createGraph(3, 3);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "Graph contains cycle" );
    else
        printf( "Graph doesn't contain cycle" );

    return 0;
}

```

}

Java

```

// Java Program for union-find algorithm to detect cycle in a graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    int V, E;      // V-> no. of vertices & E->no.of edges
    Edge edge[]; // /collection of all edges

    class Edge
    {
        int src, dest;
    };

    // Creates a graph with V vertices and E edges
    Graph(int v,int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // A utility function to find the subset of an element i
    int find(int parent[], int i)
    {
        if (parent[i] == -1)
            return i;
        return find(parent, parent[i]);
    }

    // A utility function to do union of two subsets
    void Union(int parent[], int x, int y)
    {
        int xset = find(parent, x);
        int yset = find(parent, y);
        parent[xset] = yset;
    }

    // The main function to check whether a given graph
    // contains cycle or not
    int isCycle( Graph graph)
    {
        // Allocate memory for creating V subsets
        int parent[] = new int[graph.V];

        // Initialize all subsets as single element sets
        for(int i=0; i<graph.V; ++i)
            parent[i]=-1;

        // Iterate through all edges of graph, find subset of both
        // vertices of every edge, if both subsets are same, then
        // there is cycle in graph.
        for (int i = 0; i < graph.E; ++i)
        {
            int x = graph.find(parent, graph.edge[i].src);
            int y = graph.find(parent, graph.edge[i].dest);

            if (x == y)
                return 1;

            graph.Union(parent, x, y);
        }
        return 0;
    }

    // Driver Method
    public static void main (String[] args)
    {
        /* Let us create following graph
         0
         | \
         |   \
         |     \
         */
    }
}

```

```

1-----2 */
Graph graph = new Graph(3,3);

// add edge 0-1
graph.edge[0].src = 0;
graph.edge[0].dest = 1;

// add edge 1-2
graph.edge[1].src = 1;
graph.edge[1].dest = 2;

// add edge 0-2
graph.edge[2].src = 0;
graph.edge[2].dest = 2;

if (graph.isCycle(graph)==1)
    System.out.println( "Graph contains cycle" );
else
    System.out.println( "Graph doesn't contain cycle" );
}
}

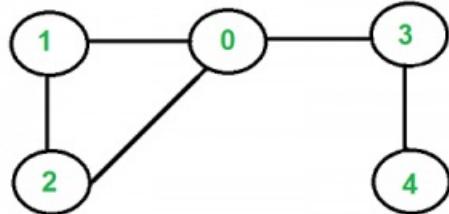
```

Graph contains cycle

Note that the implementation of *union()* and *find()* is naive and takes $O(n)$ time in worst case. These methods can be improved to $O(\log n)$ using *Union by Rank or Height*. We will soon be discussing *Union by Rank* in a separate post.

Detect cycle in an undirected graph

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.



We have discussed [cycle detection for directed graph](#). We have also discussed a [union-find algorithm for cycle detection in undirected graphs](#). The time complexity of the union-find algorithm is $O(E \log V)$. Like directed graphs, we can use [DFS](#) to detect cycle in an undirected graph in $O(V+E)$ time. We do a DFS traversal of the given graph. For every visited vertex v , if there is an adjacent u such that u is already visited and u is not parent of v , then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

C++

```
// A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
    adj[w].push_back(v); // Add v to ws list.
}

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])
        {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of current vertex,
        // then there is a cycle.
        else if (*i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
```

```

bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if it is already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isCyclic() ? cout << "Graph contains cycle\n":
                    cout << "Graph doesn't contain cycle\n";

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.isCyclic() ? cout << "Graph contains cycle\n":
                    cout << "Graph doesn't contain cycle\n";

    return 0;
}

```

Java

```

// A Java Program to detect cycle in an undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;      // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List Representation

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for(int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    // A recursive function that uses visited[] and parent to detect
    // cycle in subgraph reachable from vertex v.
    Boolean isCyclicUtil(int v, Boolean visited[], int parent)
    {
        // Mark the current node as visited
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();

```

```

        // If an adjacent is not visited, then recur for that
        // adjacent
        if (!visited[i])
        {
            if (isCyclicUtil(i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of current
        // vertex, then there is a cycle.
        else if (i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
Boolean isCyclic()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in
    // different DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver method to test above methods
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    if (g1.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't contain cycle");

    Graph g2 = new Graph(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    if (g2.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't contain cycle");
}
}

// This code is contributed by Aakash Hasija

```

Output:

```

Graph contains cycle
Graph doesn't contain cycle

```

Time Complexity: The program does a simple DFS Traversal of graph and graph is represented using adjacency list. So the time complexity is $O(V+E)$

Exercise: Can we use BFS to detect cycle in an undirected graph in $O(V+E)$ time? What about directed graphs?

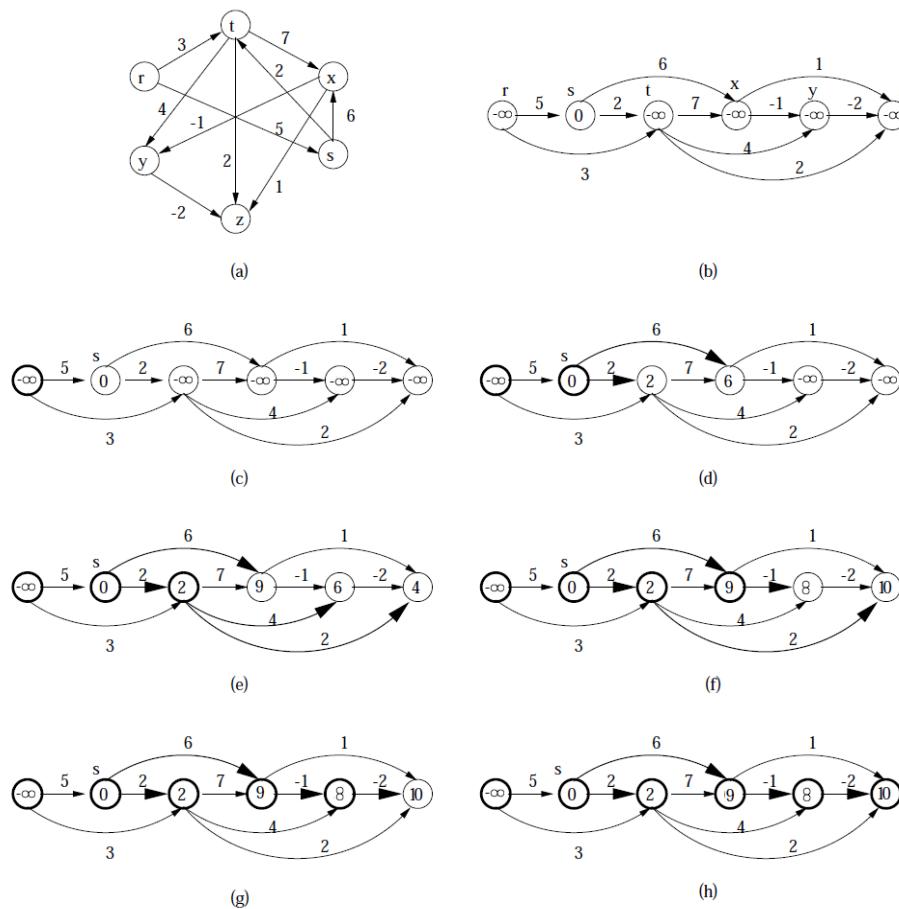
Longest Path in a Directed Acyclic Graph

Given a Weighted Directed Acyclic Graph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn't have [optimal substructure property](#). In fact, [the Longest Path problem is NP-Hard for a general graph](#). However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to [linear time solution for shortest path in a directed acyclic graph](#), we use [Topological Sorting](#).

We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a [topological sorting](#) of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure shows step by step process of finding longest paths.



Following is complete algorithm for finding longest distances.

1) Initialize $\text{dist}[] = \{\text{NINF}, \text{NINF}, \dots\}$ and $\text{dist}[s] = 0$ where s is the source vertex. Here NINF means negative infinite.

2) Create a topological order of all vertices.

3) Do following for every vertex u in topological order.

..Do following for every adjacent vertex v of u

if ($\text{dist}[v] < \text{dist}[u] + \text{weight}(u, v)$) $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$

Following is C++ implementation of the above algorithm.

```
// A C++ program to find single source longest distances in a DAG
#include <iostream>
#include <list>
#include <stack>
#include <limits.h>
#define NINF INT_MIN
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
}
```

```

public:
    AdjListNode(int _v, int _w) { v = _v; weight = _w; }
    int getV() { return v; }
    int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing adjacency lists
    list<AdjListNode> *adj;

    // A function used by longestPath
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds longest distances from given source vertex
    void longestPath(int s);
};

Graph::Graph(int V) // Constructor
{
    this->V = V;
    adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to us list
}

// A recursive function used by longestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological sort
    Stack.push(v);
}

// The function to find longest distances from a given vertex. It uses
// recursive topologicalSortUtil() to get topological sorting.
void Graph::longestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance
    // to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = NINF;
    dist[s] = 0;
}

```

```

// Process vertices in topological order
while (Stack.empty() == false)
{
    // Get the next vertex from topological order
    int u = Stack.top();
    Stack.pop();

    // Update distances of all adjacent vertices
    list<AdjListNode>::iterator i;
    if (dist[u] != NINF)
    {
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (dist[i->getV()] < dist[u] + i->getWeight())
                dist[i->getV()] = dist[u] + i->getWeight();
    }
}

// Print the calculated longest distances
for (int i = 0; i < V; i++)
    (dist[i] == NINF)? cout << "INF ": cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram. Here vertex numbers are
    // 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
    Graph g(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 5, 1);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    cout << "Following are longest distances from source vertex " << s << "\n";
    g.longestPath(s);

    return 0;
}

```

Output:

```

Following are longest distances from source vertex 1
INF 0 2 9 8 10

```

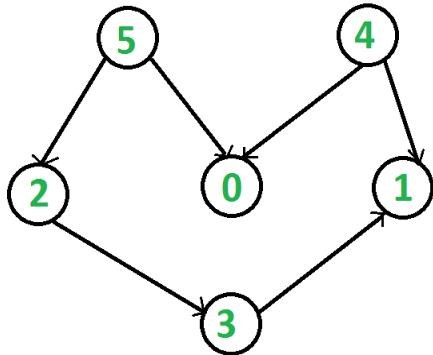
Time Complexity: Time complexity of topological sorting is $O(V+E)$. After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is $O(E)$. So the inner loop runs $O(V+E)$ times. Therefore, overall time complexity of this algorithm is $O(V+E)$.

Exercise: The above solution print longest distances, extend the code to print paths also.

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is $5\ 4\ 2\ 3\ 1\ 0?$. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is $4\ 5\ 2\ 3\ 1\ 0?$. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



Topological Sorting vs Depth First Traversal (DFS):

In [DFS](#), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex 5 should be printed before vertex 0, but unlike [DFS](#), the vertex 4 should also be printed before vertex 0. So Topological sorting is different from DFS. For example, a DFS of the above graph is $5\ 2\ 3\ 1\ 0\ 4?$, but it is not a topological sorting

Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS [here](#). We can modify [DFS](#) to find Topological Sorting of a graph. In [DFS](#), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We dont print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Following are C++ and Java implementations of topological sorting. Please see the code for Depth [First Traversal for a disconnected Graph](#) and note the differences between the second code given there and the below code.

C++

```
// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;      // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
```

```

        adj[v].push_back(w); // Add w to vs list.
    }

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given graph \n";
    g.topologicalSort();

    return 0;
}

```

Java

```

// A Java program to print topological sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
    }
}
```

```

adj = new LinkedList[v];
for (int i=0; i<v; ++i)
    adj[i] = new LinkedList();
}

// Function to add an edge into the graph
void addEdge(int v,int w) { adj[v].add(w); }

// A recursive function used by topologicalSort
void topologicalSortUtil(int v, Boolean visited[],Stack stack)
{
    // Mark the current node as visited.
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext())
    {
        i = it.next();
        if (!visited[i])
            topologicalSortUtil(i, visited, stack);
    }

    // Push current vertex to stack which stores result
    stack.push(new Integer(v));
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void topologicalSort()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

    // Print contents of stack
    while (stack.empty() == false)
        System.out.print(stack.pop() + " ");
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    System.out.println("Following is a Topological " +
                       "sort of the given graph");
    g.topologicalSort();
}
}

// This code is contributed by Aakash Hasija

```

Following is a Topological Sort of the given graph
5 4 2 3 1 0

Time Complexity: The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is $O(V+E)$.

Applications:

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

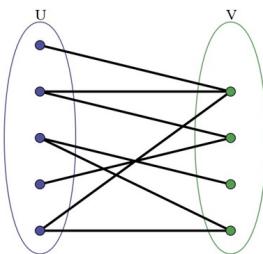
References:

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm>

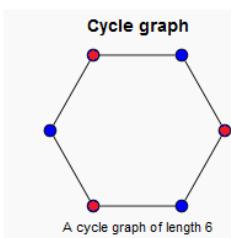
http://en.wikipedia.org/wiki/Topological_sorting

Check whether a given graph is Bipartite or not

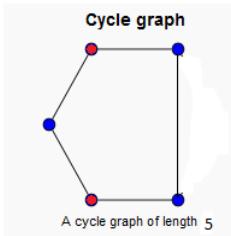
A [Bipartite Graph](#) is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v) , either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



It is not possible to color a cycle graph with odd cycle using two colors.



Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using [backtracking algorithm m coloring problem](#).

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbors neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

C++

```
// C++ program to find out whether a given graph is Bipartite or not
#include <iostream>
#include <queue>
#define V 4
using namespace std;

// This function returns true if graph G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors assigned to all vertices. Vertex
    // number is used as index in this array. The value '-1' of colorArr[i]
    // is used to indicate that no color is assigned to vertex 'i'. The value
    // 1 is used to indicate first color is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and enqueue source vertex
    queue<int> q;
    q.push(src);

    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        // Get all adjacent vertices of the dequeued vertex u
        // If an adjacent vertex is not colored, then assign
        // current color to this adjacent vertex, enqueue it
        // If an adjacent vertex is colored with same color as
        // current vertex, then return false
        for (int v = 0; v < V; v++)
        {
            if (G[u][v] == 1)
            {
                if (colorArr[v] == colorArr[u])
                    return false;
                if (colorArr[v] == -1)
                {
                    colorArr[v] = 1 - colorArr[u];
                    q.push(v);
                }
            }
        }
    }
    return true;
}
```

```

// for BFS traversal
queue <int> q;
q.push(src);

// Run while there are vertices in queue (Similar to BFS)
while (!q.empty())
{
    // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
    int u = q.front();
    q.pop();

    // Find all non-colored adjacent vertices
    for (int v = 0; v < V; ++v)
    {
        // An edge from u to v exists and destination v is not colored
        if (G[u][v] && colorArr[v] == -1)
        {
            // Assign alternate color to this adjacent v of u
            colorArr[v] = 1 - colorArr[u];
            q.push(v);
        }

        // An edge from u to v exists and destination v is colored with
        // same color as u
        else if (G[u][v] && colorArr[v] == colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent vertices can be colored with
// alternate color
return true;
}

// Driver program to test above function
int main()
{
    int G[][][V] = {{0, 1, 0, 1},
                    {1, 0, 1, 0},
                    {0, 1, 0, 1},
                    {1, 0, 1, 0}
    };

    isBipartite(G, 0) ? cout << "Yes" : cout << "No";
    return 0;
}

```

Java

```

// Java program to find out whether a given graph is Bipartite or not
import java.util.*;
import java.lang.*;
import java.io.*;

class Bipartite
{
    final static int V = 4; // No. of Vertices

    // This function returns true if graph G[V][V] is Bipartite, else false
    boolean isBipartite(int G[][],int src)
    {
        // Create a color array to store colors assigned to all vertices.
        // Vertex number is used as index in this array. The value '-1'
        // of colorArr[i] is used to indicate that no color is assigned
        // to vertex 'i'. The value 1 is used to indicate first color
        // is assigned and value 0 indicates second color is assigned.
        int colorArr[] = new int[V];
        for (int i=0; i<V; ++i)
            colorArr[i] = -1;

        // Assign first color to source
        colorArr[src] = 1;

        // Create a queue (FIFO) of vertex numbers and enqueue
        // source vertex for BFS traversal
        LinkedList<Integer>q = new LinkedList<Integer>();
        q.add(src);

        // Run while there are vertices in queue (Similar to BFS)
        while (q.size() != 0)

```

```

{
    // Dequeue a vertex from queue
    int u = q.poll();

    // Find all non-colored adjacent vertices
    for (int v=0; v<V; ++v)
    {
        // An edge from u to v exists and destination v is
        // not colored
        if (G[u][v]==1 && colorArr[v]==-1)
        {
            // Assign alternate color to this adjacent v of u
            colorArr[v] = 1-colorArr[u];
            q.add(v);
        }

        // An edge from u to v exists and destination v is
        // colored with same color as u
        else if (G[u][v]==1 && colorArr[v]==colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent vertices can
// be colored with alternate color
return true;
}

// Driver program to test above function
public static void main (String[] args)
{
    int G[][] = {{0, 1, 0, 1},
                 {1, 0, 1, 0},
                 {0, 1, 0, 1},
                 {1, 0, 1, 0}
    };
    Bipartite b = new Bipartite();
    if (b.isBipartite(G, 0))
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

// Contributed by Aakash Hasija

```

Yes

Refer [this](#) for C implementation of the same.

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is $O(V^2)$ where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes $O(V+E)$.

Exercise:

1. Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?
2. The above algorithm works if the graph is strongly connected. Extend above code to work for graph with more than one component.

References:

- http://en.wikipedia.org/wiki/Graph_coloring
- http://en.wikipedia.org/wiki/Bipartite_graph

Snake and Ladder Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.



For example consider the board shown on right side (taken from [here](#)), the minimum number of dice throws required to reach cell 30 from cell 1 is 3. Following are steps.

- First throw two on dice to reach cell number 3 and then ladder to reach 22.
- Then throw 6 to reach 28.
- Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

Following is C++ implementation of the above idea. The input is represented by two things, first is N which is number of cells in the given board, second is an array move[0:N-1] of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

```
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
    int v;      // Vertex number
    int dist;   // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s = {0, 0}; // distance of 0't vertex is also 0
    q.push(s); // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe; // A queue entry (qe)
    while (!q.empty())
    {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry
        int dist = qe.dist;
```

```

// If front vertex is the destination vertex,
// we are done
if (v == N-1)
    break;

// Otherwise dequeue the front vertex and enqueue
// its adjacent vertices (or cell numbers reachable
// through a dice throw)
q.pop();
for (int j=v+1; j<=(v+6) && j<N; ++j)
{
    // If this cell is already visited, then ignore
    if (!visited[j])
    {
        // Otherwise calculate its distance and mark it
        // as visited
        queueEntry a;
        a.dist = (qe.dist + 1);
        visited[j] = true;

        // Check if there a snake or ladder at 'j'
        // then tail of snake or top of ladder
        // become the adjacent of 'i'
        if (move[j] != -1)
            a.v = move[j];
        else
            a.v = j;
        q.push(a);
    }
}
}

// We reach here when 'qe' has last vertex
// return the distance of vertex in 'qe'
return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i<N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}

```

Output:

Min Dice throws required is 3

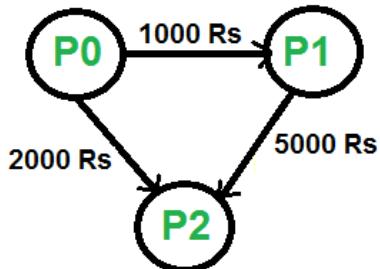
Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.

Minimize Cash Flow among a given set of friends who have borrowed money from each other

Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.

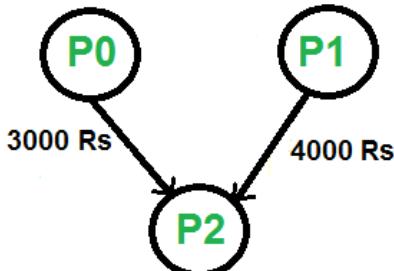
Example:

Following diagram shows input debts to be settled.



P0 has to pay 1000 Rs to P1
P0 also has to pay 2000 Rs to P2
P1 has to pay 5000 Rs to P2.

Above debts can be settled in following optimized way



P1 pays 4000 Rs to P2
P0 pays 3000 Rs to P2

The following is detailed algorithm

Do following for every person P_i where i is from 0 to $n-1$.

- 1) Compute the net amount for every person. The net amount for person i can be computed by subtracting sum of all debts from sum of all credits.
- 2) Find the two persons that are maximum creditor and maximum debtor. Let the maximum amount to be credited maximum creditor be maxCredit and maximum amount to be debited from maximum debtor be maxDebit . Let the maximum debtor be P_d and maximum creditor be P_c .
- 3) Find the minimum of maxDebit and maxCredit . Let minimum of two be x . Debit x from P_d and credit this amount to P_c
- 4) If x is equal to maxCredit , then remove P_c from set of persons and recur for remaining $(n-1)$ persons.
- 5) If x is equal to maxDebit , then remove P_d from set of persons and recur for remaining $(n-1)$ persons.

Thanks to Balaji S for suggesting this method in a comment [here](#).

The following is C++ implementation of above algorithm.

```
// C++ program to find maximum cash flow among a set of persons
#include<iostream>
using namespace std;

// Number of persons (or vertices in the graph)
#define N 3

// A utility function that returns index of minimum value in arr[]
int getMin(int arr[])
{
    int minInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] < arr[minInd])
            minInd = i;
    return minInd;
}

// A utility function that returns index of maximum value in arr[]
int getMax(int arr[])
{
    int maxInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}
```

```

{
    int maxInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}

// A utility function to return minimum of 2 values
int minOf2(int x, int y)
{
    return (x<y)? x: y;
}

// amount[p] indicates the net amount to be credited/debited
// to/from person 'p'
// If amount[p] is positive, then i'th person will amount[i]
// If amount[p] is negative, then i'th person will give -amount[i]
void minCashFlowRec(int amount[])
{
    // Find the indexes of minimum and maximum values in amount[]
    // amount[mxCredit] indicates the maximum amount to be given
    // (or credited) to any person .
    // And amount[mxDabit] indicates the maximum amount to be taken
    // (or debited) from any person.
    // So if there is a positive value in amount[], then there must
    // be a negative value
    int mxCredit = getMax(amount), mxDebit = getMin(amount);

    // If both amounts are 0, then all amounts are settled
    if (amount[mxCredit] == 0 && amount[mxDabit] == 0)
        return;

    // Find the minimum of two amounts
    int min = minOf2(-amount[mxDabit], amount[mxCredit]);
    amount[mxCredit] -= min;
    amount[mxDabit] += min;

    // If minimum is the maximum amount to be
    cout << "Person " << mxDebit << " pays " << min
        << " to " << "Person " << mxCredit << endl;

    // Recur for the amount array. Note that it is guaranteed that
    // the recursion would terminate as either amount[mxCredit]
    // or amount[mxDabit] becomes 0
    minCashFlowRec(amount);
}

// Given a set of persons as graph[] where graph[i][j] indicates
// the amount that person i needs to pay person j, this function
// finds and prints the minimum cash flow to settle all debts.
void minCashFlow(int graph[][N])
{
    // Create an array amount[], initialize all value in it as 0.
    int amount[N] = {0};

    // Calculate the net amount to be paid to person 'p', and
    // stores it in amount[p]. The value of amount[p] can be
    // calculated by subtracting debts of 'p' from credits of 'p'
    for (int p=0; p<N; p++)
        for (int i=0; i<N; i++)
            amount[p] += (graph[i][p] - graph[p][i]);

    minCashFlowRec(amount);
}

// Driver program to test above function
int main()
{
    // graph[i][j] indicates the amount that person i needs to
    // pay person j
    int graph[N][N] = { {0, 1000, 2000},
                        {0, 0, 5000},
                        {0, 0, 0}, };

    // Print the solution
    minCashFlow(graph);
    return 0;
}

```

Output:

Person 1 pays 4000 to Person 2
Person 0 pays 3000 to Person 2

Algorithmic Paradigm: Greedy

Time Complexity: $O(N^2)$ where N is the number of persons.

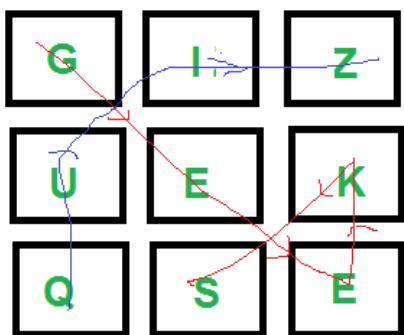
Boggle (Find all possible words in a board of characters)

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}};
       isWord(str): returns true if str is present in dictionary
                     else false.
```

```
Output: Following words of dictionary are present
        GEEKS
        QUIZ
```



The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using [Depth First Traversal](#). We do depth first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

```
// C++ program for Boggle game
#include<iostream>
#include<cstring>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in
// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is given to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
            return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
                   int j, string &str)
{
    // Mark current cell as visited and append current character
    // to str
    visited[i][j] = true;
    str = str + boggle[i][j];

    // If str is present in dictionary, then print it
    if (isWord(str))
        cout << str << endl;

    // Traverse 8 adjacent cells of boggle[i][j]
    for (int row=i-1; row<=i+1 && row<M; row++)
        for (int col=j-1; col<=j+1 && col<N; col++)
            if (row>=0 && col>=0 && !visited[row][col])
}
```

```

    findWordsUtil(boggle, visited, row, col, str);

    // Erase current character from string and mark visited
    // of current cell as false
    str.erase(str.length()-1);
    visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            findWordsUtil(boggle, visited, i, j, str);
}

// Driver program to test above function
int main()
{
    char boggle[M][N] = {{'G','I','Z'},
                         {'U','E','K'},
                         {'Q','S','E'}};

    cout << "Following words of dictionary are present\n";
    findWords(boggle);
    return 0;
}

```

Output:

```

Following words of dictionary are present
GEEKS
QUIZ

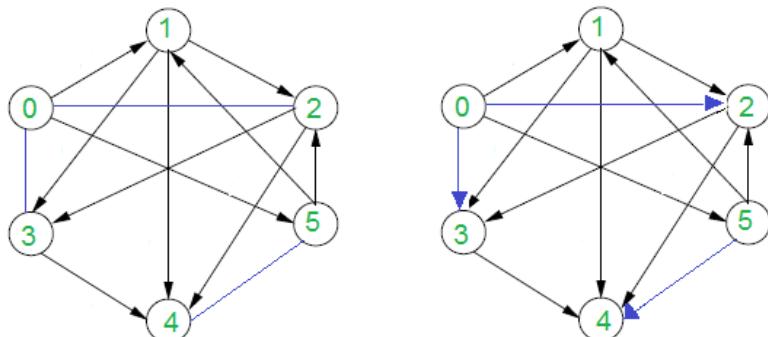
```

Note that the above solution may print same word multiple times. For example, if we add SEEK to dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

Assign directions to edges so that the directed graph remains acyclic

Given a graph with both directed and undirected edges. It is given that the directed edges don't form a cycle. How to assign directions to undirected edges so that the graph (with all directed edges) remains acyclic even after the assignment?

For example, in the below graph, blue edges don't have directions.

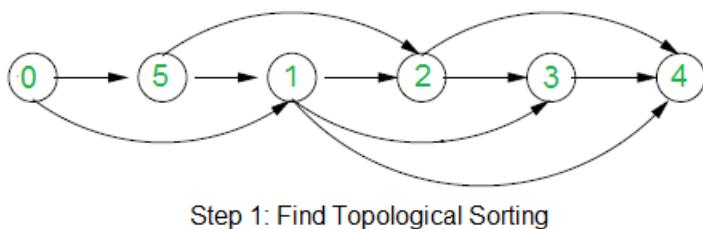


Given Graph

Graph after adding directions to undirected edges such that the graph remains acyclic.

The idea is to use [Topological Sorting](#). Following are two steps used in the algorithm.

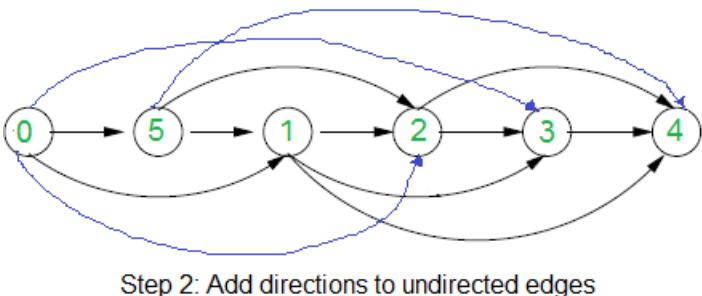
- 1) Consider the subgraph with directed edges only and find topological sorting of the subgraph. In the above example, topological sorting is {0, 5, 1, 2, 3, 4}. Below diagram shows topological sorting for the above example graph.



Step 1: Find Topological Sorting

- 2) Use above topological sorting to assign directions to undirected edges. For every undirected edge (u, v) , assign it direction from u to v if u comes before v in topological sorting, else assign it direction from v to u .

Below diagram shows assigned directions in the example graph.



Step 2: Add directions to undirected edges

Source: <http://courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf>

Greedy Algorithms | Set 5 (Prims Minimum Spanning Tree (MST))

We have discussed [Kruskals algorithm for Minimum Spanning Tree](#). Like Kruskals algorithm, Prims algorithm is also a [Greedy algorithm](#). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called [cut in graph theory](#). So, at every step of Prims algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

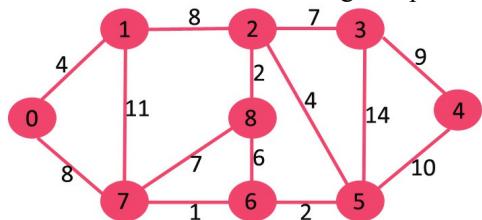
How does Prims Algorithm Work? The idea behind Prims algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *SpanningTree*. And they must be connected with the minimum weight edge to make it a *MinimumSpanning Tree*.

Algorithm

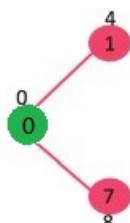
- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
 - a) Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
 - b) Include *u* to *mstSet*.
 - c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

The idea of using key values is to pick the minimum weight edge from [cut](#). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

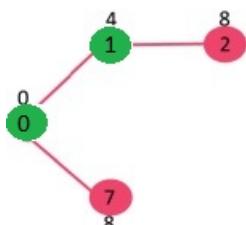
Let us understand with the following example:



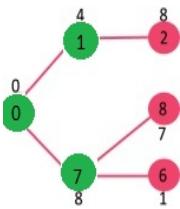
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



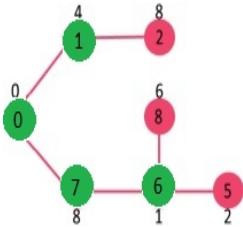
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



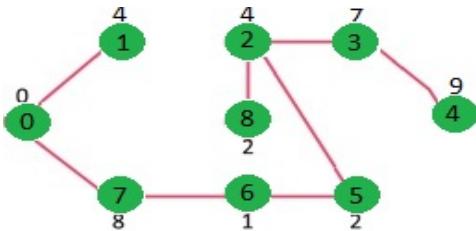
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in `mstSet`). Vertex 6 is picked. So `mstSet` now becomes $\{0, 1, 7, 6\}$. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until `mstSet` includes all vertices of given graph. Finally, we get the following graph.



How to implement the above algorithm?

We use a boolean array `mstSet[]` to represent the set of vertices included in MST. If a value `mstSet[v]` is true, then vertex `v` is included in MST, otherwise not. Array `key[]` is used to store key values of all vertices. Another array `parent[]` to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C/C++

```
// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge  Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST
```

```

// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.
key[0] = 0;           // Make key 0 so that this vertex is picked as first vertex
parent[0] = -1; // First node is always root of MST

// The MST will have V vertices
for (int count = 0; count < V-1; count++)
{
    // Pick thd minimum key vertex from the set of vertices
    // not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of the adjacent vertices of
    // the picked vertex. Consider only those vertices which are not yet
    // included in MST
    for (int v = 0; v < V; v++)

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
}

// print the constructed MST
printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
       2      3
     (0)--(1)--(2)
      |   / \ |
      6| 8/   \5 |7
      | /       \ |
     (3)-----(4)
                  9      */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0},
                       };
}

// Print the solution
primMST(graph);

return 0;
}

```

Java

```

// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

class MST
{
    // Number of vertices in the graph
    private static final int V=5;

    // A utility function to find the vertex with minimum key
    // value, from the set of vertices not yet included in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

```

```

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min)
            {
                min = key[v];
                min_index = v;
            }

        return min_index;
    }

// A utility function to print the constructed MST stored in
// parent[]
void printMST(int parent[], int n, int graph[][])
{
    System.out.println("Edge    Weight");
    for (int i = 1; i < V; i++)
        System.out.println(parent[i]+ " - "+ i+ "    "+
                           graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented
// using adjacency matrix representation
void primMST(int graph[][])
{
    // Array to store constructed MST
    int parent[] = new int[V];

    // Key values used to pick minimum weight edge in cut
    int key[] = new int [V];

    // To represent set of vertices not yet included in MST
    Boolean mstSet[] = new Boolean[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
    {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }

    // Always include first 1st vertex in MST.
    key[0] = 0;      // Make key 0 so that this vertex is
                    // picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick thd minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent
        // vertices of the picked vertex. Consider only those
        // vertices which are not yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v]!=0 && mstSet[v] == false &&
                graph[u][v] < key[v])
            {
                parent[v] = u;
                key[v] = graph[u][v];
            }
    }

    // print the constructed MST
    printMST(parent, V, graph);
}

public static void main (String[] args)
{
    /* Let us create the following graph
       2      3
       (0)--(1)--(2)
}

```

```

|   / \
6| 8/ \5 |7
| /   \ |
(3)----- (4)
      9          */
MST t = new MST();
int graph[][] = new int[][] {{0, 2, 0, 6, 0},
                            {2, 0, 3, 8, 5},
                            {0, 3, 0, 0, 7},
                            {6, 8, 0, 0, 9},
                            {0, 5, 7, 9, 0},
                            };
// Print the solution
t.primMST(graph);
}
// This code is contributed by Aakash Hasija

```

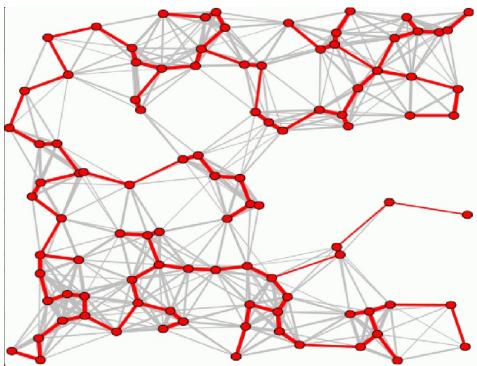
| Edge | Weight |
|-------|--------|
| 0 - 1 | 2 |
| 1 - 2 | 3 |
| 0 - 3 | 6 |
| 1 - 4 | 5 |

Time Complexity of the above program is $O(V^2)$. If the input [graph is represented using adjacency list](#), then the time complexity of Prims algorithm can be reduced to $O(E \log V)$ with the help of binary heap. Please see [Prims MST for Adjacency List Representation](#) for more details.

Applications of Minimum Spanning Tree Problem

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

MST is fundamental problem with diverse applications.



Network design.

telephone, electrical, hydraulic, TV cable, computer, road

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

Approximation algorithms for NP-hard problems.

[traveling salesperson problem](#), [Steiner tree](#)

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, its a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because its a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

Indirect applications.

max bottleneck paths

LDPC codes for error correction

image registration with Renyi entropy

learning salient features for real-time face verification

reducing data storage in sequencing amino acids in a protein

model locality of particle interactions in turbulent fluid flows

autoconfig protocol for Ethernet bridging to avoid cycles in a network

Cluster analysis

k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

Sources:

<http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/mst.pdf>

<http://www.ics.uci.edu/~eppstein/161/960206.html>

Greedy Algorithms | Set 6 (Prims MST for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

[1. Greedy Algorithms | Set 5 \(Prims Minimum Spanning Tree \(MST\)\)](#)

[2. Graph and its representations](#)

We have discussed [Prims algorithm and its implementation for adjacency matrix representation of graphs](#). The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E \log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Prims algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the [cut](#). Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is $O(\log V)$ in Min Heap.

Following are the detailed steps.

1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.

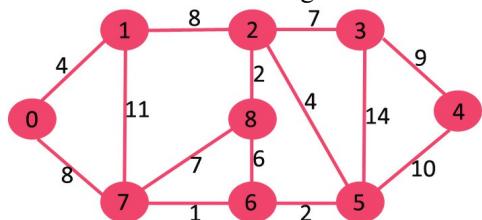
2) Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).

3) While Min Heap is not empty, do following

..a) Extract the min value node from Min Heap. Let the extracted vertex be u .

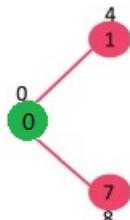
..b) For every adjacent vertex v of u , check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of $u-v$, then update the key value of v as weight of $u-v$.

Let us understand the above algorithm with the following example:

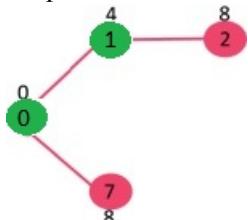


Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

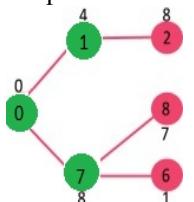
The vertices in green color are the vertices included in MST.



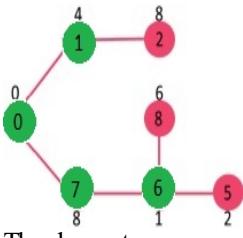
Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.



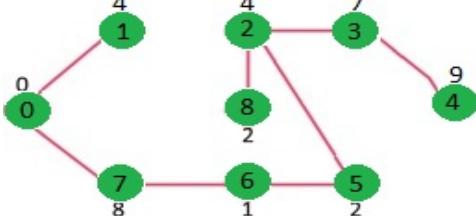
Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.



Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



```
// C / C++ program for Prim's MST for adjacency list representation of graph
```

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency liat
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
}

```

```

graph->array[src].head = newNode;

// Since graph is undirected, add an edge from dest to src also
newNode = newAdjListNode(src, weight);
newNode->next = graph->array[dest].head;
graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int v;
    int key;
};

// Structure to represent a min heap
struct MinHeap
{
    int size;      // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int *pos;      // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int key)
{
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->key = key;
    return minHeapNode;
}

// A utilit function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->key < minHeap->array[smallest]->key )
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->key < minHeap->array[smallest]->key )
        smallest = right;

    if (smallest != idx)
    {
        // The nodes to be swapped in min heap
        MinHeapNode *smallestNode = minHeap->array[smallest];
        MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
    }
}

```

```

        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decrease key value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its key value
    minHeap->array[i]->key = key;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key)
    {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n)
{
    for (int i = 1; i < n; ++i)
        printf("%d - %d\n", arr[i], i);
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algoritm

```

```

void PrimMST(struct Graph* graph)
{
    int V = graph->V; // Get the number of vertices in graph
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. Key value of
    // all vertices (except 0th vertex) is initially infinite
    for (int v = 1; v < V; ++v)
    {
        parent[v] = -1;
        key[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, key[v]);
        minHeap->pos[v] = v;
    }

    // Make key value of 0th vertex as 0 so that it
    // is extracted first
    key[0] = 0;
    minHeap->array[0] = newMinHeapNode(0, key[0]);
    minHeap->pos[0] = 0;

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the following loop, min heap contains all nodes
    // not yet added to MST.
    while (!isEmpty(minHeap))
    {
        // Extract the vertex with minimum key value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Store the extracted vertex number

        // Traverse through all adjacent vertices of u (the extracted
        // vertex) and update their key values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL)
        {
            int v = pCrawl->dest;

            // If v is not yet included in MST and weight of u-v is
            // less than key value of v, then update key value and
            // parent of v
            if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v])
            {
                key[v] = pCrawl->weight;
                parent[v] = u;
                decreaseKey(minHeap, v, key[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print edges of MST
    printArr(parent, V);
}

// Driver program to test above functions
int main()
{
    // Let us create the graph given in above figure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);
}

```

```
PrimMST(graph);  
    return 0;  
}
```

Output:

```
0 - 1  
5 - 2  
2 - 3  
3 - 4  
6 - 5  
7 - 6  
0 - 7  
2 - 8
```

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has decreaseKey() operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V)*O(\log V)$ which is $O((E+V)*\log V) = O(E \log V)$ (For a connected graph, $V = O(E)$)

References:

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

http://en.wikipedia.org/wiki/Prims_algorithm

Greedy Algorithms | Set 2 (Kruskals Minimum Spanning Tree Algorithm)

What is Minimum Spanning Tree?

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

See [this](#) for applications of MST.

Below are the steps for finding MST using Kruskals algorithm

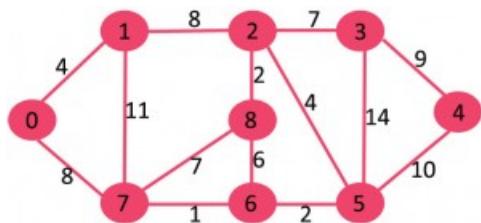
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

The step#2 uses [Union-Find algorithm](#) to detect cycle. So we recommend to read following post as a prerequisite.

[Union-Find Algorithm | Set 1 \(Detect Cycle in a Graph\)](#)

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

After sorting:

| Weight | Src | Dest |
|--------|-----|------|
| 1 | 7 | 6 |
| 2 | 8 | 2 |
| 2 | 6 | 5 |
| 4 | 0 | 1 |
| 4 | 2 | 5 |
| 6 | 8 | 6 |
| 7 | 2 | 3 |
| 7 | 7 | 8 |
| 8 | 0 | 7 |
| 8 | 1 | 2 |
| 9 | 3 | 4 |
| 10 | 5 | 4 |
| 11 | 1 | 7 |
| 14 | 3 | 5 |

Now pick all edges one by one from sorted list of edges

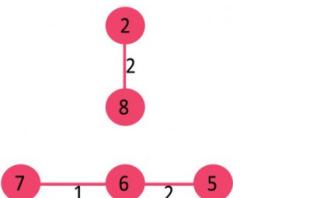
1. Pick edge 7-6: No cycle is formed, include it.



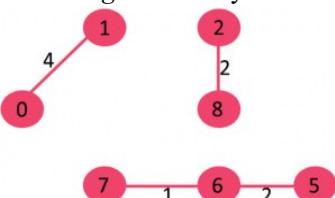
2. Pick edge 8-2: No cycle is formed, include it.



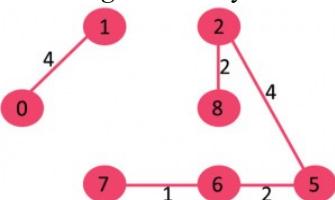
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

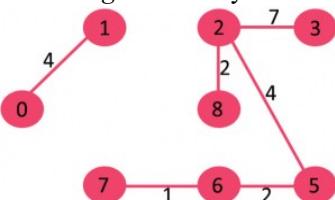


5. Pick edge 2-5: No cycle is formed, include it.



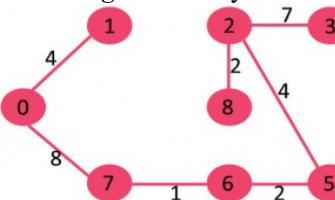
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



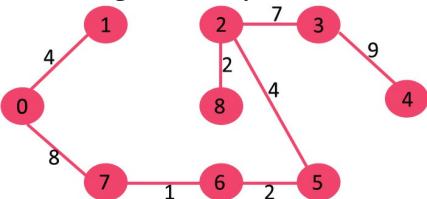
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals ($V - 1$), the algorithm stops here.

C/C++

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
```

```

};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges. Since the graph is
    // undirected, the edge from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V]; // This will store the resultant MST
}

```

```

int e = 0; // An index variable, used for result[]
int i = 0; // An index variable, used for sorted edges

// Step 1: Sort all the edges in non-decreasing order of their weight
// If we are not allowed to change the given graph, we can create a copy of
// array of edges
qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

// Allocate memory for creating V subsets
struct subset *subsets =
    (struct subset*) malloc( V * sizeof(struct subset) );

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment the index
    // for next iteration
    struct Edge next_edge = graph->edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge does't cause cycle, include it
    // in result and increment the index of result for next edge
    if (x != y)
    {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the built MST
printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
    printf("%d -- %d == %d\n", result[i].src, result[i].dest,
           result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
      0-----1
      | \   |
      6|   5\  |15
      |     \ |
      2-----3
      4           */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
}

```

```

graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

KruskalMST(graph);

return 0;
}

```

Java

```

// Java program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

        // Comparator function used for sorting edges based on
        // their weight
        public int compareTo(Edge compareEdge)
        {
            return this.weight - compareEdge.weight;
        }
    };

    // A class to represent a subset for union-find
    class subset
    {
        int parent, rank;
    };

    int V, E;      // V-> no. of vertices & E->no.of edges
    Edge edge[]; // collection of all edges

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // A utility function to find set of an element i
    // (uses path compression technique)
    int find(subset subsets[], int i)
    {
        // find root and make root as parent of i (path compression)
        if (subsets[i].parent != i)
            subsets[i].parent = find(subsets, subsets[i].parent);

        return subsets[i].parent;
    }

    // A function that does union of two sets of x and y
    // (uses union by rank)
    void Union(subset subsets[], int x, int y)
    {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        // Attach smaller rank tree under root of high rank tree
        // (Union by Rank)
        if (subsets[xroot].rank < subsets[yroot].rank)
            subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
            subsets[yroot].parent = xroot;

        // If ranks are same, then make one as root and increment
    }
}

```

```

// its rank by one
else
{
    subsets[yroot].parent = xroot;
    subsets[xroot].rank++;
}
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST()
{
    Edge result[] = new Edge[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges
    for (i=0; i<V; ++i)
        result[i] = new Edge();

    // Step 1: Sort all the edges in non-decreasing order of their
    // weight. If we are not allowed to change the given graph, we
    // can create a copy of array of edges
    Arrays.sort(edge);

    // Allocate memory for creating V ssubsets
    subset subsets[] = new subset[V];
    for(i=0; i<V; ++i)
        subsets[i]=new subset();

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    i = 0; // Index used to pick next edge

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        Edge next_edge = new Edge();
        next_edge = edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the built MST
    System.out.println("Following are the edges in the constructed MST");
    for (i = 0; i < e; ++i)
        System.out.println(result[i].src+" -- "+result[i].dest+" == "+
                           result[i].weight);
}

// Driver Program
public static void main (String[] args)
{

    /* Let us create following weighted graph
       10
      0-----1
      | \   |
      6|   5\  |15
      |     \ |
      2-----3
      4           */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    Graph graph = new Graph(V, E);

    // add edge 0-1

```

```

graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;

// add edge 0-2
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;

// add edge 0-3
graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;

// add edge 1-3
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;

// add edge 2-3
graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;

graph.KruskalMST();
}
}

//This code is contributed by Aakash Hasija

```

Following are the edges in the constructed MST

2 -- 3 == 4
 0 -- 3 == 5
 0 -- 1 == 10

Time Complexity: $O(E \log E)$ or $O(E \log V)$. Sorting of edges takes $O(E \log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E \log E + E \log V)$ time. The value of E can be atmost V^2 , so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E \log E)$ or $O(E \log V)$

References:

<http://www.ics.uci.edu/~eppstein/161/960206.html>
http://en.wikipedia.org/wiki/Minimum_spanning_tree

Greedy Algorithms | Set 9 (Boruvkas algorithm)

We have discussed following topics on Minimum Spanning Tree.

[Applications of Minimum Spanning Tree Problem](#)

[Kruskals Minimum Spanning Tree Algorithm](#)

[Prims Minimum Spanning Tree Algorithm](#)

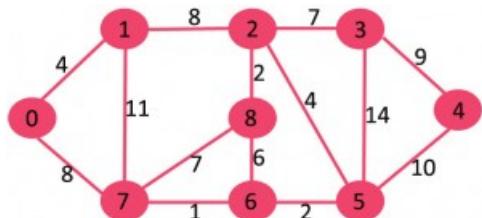
In this post, Boruvkas algorithm is discussed. Like Prims and Kruskals, Boruvkas algorithm is also a Greedy algorithm. Below is complete algorithm.

- 1) Input is a connected, weighted and directed graph.
- 2) Initialize all vertices as individual components (or sets).
- 3) Initialize MST as empty.
- 4) While there are more than one components, do following for each component.
 - a) Find the closest weight edge that connects this component to any other component.
 - b) Add this closest edge to MST if not already added.
- 5) Return MST.

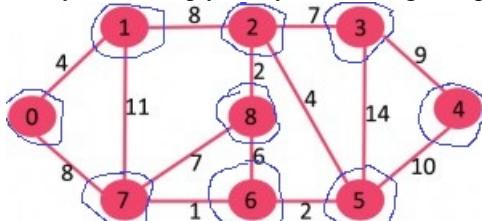
Below is the idea behind above algorithm (The idea is same as [Prims MST algorithm](#)).

A spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Let us understand the algorithm with below example.



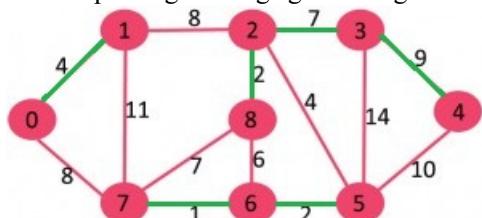
Initially MST is empty. Every vertex is single component as highlighted in blue color in below diagram



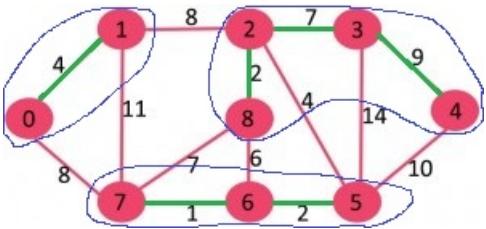
For every component, find the cheapest edge that connects it to some other component.

| Component | Cheapest Edge that connects it to some other component |
|-----------|--|
| {0} | 0-1 |
| {1} | 0-1 |
| {2} | 2-8 |
| {3} | 2-3 |
| {4} | 3-4 |
| {5} | 5-6 |
| {6} | 6-7 |
| {7} | 6-7 |
| {8} | 2-8 |

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.



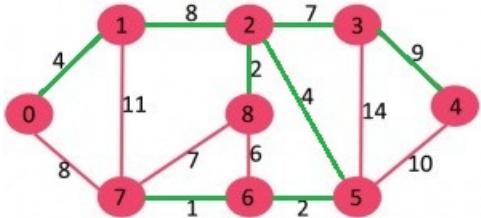
After above step, components are $\{0,1\}, \{2,3,4,8\}, \{5,6,7\}\}$. The components are encircled with blue color.



We again repeat the step, i.e., for every component, find the cheapest edge that connects it to some other component.

| Component | Cheapest Edge that connects it to some other component |
|-----------|--|
| {0,1} | 1-2 (or 0-7) |
| {2,3,4,8} | 2-5 |
| {5,6,7} | 2-5 |

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5}



At this stage, there is only one component {0, 1, 2, 3, 4, 5, 6, 7, 8} which has all edges. Since there is only one component left, we stop and return MST.

Implementation:

Below is C++ implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

```

// Boruvka's algorithm to find Minimum Spanning
// Tree of a given connected, undirected and
// weighted graph
#include <stdio.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after boruvkaMST() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// The main function for MST using Boruvka's algorithm
void boruvkaMST(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;
}

```

```

// Allocate memory for creating V subsets.
struct subset *subsets = new subset[V];

// An array to store index of the cheapest edge of
// subset. The stored index for indexing array 'edge[]'
int *cheapest = new int[V];

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
    cheapest[v] = -1;
}

// Initially there are V different trees.
// Finally there will be one tree that will be MST
int numTrees = V;
int MSTweight = 0;

// Keep combining components (or sets) until all
// components are not combined into single MST.
while (numTrees > 1)
{
    // Traverse through all edges and update
    // cheapest of every component
    for (int i=0; i<E; i++)
    {
        // Find components (or sets) of two corners
        // of current edge
        int set1 = find(subsets, edge[i].src);
        int set2 = find(subsets, edge[i].dest);

        // If two corners of current edge belong to
        // same set, ignore current edge
        if (set1 == set2)
            continue;

        // Else check if current edge is closer to previous
        // cheapest edges of set1 and set2
        else
        {
            if (cheapest[set1] == -1 ||
                edge[cheapest[set1]].weight > edge[i].weight)
                cheapest[set1] = i;

            if (cheapest[set2] == -1 ||
                edge[cheapest[set2]].weight > edge[i].weight)
                cheapest[set2] = i;
        }
    }

    // Consider the above picked cheapest edges and add them
    // to MST
    for (int i=0; i<V; i++)
    {
        // Check if cheapest for current set exists
        if (cheapest[i] != -1)
        {
            int set1 = find(subsets, edge[cheapest[i]].src);
            int set2 = find(subsets, edge[cheapest[i]].dest);

            if (set1 == set2)
                continue;
            MSTweight += edge[cheapest[i]].weight;
            printf("Edge %d-%d included in MST\n",
                  edge[cheapest[i]].src, edge[cheapest[i]].dest,
                  edge[cheapest[i]].weight);

            // Do a union of set1 and set2 and decrease number
            // of trees
            Union(subsets, set1, set2);
            numTrees--;
        }
    }
}

printf("Weight of MST is %d\n", MSTweight);
return;
}

```

```

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent =
            find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
      0-----1
      | \   |
      6|   5\  |15
      |     \ |
      2-----3
      4           */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
}

```

```

graph->edge[3].weight = 15;
// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

boruvkaMST(graph);

return 0;
}

```

Output:

```

Edge 0-3 included in MST
Edge 0-1 included in MST
Edge 2-3 included in MST
Weight of MST is 19

```

Interesting Facts about Boruvkas algorithm:

- 1) Time Complexity of Boruvkas algorithm is $O(E \log V)$ which is same as Kruskals and Prims algorithms.
- 2) Boruvkas algorithm is used as a step in a [faster randomized algorithm that works in linear time \$O\(E\)\$](#) .
- 3) Boruvkas algorithm is the oldest minimum spanning tree algorithm was discovered by Boruvka in 1926, long before computers even existed. The algorithm was published as a method of constructing an efficient electricity network.

Exercise:

The above code assumes that input graph is connected and it fails if a disconnected graph is given. Extend the above algorithm so that it works for a disconnected graph also and produces a forest.

References:

http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Greedy Algorithms | Set 7 (Dijkstras shortest path algorithm)

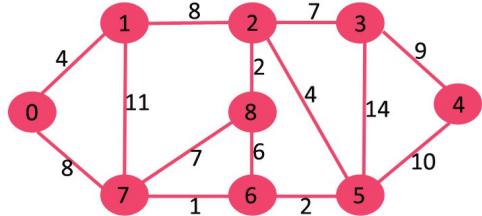
Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstras algorithm is very similar to [Prims algorithm for minimum spanning tree](#). Like Prims MST, we generate a *SPT* (*shortest path tree*) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

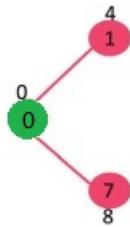
Below are the detailed steps used in Dijkstras algorithm to find the shortest path from a single source vertex to all other vertices in the given graph. Algorithm

- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While *sptSet* doesn't include all vertices
 - a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.
 - b) Include *u* to *sptSet*.
 - c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

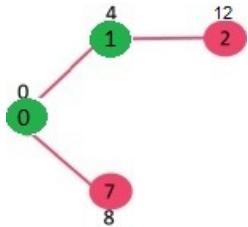
Let us understand with the following example:



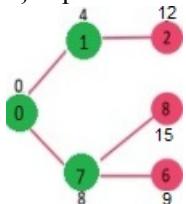
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.



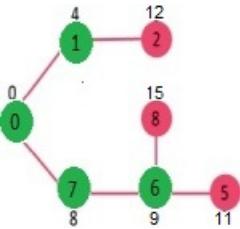
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



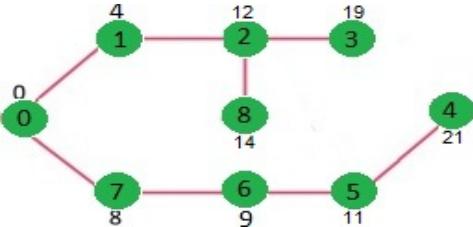
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). Vertex 7 is picked. So *sptSet* now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). Vertex 6 is picked. So *sptSet* now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

We use a boolean array *sptSet[]* to represent the set of vertices included in SPT. If a value *sptSet[v]* is true, then vertex *v* is included in SPT, otherwise not. Array *dist[]* is used to store shortest distance values of all vertices.

C/C++

```
// A C / C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V];      // The output array. dist[i] will hold the shortest
                      // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum distance vertex from the set of vertices not
```

```

// yet processed. u is always equal to src in first iteration.
int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed
sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if is not in sptSet, there is an edge from
    // u to v, and total weight of path from src to v through u is
    // smaller than current value of dist[v]
    if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
        && dist[u]+graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
    }

// print the constructed distance array
printSolution(dist, V);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                        {4, 0, 8, 0, 0, 0, 0, 11, 0},
                        {0, 8, 0, 7, 0, 4, 0, 0, 2},
                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                        {0, 0, 4, 0, 10, 0, 2, 0, 0},
                        {0, 0, 0, 14, 0, 2, 0, 1, 6},
                        {8, 11, 0, 0, 0, 0, 1, 0, 7},
                        {0, 0, 2, 0, 0, 0, 6, 7, 0}
                    };

    dijkstra(graph, 0);

    return 0;
}

```

Java

```

// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // A utility function to find the vertex with minimum distance value,
    // from the set of vertices not yet included in shortest path tree
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
            {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex   Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i+" \t\t "+dist[i]);
    }

    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation

```

```

void dijkstra(int graph[][], int src)
{
    int dist[] = new int[V]; // The output array. dist[i] will hold
                           // the shortest distance from src to i

    // sptSet[i] will true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized
    Boolean sptSet[] = new Boolean[V];

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
    {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum distance vertex from the set of vertices
        // not yet processed. u is always equal to src in first
        // iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the
        // picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an
            // edge from u to v, and total weight of path from src to
            // v through u is smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v]!=0 &&
                dist[u] != Integer.MAX_VALUE &&
                dist[u]+graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}

// Driver method
public static void main (String[] args)
{
/* Let us create the example graph discussed above */
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                               {4, 0, 8, 0, 0, 0, 0, 11, 0},
                               {0, 8, 0, 7, 0, 4, 0, 0, 2},
                               {0, 0, 7, 0, 9, 14, 0, 0, 0},
                               {0, 0, 0, 9, 0, 10, 0, 0, 0},
                               {0, 0, 4, 0, 10, 0, 2, 0, 0},
                               {0, 0, 0, 14, 0, 2, 0, 1, 6},
                               {8, 11, 0, 0, 0, 0, 1, 0, 7},
                               {0, 0, 2, 0, 0, 0, 6, 7, 0}
                            };
    ShortestPath t = new ShortestPath();
    t.dijkstra(graph, 0);
}
//This code is contributed by Aakash Hasija

```

| Vertex | Distance from Source |
|--------|----------------------|
| 0 | 0 |
| 1 | 4 |
| 2 | 12 |
| 3 | 19 |
| 4 | 21 |
| 5 | 11 |
| 6 | 9 |
| 7 | 8 |
| 8 | 14 |

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prims implementation](#)) and use it to show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
- 4) Time Complexity of the implementation is $O(V^2)$. If the input [graph is represented using adjacency list](#), it can be reduced to $O(E \log V)$ with the help of binary heap. Please see [Dijkstras Algorithm for Adjacency List Representation](#) for more details.
- 5) Dijkstras algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [BellmanFord algorithm](#) can be used, we will soon be discussing it as a separate post.

[Dijkstras Algorithm for Adjacency List Representation](#)

Greedy Algorithms | Set 8 (Dijkstras Algorithm for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

[1. Greedy Algorithms | Set 7 \(Dijkstras shortest path algorithm\)](#)

[2. Graph and its representations](#)

We have discussed [Dijkstras algorithm and its implementation for adjacency matrix representation of graphs](#). The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E \log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Dijkstras algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is $O(\log V)$ for Min Heap.

Following are the detailed steps.

1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.

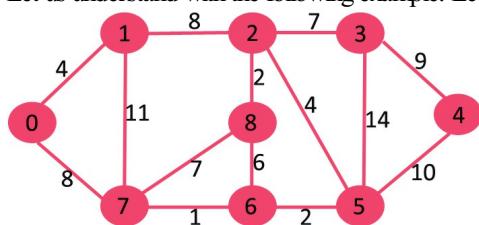
2) Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).

3) While Min Heap is not empty, do following

..a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u .

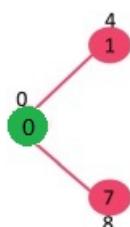
..b) For every adjacent vertex v of u , check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of $u-v$ plus distance value of u , then update the distance value of v .

Let us understand with the following example. Let the given source vertex be 0

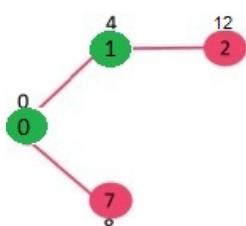


Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

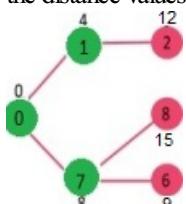
The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



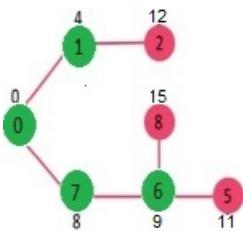
Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



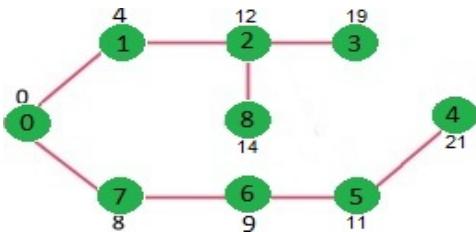
Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Above steps are repeated till min heap doesn't become empty. Finally, we get the following shortest path tree.



```
// C / C++ program for Dijkstra's shortest path algorithm for adjacency
// list representation of graph

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}
```

```

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int v;
    int dist;
};

// Structure to represent a min heap
struct MinHeap
{
    int size;          // Number of heap nodes present currently
    int capacity;     // Capacity of min heap
    int *pos;          // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->dist < minHeap->array[smallest]->dist )
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->dist < minHeap->array[smallest]->dist )
        smallest = right;

    if (smallest != idx)

```

```

{
    // The nodes to be swapped in min heap
    MinHeapNode *smallestNode = minHeap->array[smallest];
    MinHeapNode *idxNode = minHeap->array[idx];

    // Swap positions
    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;

    // Swap nodes
    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

    minHeapify(minHeap, smallest);
}
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy dist value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
    // Get the index of v in  heap array
    int i = minHeap->pos[v];

    // Get the node and update its dist value
    minHeap->array[i]->dist = dist;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)
    {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n)

```

```

{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t %d\n", i, dist[i]);
}

// The main function that calculates distances of shortest paths from src to all
// vertices. It is a O(ELogV) function
void dijkstra(struct Graph* graph, int src)
{
    int V = graph->V;// Get the number of vertices in graph
    int dist[V];      // dist values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. dist value of all vertices
    for (int v = 0; v < V; ++v)
    {
        dist[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, dist[v]);
        minHeap->pos[v] = v;
    }

    // Make dist value of src vertex as 0 so that it is extracted first
    minHeap->array[src] = newMinHeapNode(src, dist[src]);
    minHeap->pos[src] = src;
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]);

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the followin loop, min heap contains all nodes
    // whose shortest distance is not yet finalized.
    while (!isEmpty(minHeap))
    {
        // Extract the vertex with minimum distance value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Store the extracted vertex number

        // Traverse through all adjacent vertices of u (the extracted
        // vertex) and update their distance values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL)
        {
            int v = pCrawl->dest;

            // If shortest distance to v is not finalized yet, and distance to v
            // through u is less than its previously calculated distance
            if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
                pCrawl->weight + dist[u] < dist[v])
            {
                dist[v] = dist[u] + pCrawl->weight;

                // update distance value in min heap also
                decreaseKey(minHeap, v, dist[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print the calculated shortest distances
    printArr(dist, V);
}

// Driver program to test above functions
int main()
{
    // create the graph given in above fugure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
}

```

```

addEdge(graph, 3, 5, 14);
addEdge(graph, 4, 5, 10);
addEdge(graph, 5, 6, 2);
addEdge(graph, 6, 7, 1);
addEdge(graph, 6, 8, 6);
addEdge(graph, 7, 8, 7);

dijkstra(graph, 0);

return 0;
}

```

Output:

| Vertex | Distance from Source |
|--------|----------------------|
| 0 | 0 |
| 1 | 4 |
| 2 | 12 |
| 3 | 19 |
| 4 | 21 |
| 5 | 11 |
| 6 | 9 |
| 7 | 8 |
| 8 | 14 |

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has `decreaseKey()` operation which takes $O(\log V)$ time. So overall time complexity is $O((E+V)*O(\log V))$ which is $O((E+V)*\log V) = O(E \log V)$

Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + V \log V)$ using Fibonacci Heap. The reason is, Fibonacci Heap takes $O(1)$ time for decrease-key operation while Binary Heap takes $O(\log n)$ time.

Notes:

- 1)The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prims implementation](#)) and use it to show the shortest path from source to different vertices.
- 2)The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3)The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
- 4)Dijkstras algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [BellmanFord algorithm](#) can be used, we will soon be discussing it as a separate post.

References:

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)
[Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani](#)

Dynamic Programming | Set 23 (BellmanFord Algorithm)

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed [Dijkstras algorithm](#) for this problem. Dijkstras algorithm is a Greedy algorithm and time complexity is $O(V \log V)$ (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is $O(VE)$, which is more than Dijkstra.*

Algorithm

Following are the detailed steps.

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src . If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array $dist[]$ of size $|V|$ with all values as infinite except $dist[src]$ where src is source vertex.

2) This step calculates shortest distances. Do following $|V|-1$ times where $|V|$ is the number of vertices in given graph.

..a) Do following for each edge $u-v$

If $dist[v] > dist[u] + \text{weight of edge } uv$, then update $dist[v]$

$.dist[v] = dist[u] + \text{weight of edge } uv$

3) This step reports if there is a negative weight cycle in graph. Do following for each edge $u-v$

If $dist[v] > dist[u] + \text{weight of edge } uv$, then Graph contains negative weight cycle

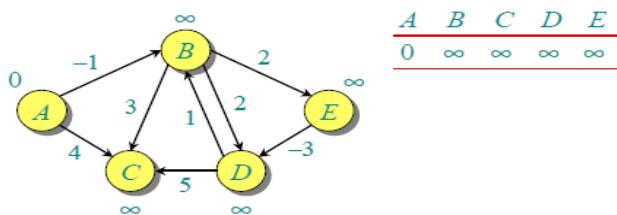
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum $|V| - 1$ edges in any simple path, that is why the outer loop runs $|V| - 1$ times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most $(i+1)$ edges (Proof is simple, you can refer [this](#) or [MIT Video Lecture](#))

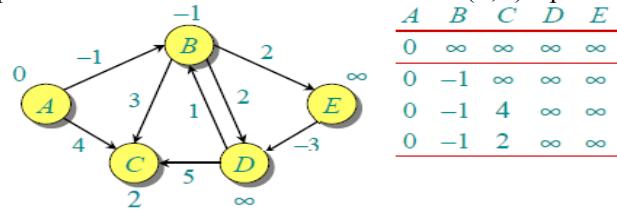
Example

Let us understand the algorithm with following example graph. The images are taken from [this](#) source.

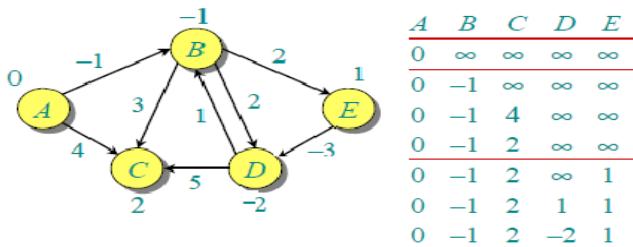
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Implementation:

C++

```
// A C / C++ program for Bellman-Ford's single source
// shortest path algorithm.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;
```

```

// Step 2: Relax all edges |V| - 1 times. A simple shortest
// path from src to any other vertex can have at-most |V| - 1
// edges
for (int i = 1; i <= V-1; i++)
{
    for (int j = 0; j < E; j++)
    {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}

// Step 3: check for negative-weight cycles. The above step
// guarantees shortest distances if graph doesn't contain
// negative weight cycle. If we get a shorter path, then there
// is a cycle.
for (int i = 0; i < E; i++)
{
    int u = graph->edge[i].src;
    int v = graph->edge[i].dest;
    int weight = graph->edge[i].weight;
    if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        printf("Graph contains negative weight cycle");
}

printArr(dist, V);

return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;
}

```

```

    BellmanFord(graph, 0);

    return 0;
}

```

Java

```

// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };
    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm. The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];

        // Step 1: Initialize distances from src to all other
        // vertices as INFINITE
        for (int i=0; i<V; ++i)
            dist[i] = Integer.MAX_VALUE;
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i=1; i<V; ++i)
        {
            for (int j=0; j<E; ++j)
            {
                int u = graph.edge[j].src;
                int v = graph.edge[j].dest;
                int weight = graph.edge[j].weight;
                if (dist[u]!=Integer.MAX_VALUE &&
                    dist[u]+weight<dist[v])
                    dist[v]=dist[u]+weight;
            }
        }

        // Step 3: check for negative-weight cycles. The above
        // step guarantees shortest distances if graph doesn't
        // contain negative weight cycle. If we get a shorter
        // path, then there is a cycle.
        for (int j=0; j<E; ++j)
        {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u]!=Integer.MAX_VALUE &&
                dist[u]+weight<dist[v])
                System.out.println("Graph contains negative weight cycle");
        }
    }
}

```

```

    }
    printArr(dist, V);
}

// A utility function used to print the solution
void printArr(int dist[], int V)
{
    System.out.println("Vertex      Distance from Source");
    for (int i=0; i<V; ++i)
        System.out.println(i+"\t\t"+dist[i]);
}

// Driver method to test above function
public static void main(String[] args)
{
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph

    Graph graph = new Graph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;
    graph.edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph.edge[1].src = 0;
    graph.edge[1].dest = 2;
    graph.edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph.edge[2].src = 1;
    graph.edge[2].dest = 2;
    graph.edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph.edge[3].src = 1;
    graph.edge[3].dest = 3;
    graph.edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph.edge[4].src = 1;
    graph.edge[4].dest = 4;
    graph.edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph.edge[5].src = 3;
    graph.edge[5].dest = 2;
    graph.edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph.edge[6].src = 3;
    graph.edge[6].dest = 1;
    graph.edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph.edge[7].src = 4;
    graph.edge[7].dest = 3;
    graph.edge[7].weight = -3;

    graph.BellmanFord(graph, 0);
}
}
// Contributed by Aakash Hasija

```

| Vertex | Distance from Source |
|--------|----------------------|
| 0 | 0 |
| 1 | -1 |
| 2 | 2 |
| 3 | -2 |
| 4 | 1 |

Notes

- 1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
- 2) Bellman-Ford works better (better than Dijksras) for distributed systems. Unlike Dijksras where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Exercise

- 1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
- 2) Can we use Dijksras algorithm for shortest paths for graphs with negative weights one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksras algorithm for the modified graph. Will this algorithm work?

References:

<http://www.youtube.com/watch?v=Ttezuzs39nk>
http://en.wikipedia.org/wiki/Bellman%20%26%2393Ford_algorithm
<http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.ppt.pdf>

Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

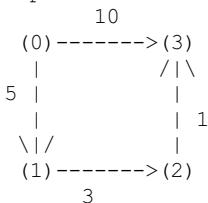
The [Floyd Warshall Algorithm](#) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
graph[][] = { {0, 5, INF, 10},
              {INF, 0, 3, INF},
              {INF, INF, 0, 1},
              {INF, INF, INF, 0} }
```

which represents the following graph



Note that the value of $graph[i][j]$ is 0 if i is equal to j .
And $graph[i][j]$ is INF (infinite) if there is no edge from vertex i to j .

Output:

Shortest distance matrix

| | | | |
|-----|-----|-----|---|
| 0 | 5 | 8 | 9 |
| INF | 0 | 3 | 4 |
| INF | INF | 0 | 1 |
| INF | INF | INF | 0 |

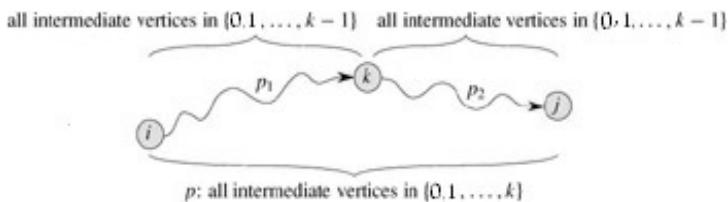
Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j . We keep the value of $dist[i][j]$ as it is.

2) k is an intermediate vertex in shortest path from i to j . We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$.

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm

C/C++

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
   for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
       distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
}
```

```

for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
        dist[i][j] = graph[i][j];

/* Add all vertices one by one to the set of intermediate vertices.
--> Before start of a iteration, we have shortest distances between all
pairs of vertices such that the shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of a iteration, vertex no. k is added to the set of
intermediate vertices and the set becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
           " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
          |           /|\
          |           |
          |           | 1
          \|/           |
       (1)----->(2)
          3           */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
}

```

Java

```

// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

```

```

class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
         * Or we can say the initial values of shortest distances
         * are based on shortest paths considering no intermediate
         * vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
         * vertices.
         *--> Before start of a iteration, we have shortest
             distances between all pairs of vertices such that
             the shortest distances consider only the vertices in
             set {0, 1, 2, .. k-1} as intermediate vertices.
         *--> After the end of a iteration, vertex no. k is added
             to the set of intermediate vertices and the set
             becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)
            {
                // Pick all vertices as destination for the
                // above picked source
                for (j = 0; j < V; j++)
                {
                    // If vertex k is on the shortest path from
                    // i to j, then update the value of dist[i][j]
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }

        // Print the shortest distance matrix
        printSolution(dist);
    }

    void printSolution(int dist[][])
    {
        System.out.println("Following matrix shows the shortest "+
                           "distances between every pair of vertices");
        for (int i=0; i<V; ++i)
        {
            for (int j=0; j<V; ++j)
            {
                if (dist[i][j]==INF)
                    System.out.print("INF ");
                else
                    System.out.print(dist[i][j]+"   ");
            }
            System.out.println();
        }
    }

    // Driver program to test above function
    public static void main (String[] args)
    {
        /* Let us create the following weighted graph
           10
           (0)----->(3)
           |           /|\
           5 |           |
           |           | 1
           \|/          |
           (1)----->(2)
           3           */
        int graph[][] = { {0, 5, INF, 10},
                         {INF, 0, 3, INF},
                         {INF, INF, 0, 1},
                         {INF, INF, INF, 0}
                     };
    }
}

```

```

        };
AllPairShortestPath a = new AllPairShortestPath();

// Print the solution
a.floydWarshall(graph);
}

// Contributed by Aakash Hasija

```

Output:

Following matrix shows the shortest distances between every pair of vertices

| | | | |
|-----|-----|-----|---|
| 0 | 5 | 8 | 9 |
| INF | 0 | 3 | 4 |
| INF | INF | 0 | 1 |
| INF | INF | INF | 0 |

Time Complexity: $O(V^3)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```

#include<limits.h>

#define INF INT_MAX
.....
if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j])
    dist[i][j] = dist[i][k] + dist[k][j];
.....

```

Johnsons algorithm for All-pairs shortest paths

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed [Floyd Warshall Algorithm](#) for this problem. Time complexity of Floyd Warshall Algorithm is $\Theta(V^3)$. Using Johnsons algorithm, we can find all pair shortest paths in $O(V^2 \log V + VE)$ time. Johnsons algorithm uses both [Dijkstra](#) and [Bellman-Ford](#) as subroutines.

If we apply [Dijkstras Single Source shortest path algorithm](#) for every vertex, considering every vertex as source, we can find all pair shortest paths in $O(V^2 \log V)$ time. So using Dijkstras single source shortest path seems to be a better option than [Floyd Warshell](#), but the problem with Dijkstras algorithm is, it doesn't work for negative weight edge.

The idea of Johnsons algorithm is to re-weight all edges and make them all positive, then apply Dijkstras algorithm for every vertex.

How to transform a given graph to a graph with all non-negative weight edges?

One may think of a simple approach of finding the minimum weight edge and adding this weight to all edges. Unfortunately, this doesn't work as there may be different number of edges in different paths (See [this](#) for an example). If there are multiple paths from a vertex u to v, then all paths must be increased by same amount, so that the shortest path remains the shortest in the transformed graph.

The idea of Johnsons algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be $h[u]$. We reweight edges using vertex weights. For example, for an edge (u, v) of weight $w(u, v)$, the new weight becomes $w(u, v) + h[u] - h[v]$. The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by $h[s] - h[t]$, all $h[]$ values of vertices on path from s to t cancel each other.

How do we calculate $h[]$ values? [Bellman-Ford algorithm](#) is used for this purpose. Following is the complete algorithm. A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are $h[]$ values.

Algorithm:

- 1) Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G'.
- 2) Run [Bellman-Ford algorithm](#) on G' with s as source. Let the distances calculated by Bellman-Ford be $h[0], h[1], \dots, h[V-1]$. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.
- 3) Reweight the edges of original graph. For each edge (u, v) , assign the new weight as original weight + $h[u] - h[v]$.
- 4) Remove the added vertex s and run [Dijkstras algorithm](#) for every vertex.

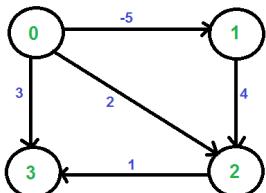
How does the transformation ensure nonnegative weight edges?

The following property is always true about $h[]$ values as they are shortest distances.

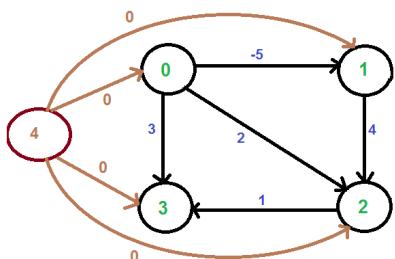
$$h[v] \leq h[u] + w(u, v)$$

The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge (u, v) . The new weights are $w(u, v) + h[u] - h[v]$. The value of the new weights must be greater than or equal to zero because of the inequality " $h[v] \leq h[u] + w(u, v)$ ". **Example:**

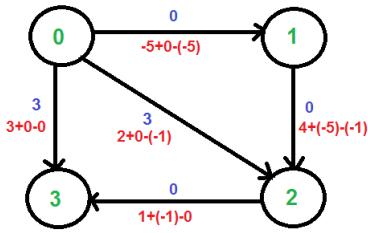
Let us consider the following graph.



We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.



We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e., $h[] = \{0, -5, -1, 0\}$. Once we get these distances, we remove the source vertex 4 and reweight the edges using following formula. $w'(u, v) = w(u, v) + h[u] - h[v]$.



Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively.

Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as source.

Time Complexity: The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is $O(VE)$ and time complexity of Dijkstra is $O(V\log V)$. So overall time complexity is $O(V^2 \log V + VE)$.

The time complexity of Johnson's algorithm becomes same as [Floyd Warshell](#) when the graphs is complete (For a complete graph $E = O(V^2)$). But for sparse graphs, the algorithm performs much better than [Floyd Warshell](#).

References:

- [Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)
- <http://www.youtube.com/watch?v=b6LOHvCznkI>
- <http://www.youtube.com/watch?v=TV2Z6nbo1ic>
- http://en.wikipedia.org/wiki/Johnson%27s_algorithm
- <http://www.youtube.com/watch?v=Sygg1e0xWnM>

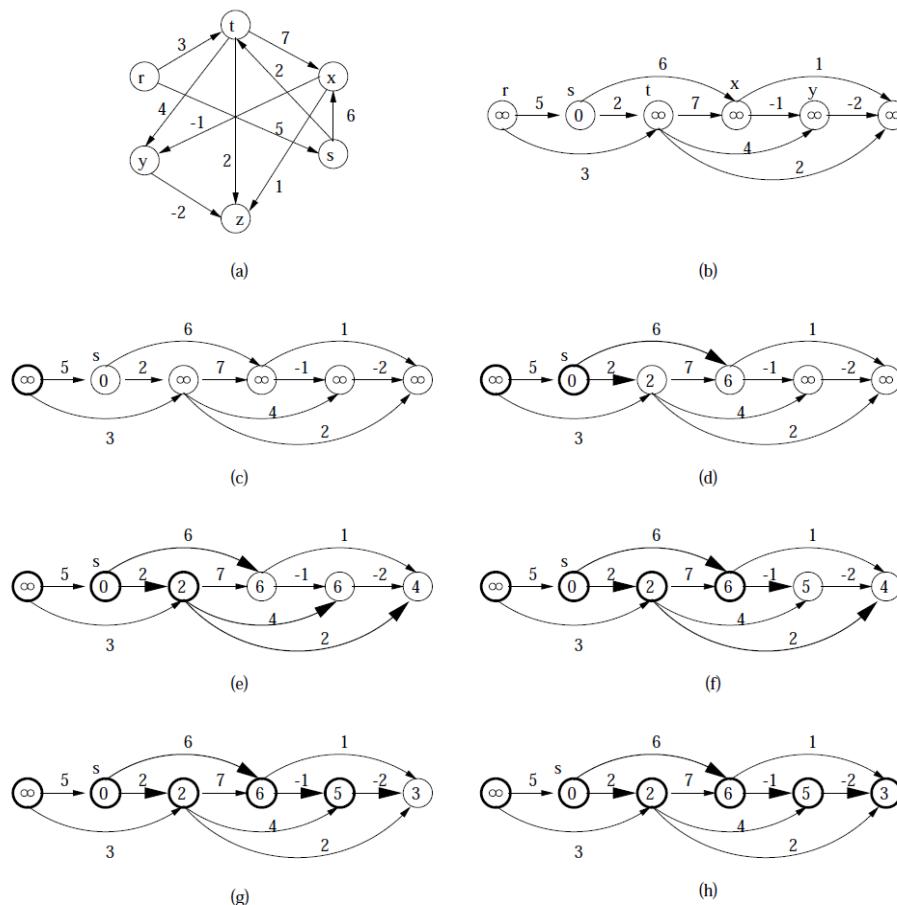
Shortest Path in Directed Acyclic Graph

Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.

For a general weighted graph, we can calculate single source shortest distances in $O(VE)$ time using [BellmanFord Algorithm](#). For a graph with no negative weights, we can do better and calculate single source shortest distances in $O(E + V\log V)$ time using [Dijkstras algorithm](#). Can we do even better for Directed Acyclic Graph (DAG)? We can calculate single source shortest distances in $O(V+E)$ time for DAGs. The idea is to use [Topological Sorting](#).

We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](#) of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure is taken from [this](#) source. It shows step by step process of finding shortest paths.



Following is complete algorithm for finding shortest distances.

1) Initialize $\text{dist}[] = \{\text{INF}, \text{INF}, \dots\}$ and $\text{dist}[s] = 0$ where s is the source vertex.

2) Create a toplogical order of all vertices.

3) Do following for every vertex u in topological order.

..Do following for every adjacent vertex v of u

$\text{if}(\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v))$

$\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$

C++

```
// C++ program to find single source shortest paths for Directed Acyclic Graphs
#include<iostream>
#include <list>
#include <stack>
#include <limits.h>
#define INF INT_MAX
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
public:
    AdjListNode(int v, int weight)
    {
        this->v = v;
        this->weight = weight;
    }
};
```

```

    int weight;
public:
    AdjListNode(int _v, int _w) { v = _v; weight = _w; }
    int getV() { return v; }
    int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
    int V;      // No. of vertices'

    // Pointer to an array containing adjacency lists
    list<AdjListNode> *adj;

    // A function used by shortestPath
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds shortest paths from given source vertex
    void shortestPath(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by shortestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological sort
    Stack.push(v);
}

// The function to find shortest paths from given vertex. It uses recursive
// topologicalSortUtil() to get topological sorting of given graph.
void Graph::shortestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance
    // to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = INF;
}

```

```

dist[s] = 0;

// Process vertices in topological order
while (Stack.empty() == false)
{
    // Get the next vertex from topological order
    int u = Stack.top();
    Stack.pop();

    // Update distances of all adjacent vertices
    list<AdjListNode>::iterator i;
    if (dist[u] != INF)
    {
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (dist[i->getV()] > dist[u] + i->getWeight())
                dist[i->getV()] = dist[u] + i->getWeight();
    }
}

// Print the calculated shortest distances
for (int i = 0; i < V; i++)
    (dist[i] == INF)? cout << "INF ": cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram. Here vertex numbers are
    // 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
    Graph g(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    cout << "Following are shortest distances from source " << s << "\n";
    g.shortestPath(s);

    return 0;
}

```

Java

```

// Java program to find single source shortest paths in Directed Acyclic Graphs
import java.io.*;
import java.util.*;

class ShortestPath
{
    static final int INF=Integer.MAX_VALUE;
    class AdjListNode
    {
        private int v;
        private int weight;
        AdjListNode(int _v, int _w) { v = _v; weight = _w; }
        int getV() { return v; }
        int getWeight() { return weight; }
    }

    // Class to represent graph as an adjacency list of
    // nodes of type AdjListNode
    class Graph
    {
        private int v;
        private LinkedList<AdjListNode>adj[];
        Graph(int v)
        {
            v=v;
            adj = new LinkedList[v];
            for (int i=0; i<v; ++i)
                adj[i] = new LinkedList<AdjListNode>();
        }
        void addEdge(int u, int v, int weight)

```

```

{
    AdjListNode node = new AdjListNode(v,weight);
    adj[u].add(node); // Add v to u's list
}

// A recursive function used by shortestPath.
// See below link for details
void topologicalSortUtil(int v, Boolean visited[], Stack stack)
{
    // Mark the current node as visited.
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this vertex
    Iterator<AdjListNode> it = adj[v].iterator();
    while (it.hasNext())
    {
        AdjListNode node = it.next();
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, stack);
    }
    // Push current vertex to stack which stores result
    stack.push(new Integer(v));
}

// The function to find shortest paths from given vertex. It
// uses recursive topologicalSortUtil() to get topological
// sorting of given graph.
void shortestPath(int s)
{
    Stack stack = new Stack();
    int dist[] = new int[V];

    // Mark all the vertices as not visited
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

    // Initialize distances to all vertices as infinite and
    // distance to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = INF;
    dist[s] = 0;

    // Process vertices in topological order
    while (stack.empty() == false)
    {
        // Get the next vertex from topological order
        int u = (int)stack.pop();

        // Update distances of all adjacent vertices
        Iterator<AdjListNode> it;
        if (dist[u] != INF)
        {
            it = adj[u].iterator();
            while (it.hasNext())
            {
                AdjListNode i= it.next();
                if (dist[i.getV()] > dist[u] + i.getWeight())
                    dist[i.getV()] = dist[u] + i.getWeight();
            }
        }
    }

    // Print the calculated shortest distances
    for (int i = 0; i < V; i++)
    {
        if (dist[i] == INF)
            System.out.print( "INF ");
        else
            System.out.print( dist[i] + " ");
    }
}
}

```

```

// Method to create a new graph instance through an object
// of ShortestPath class.
Graph newGraph(int number)
{
    return new Graph(number);
}

public static void main(String args[])
{
    // Create a graph given in the above diagram. Here vertex
    // numbers are 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
    ShortestPath t = new ShortestPath();
    Graph g = t.newGraph(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    System.out.println("Following are shortest distances "+
                       "from source " + s );
    g.shortestPath(s);
}
//This code is contributed by Aakash Hasija

```

Following are shortest distances from source 1
INF 0 2 6 5 3

Time Complexity: Time complexity of topological sorting is $O(V+E)$. After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is $O(E)$. So the inner loop runs $O(V+E)$ times. Therefore, overall time complexity of this algorithm is $O(V+E)$.

References:

<http://www.utdallas.edu/~sizheng/CS4349.d/l-notes.d/L17.pdf>

Some interesting shortest path questions | Set 1

Question 1: Given a directed weighted graph. You are also given the shortest path from a source vertex s to a destination vertex t. If weight of every edge is increased by 10 units, does the shortest path remain same in the modified graph?

The shortest path may change. The reason is, there may be different number of edges in different paths from s to t. For example, let shortest path be of weight 15 and has 5 edges. Let there be another path with 2 edges and total weight 25. The weight of the shortest path is increased by 5×10 and becomes $15 + 50$. Weight of the other path is increased by 2×10 and becomes $25 + 20$. So the shortest path changes to the other path with weight as 45.

Question 2: This is similar to above question. Does the shortest path change when weights of all edges are multiplied by 10?

If we multiply all edge weights by 10, the shortest path doesn't change. The reason is simple, weights of all paths from s to t get multiplied by same amount. The number of edges on a path doesn't matter. It is like changing unit of weights.

Question 3: Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex s to a given destination vertex t. Expected time complexity is $O(V+E)$.

If we apply [Dijkstra's shortest path algorithm](#), we can get a shortest path in $O(E + V\log V)$ time. How to do it in $O(V+E)$ time? The idea is to use [BFS](#). One important observation about [BFS](#) is, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight 2 into two edges of weight 1 each. In the modified graph, we can use BFS to find the shortest path. How is this approach $O(V+E)$? In worst case, all edges are of weight 2 and we need to do $O(E)$ operations to split all edges, so the time complexity becomes $O(E) + O(V+E)$ which is $O(V+E)$.

Question 4: Given a directed acyclic weighted graph, how to find the shortest path from a source s to a destination t in $O(V+E)$ time?

See: [Shortest Path in Directed Acyclic Graph](#)

More Questions See following links for more questions.

<http://algs4.cs.princeton.edu/44sp/>

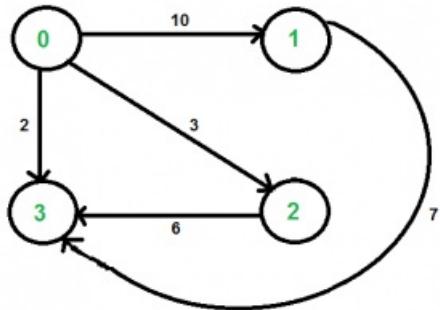
<http://geeksquiz.com/graph-shortest-paths/>

Shortest path with exactly k edges in a directed and weighted graph

Given a directed and two vertices u and v in it, find shortest path from u to v with exactly k edges on the path.

The graph is given as [adjacency matrix representation](#) where value of $\text{graph}[i][j]$ indicates the weight of an edge from vertex i to vertex j and a value INF(infinite) indicates no edge from i to j.

For example consider the following graph. Let source u be vertex 0, destination v be 3 and k be 2. There are two walks of length 2, the walks are $\{0, 2, 3\}$ and $\{0, 1, 3\}$. The shortest among the two is $\{0, 2, 3\}$ and weight of path is $3+6=9$.



The idea is to browse through all paths of length k from u to v using the approach discussed in the [previous post](#) and return weight of the shortest path. A **simple solution** is to start from u, go to all adjacent vertices and recur for adjacent vertices with k as k-1, source as adjacent vertex and destination as v. Following are C++ and Java implementations of this simple solution.

C++

```
// C++ program to find shortest path with exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A naive recursive function to count walks from u to v with k edges
int shortestPath(int graph[][V], int u, int v, int k)
{
    // Base cases
    if (k == 0 && u == v)          return 0;
    if (k == 1 && graph[u][v] != INF) return graph[u][v];
    if (k <= 0)                    return INF;

    // Initialize result
    int res = INF;

    // Go to all adjacents of u and recur
    for (int i = 0; i < V; i++)
    {
        if (graph[u][i] != INF && u != i && v != i)
        {
            int rec_res = shortestPath(graph, i, v, k-1);
            if (rec_res != INF)
                res = min(res, graph[u][i] + rec_res);
        }
    }
    return res;
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 10, 3, 2},
                        {INF, 0, INF, 7},
                        {INF, INF, 0, 6},
                        {INF, INF, INF, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << "Weight of the shortest path is " <<
           shortestPath(graph, u, v, k);
    return 0;
}
```

Java

```
// Dynamic Programming based Java program to find shortest path
// with exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // Define number of vertices in the graph and infinite value
    static final int V = 4;
    static final int INF = Integer.MAX_VALUE;

    // A naive recursive function to count walks from u to v
    // with k edges
    int shortestPath(int graph[][], int u, int v, int k)
    {
        // Base cases
        if (k == 0 && u == v) return 0;
        if (k == 1 && graph[u][v] != INF) return graph[u][v];
        if (k <= 0) return INF;

        // Initialize result
        int res = INF;

        // Go to all adjacents of u and recur
        for (int i = 0; i < V; i++)
        {
            if (graph[u][i] != INF && u != i && v != i)
            {
                int rec_res = shortestPath(graph, i, v, k-1);
                if (rec_res != INF)
                    res = Math.min(res, graph[u][i] + rec_res);
            }
        }
        return res;
    }

    public static void main (String[] args)
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] = new int[][]{{0, 10, 3, 2},
                                    {INF, 0, INF, 7},
                                    {INF, INF, 0, 6},
                                    {INF, INF, INF, 0}
                                };
        ShortestPath t = new ShortestPath();
        int u = 0, v = 3, k = 2;
        System.out.println("Weight of the shortest path is "+
                           t.shortestPath(graph, u, v, k));
    }
}
```

Weight of the shortest path is 9

The worst case time complexity of the above function is $O(V^k)$ where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly V children.

We can optimize the above solution using [Dynamic Programming](#). The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other [Dynamic Programming problems](#), we fill the 3D table in bottom up manner.

C++

```
// Dynamic Programming based C++ program to find shortest path with
// exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A Dynamic programming based function to find the shortest path from
```

```

// u to v with exactly k edges.
int shortestPath(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value sp[i][j][e] will store
    // weight of the shortest path from i to j with exactly k edges
    int sp[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++) // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                sp[i][j][e] = INF;

                // from base cases
                if (e == 0 && i == j)
                    sp[i][j][e] = 0;
                if (e == 1 && graph[i][j] != INF)
                    sp[i][j][e] = graph[i][j];

                // go to adjacent only when number of edges is more than 1
                if (e > 1)
                {
                    for (int a = 0; a < V; a++)
                    {
                        // There should be an edge from i to a and a
                        // should not be same as either i or j
                        if (graph[i][a] != INF && i != a &&
                            j != a && sp[a][j][e-1] != INF)
                            sp[i][j][e] = min(sp[i][j][e], graph[i][a] +
                                sp[a][j][e-1]);
                    }
                }
            }
        }
    }
    return sp[u][v][k];
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 10, 3, 2},
                        {INF, 0, INF, 7},
                        {INF, INF, 0, 6},
                        {INF, INF, INF, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << shortestPath(graph, u, v, k);
    return 0;
}

```

Java

```

// Dynamic Programming based Java program to find shortest path with
// exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // Define number of vertices in the graph and infinite value
    static final int V = 4;
    static final int INF = Integer.MAX_VALUE;

    // A Dynamic programming based function to find the shortest path
    // from u to v with exactly k edges.
    int shortestPath(int graph[][], int u, int v, int k)
    {
        // Table to be filled up using DP. The value sp[i][j][e] will
        // store weight of the shortest path from i to j with exactly
        // k edges
        int sp[][][] = new int[V][V][k+1];

        // Loop for number of edges from 0 to k

```

```

for (int e = 0; e <= k; e++)
{
    for (int i = 0; i < V; i++) // for source
    {
        for (int j = 0; j < V; j++) // for destination
        {
            // initialize value
            sp[i][j][e] = INF;

            // from base cases
            if (e == 0 && i == j)
                sp[i][j][e] = 0;
            if (e == 1 && graph[i][j] != INF)
                sp[i][j][e] = graph[i][j];

            // go to adjacent only when number of edges is
            // more than 1
            if (e > 1)
            {
                for (int a = 0; a < V; a++)
                {
                    // There should be an edge from i to a and
                    // a should not be same as either i or j
                    if (graph[i][a] != INF && i != a &&
                        j != a && sp[a][j][e-1] != INF)
                        sp[i][j][e] = Math.min(sp[i][j][e],
                                              graph[i][a] + sp[a][j][e-1]);
                }
            }
        }
    }
}
return sp[u][v][k];
}

public static void main (String[] args)
{
    /* Let us create the graph shown in above diagram*/
    int graph[][] = new int[][]{ {0, 10, 3, 2},
                                {INF, 0, INF, 7},
                                {INF, INF, 0, 6},
                                {INF, INF, INF, 0}
                            };
    ShortestPath t = new ShortestPath();
    int u = 0, v = 3, k = 2;
    System.out.println("Weight of the shortest path is "+
                       t.shortestPath(graph, u, v, k));
}
}
//This code is contributed by Aakash Hasija

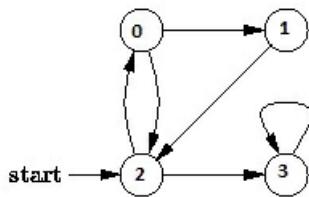
```

Weight of the shortest path is 9

Time complexity of the above DP based solution is $O(V^3K)$ which is much better than the naive solution.

Find if there is a path between two vertices in a directed graph

Given a Directed Graph and two vertices in it, check whether there is a path from the first given vertex to second. For example, in the following graph, there is a path from vertex 1 to 3. As another example, there is no path from 3 to 0.



We can either use [Breadth First Search \(BFS\)](#) or [Depth First Search \(DFS\)](#) to find path between two vertices. Take the first vertex as source in BFS (or DFS), follow the standard BFS (or DFS). If we see the second vertex in our traversal, then return true. Else return false.

Following are C++ and Java codes that use BFS for finding reachability of second vertex from first vertex.

C++

```
// C++ program to check if there is exist a path between two vertices
// of a graph.
#include<iostream>
#include <list>
using namespace std;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    bool isReachable(int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

// A BFS based function to check whether d is reachable from s.
bool Graph::isReachable(int s, int d)
{
    // Base case
    if (s == d)
        return true;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // it will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        int u = queue.front();
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex u
        // If an adjacent vertex v is not visited, enqueue it
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (!visited[*i])
                queue.push_back(*i);
    }

    // Check if destination is reached
    if (visited[d])
        return true;
    else
        return false;
}
```

```

s = queue.front();
queue.pop_front();

// Get all adjacent vertices of the dequeued vertex s
// If a adjacent has not been visited, then mark it visited
// and enqueue it
for (i = adj[s].begin(); i != adj[s].end(); ++i)
{
    // If this adjacent node is the destination node, then
    // return true
    if (*i == d)
        return true;

    // Else, continue to do BFS
    if (!visited[*i])
    {
        visited[*i] = true;
        queue.push_back(*i);
    }
}

// If BFS is complete without visiting d
return false;
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    int u = 1, v = 3;
    if(g.isReachable(u, v))
        cout<< "\n There is a path from " << u << " to " << v;
    else
        cout<< "\n There is no path from " << u << " to " << v;

    u = 3, v = 1;
    if(g.isReachable(u, v))
        cout<< "\n There is a path from " << u << " to " << v;
    else
        cout<< "\n There is no path from " << u << " to " << v;

    return 0;
}

```

Java

```

// Java program to check if there is exist a path between two vertices
// of a graph.
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;      // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w) { adj[v].add(w); }
}

```

```

//prints BFS traversal from a given source s
Boolean isReachable(int s, int d)
{
    LinkedList<Integer>temp;

    // Mark all the vertices as not visited(By default set
    // as false)
    boolean visited[] = new boolean[V];

    // Create a queue for BFS
    LinkedList<Integer> queue = new LinkedList<Integer>();

    // Mark the current node as visited and enqueue it
    visited[s]=true;
    queue.add(s);

    // 'i' will be used to get all adjacent vertices of a vertex
    Iterator<Integer> i;
    while (queue.size()!=0)
    {
        // Dequeue a vertex from queue and print it
        s = queue.poll();

        int n;
        i = adj[s].listIterator();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it
        // visited and enqueue it
        while (i.hasNext())
        {
            n = i.next();

            // If this adjacent node is the destination node,
            // then return true
            if (n==d)
                return true;

            // Else, continue to do BFS
            if (!visited[n])
            {
                visited[n] = true;
                queue.add(n);
            }
        }
    }

    // If BFS is complete without visited d
    return false;
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    int u = 1;
    int v = 3;
    if (g.isReachable(u, v))
        System.out.println("There is a path from " + u +" to " + v);
    else
        System.out.println("There is no path from " + u +" to " + v);;

    u = 3;
    v = 1;
    if (g.isReachable(u, v))
        System.out.println("There is a path from " + u +" to " + v);
    else
        System.out.println("There is no path from " + u +" to " + v);;

}
// This code is contributed by Aakash Hasija

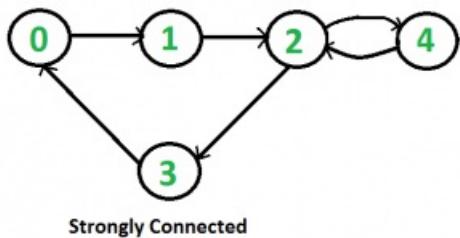
```

There is a path from 1 to 3
There is no path from 3 to 1

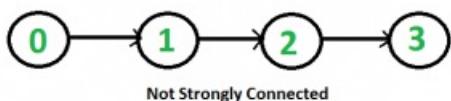
As an exercise, try an extended version of the problem where the complete path between two vertices is also needed.

Connectivity in a directed graph

Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pair of vertices. For example, following is a strongly connected graph.



It is easy for **undirected graph**, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach wont work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.



How to do for directed graph?

A simple idea is to use a all pair shortest path algorithm like [Floyd Warshall](#) or find [Transitive Closure](#) of graph. Time complexity of this method would be $O(v^3)$.

We can also **do DFS V times** starting from every vertex. If any DFS, doesnt visit all vertices, then graph is not strongly connected. This algorithm takes $O(V^*(V+E))$ time which can be same as transitive closure for a dense graph.

A better idea can be [Strongly Connected Components \(SCC\) algorithm](#). We can find all SCCs in $O(V+E)$ time. If number of SCCs is one, then graph is strongly connected. The algorithm for SCC does extra work as it finds all SCCs.

Following is **Kosaraju's DFS based simple algorithm that does two DFS traversals** of graph:

1) Initialize all vertices as not visited.

2) Do a DFS traversal of graph starting from any arbitrary vertex v. If DFS traversal doesnt visit all vertices, then return false.

3) Reverse all arcs (or find transpose or reverse of graph)

4) Mark all vertices as not-visited in reversed graph.

5) Do a DFS traversal of reversed graph starting from same vertex v (Same as step 2). If DFS traversal doesnt visit all vertices, then return false. Otherwise return true.

The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph).

Following is C++ implementation of above algorithm.

C++

```
// C++ program to check if a given directed graph is strongly
// connected or not
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // An array of adjacency lists

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);

public:
    // Constructor and Destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
```

```

~Graph() { delete [] adj; }

// Method to add an edge
void addEdge(int v, int w);

// The main function that returns true if the graph is strongly
// connected, otherwise false
bool isSC();

// Function that returns reverse (or transpose) of this graph
Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

// The main function that returns true if graph is strongly connected
bool Graph::isSC()
{
    // Step 1: Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from first vertex.
    // Starting Vertex must be same starting point of first DFS
    gr.DFSUtil(0, visited);

    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

```

```

}

// Driver program to test above functions
int main()
{
    // Create graphs given in the above diagrams
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    g1.isSC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.isSC()? cout << "Yes\n" : cout << "No\n";

    return 0;
}

```

Java

```

// Java program to check if a given directed graph is strongly
// connected or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;      // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w) { adj[v].add(w); }

    // A recursive function to print DFS starting from v
    void DFSUtil(int v,Boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;

        int n;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].iterator();
        while (i.hasNext())
        {
            n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // Function that returns transpose of this graph
    Graph getTranspose()
    {
        Graph g = new Graph(V);
        for (int v = 0; v < V; v++)
        {
            // Recur for all the vertices adjacent to this vertex
            Iterator<Integer> i = adj[v].listIterator();
            while (i.hasNext())
                g.adj[i.next()].add(v);
        }
    }
}

```

```

        return g;
    }

// The main function that returns true if graph is strongly
// connected
Boolean isSC()
{
    // Step 1: Mark all the vertices as not visited
    // (For first DFS)
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then
    // return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For
    // second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from
    // first vertex. Starting Vertex must be same starting
    // point of first DFS
    gr.DFSUtil(0, visited);

    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

public static void main(String args[])
{
    // Create graphs given in the above diagrams
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    if (g1.isSC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    if (g2.isSC())
        System.out.println("Yes");
    else
        System.out.println("No");
}
// This code is contributed by Aakash Hasija

```

Yes
No

Time Complexity: Time complexity of above implementation is same as [Depth First Search](#) which is $O(V+E)$ if the graph is represented using adjacency matrix representation.

Exercise:

Can we use BFS instead of DFS in above algorithm?

References:

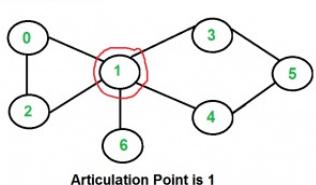
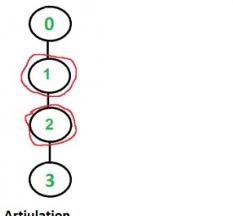
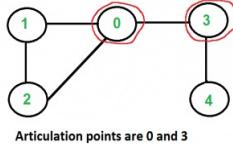
<http://www.ieor.berkeley.edu/~hochbaum/files/ieor266-2012.pdf>

Articulation Points (or Cut Vertices) in a Graph

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

Following are some example graphs with articulation points encircled with red color.



How to find all articulation points in a given graph?

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

1) For every vertex v , do following

..a) Remove v from graph

..b) See if the graph remains connected (We can either use BFS or DFS)

..c) Add v back to the graph

Time complexity of above method is $O(V^*(V+E))$ for a graph represented using adjacency list. Can we do better?

A $O(V+E)$ algorithm to find all Articulation Points (APs)

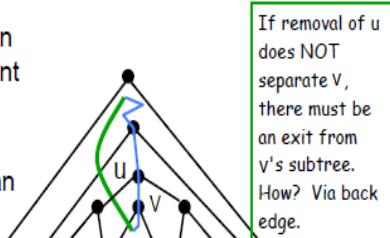
The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v , if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1) u is root of DFS tree and it has at least two children.

2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u .

Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point. (Source [Ref2](#))

- Root node is an articulation point iff it has more than one child
- Leaf is never an articulation point
- non-leaf, non-root node u is an articulation point



If removal of u does NOT separate v , there must be an exit from v 's subtree. How? Via back edge.

no non-tree edge goes above u from a sub-tree below some child of u

We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u . Among the above mentioned two cases, the first case is simple to detect. For every vertex, count

children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Following are C++ and Java implementations of Tarjans algorithm for finding articulation points.

C++

```
// A C++ program to find articulation points in an undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void APUtil(int v, bool visited[], int disc[], int low[],
                int parent[], bool ap[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void AP();    // prints articulation points
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);    // Note: the graph is undirected
}

// A recursive function that finds articulation points using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps track of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Stores articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
                   int low[], int parent[], bool ap[])
{
    // A static variable is used for simplicity, we can avoid use of static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;    // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);
        }
    }
}
```

```

// Check if the subtree rooted with v has a connection to
// one of the ancestors of u
low[u] = min(low[u], low[v]);

// u is an articulation point in following cases

// (1) u is root of DFS tree and has two or more children.
if (parent[u] == NIL && children > 1)
    ap[u] = true;

// (2) If u is not root and low value of one of its child is more
// than discovery value of u.
if (parent[u] != NIL && low[v] >= disc[u])
    ap[u] = true;
}

// Update low value of u for parent function calls.
else if (v != parent[u])
    low[u] = min(low[u], disc[v]);
}

// The function to do DFS traversal. It uses recursive function APUtil()
void Graph::AP()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    bool *ap = new bool[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point) arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation points
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
            cout << i << " ";
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nArticulation points in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.AP();

    cout << "\nArticulation points in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.AP();

    cout << "\nArticulation points in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
}

```

```

g3.addEdge(3, 5);
g3.addEdge(4, 5);
g3.AP();

return 0;
}

```

Java

```

// A Java program to find articulation points in an undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);    // Add w to v's list.
        adj[w].add(v);   //Add v to w's list
    }

    // A recursive function that finds articulation points using DFS
    // u --> The vertex to be visited next
    // visited[] --> keeps track of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    // ap[] --> Stores articulation points
    void APUtil(int u, boolean visited[], int disc[],
                int low[], int parent[], boolean ap[])
    {

        // Count of children in DFS Tree
        int children = 0;

        // Mark the current node as visited
        visited[u] = true;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;

        // Go through all vertices adjacent to this
        Iterator<Integer> i = adj[u].iterator();
        while (i.hasNext())
        {
            int v = i.next();  // v is current adjacent of u

            // If v is not visited yet, then make it a child of u
            // in DFS tree and recur for it
            if (!visited[v])
            {
                children++;
                parent[v] = u;
                APUtil(v, visited, disc, low, parent, ap);

                // Check if the subtree rooted with v has a connection to
                // one of the ancestors of u
                low[u] = Math.min(low[u], low[v]);
            }
        }

        // u is an articulation point in following cases
        // (i) u is root and has two or more children.
        // (ii) u is not root and has at least two children.
        // (iii) u is root and has only one child, and that child
        //        has at least two children.
        if (children > 1 || (children == 1 && parent[u] == NIL))
            ap[u] = true;
    }
}

```

```

// (1) u is root of DFS tree and has two or more children.
if (parent[u] == NIL && children > 1)
    ap[u] = true;

// (2) If u is not root and low value of one of its children
// is more than discovery value of u.
if (parent[u] != NIL && low[v] >= disc[u])
    ap[u] = true;
}

// Update low value of u for parent function calls.
else if (v != parent[u])
    low[u] = Math.min(low[u], disc[v]);
}

// The function to do DFS traversal. It uses recursive function APUtil()
void AP()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
    boolean ap[] = new boolean[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation
    // points in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
            System.out.print(i+" ");
}

// Driver method
public static void main(String args[])
{
    // Create graphs given in above diagrams
    System.out.println("Articulation points in first graph ");
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.AP();
    System.out.println();

    System.out.println("Articulation points in Second graph");
    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.AP();
    System.out.println();

    System.out.println("Articulation points in Third graph ");
    Graph g3 = new Graph(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.AP();
}
}

```

```
}
```

```
// This code is contributed by Aakash Hasija
```

```
Articulation points in first graph
0 3
Articulation points in second graph
1 2
Articulation points in third graph
1
```

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

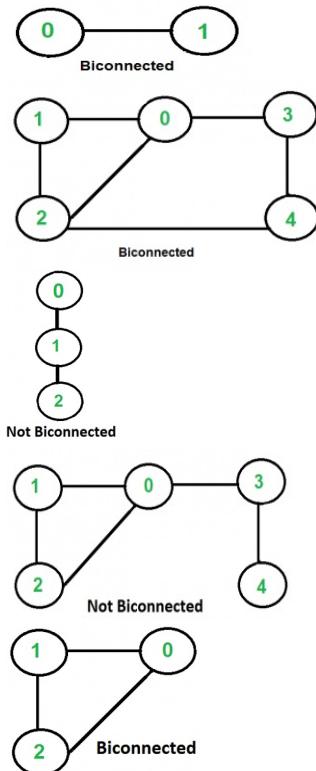
<https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>
<http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8>
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm

Biconnected graph

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices.

By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.

Following are some examples.



See [this](#) for more examples.

How to find if a given graph is Biconnected or not?

A connected graph is Biconnected if it is connected and doesn't have any [Articulation Point](#). We mainly need to check two things in a graph.

- 1) The graph is connected.
- 2) There is not articulation point in graph.

We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don't find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected.

Following is C++ implementation of above approach.

C++

```
// A C++ program to find if a given undirected graph is
// biconnected
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
```

```

list<int> *adj;      // A dynamic array of adjacency lists
bool isBCUtil(int v, bool visited[], int disc[], int low[],
              int parent[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isBC();      // returns true if graph is Biconnected
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that returns true if there is an articulation
// point in given graph, otherwise returns false.
// This function is almost same as isAPUtil() here ( http://goo.gl/Me9Fw )
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
bool Graph::isBCUtil(int u, bool visited[], int disc[], int low[], int parent[])
{
    // A static variable is used for simplicity, we can avoid use of static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;

            // check if subgraph rooted with v has an articulation point
            if (isBCUtil(v, visited, disc, low, parent))
                return true;

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u] = min(low[u], low[v]);
        }

        // u is an articulation point in following cases

        // (1) u is root of DFS tree and has two or more children.
        if (parent[u] == NIL && children > 1)
            return true;

        // (2) If u is not root and low value of one of its child is
        // more than discovery value of u.
        if (parent[u] != NIL && low[v] >= disc[u])
            return true;
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = min(low[u], disc[v]);
}
return false;
}

```

```

}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
bool Graph::isBC()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find if there is an articulation
    // point in given graph. We do DFS traversal starring from vertex 0
    if (isBCUtil(0, visited, disc, low, parent) == true)
        return false;

    // Now check whether the given graph is connected or not. An undirected
    // graph is connected if all vertices are reachable from any starting
    // point (we have taken 0 as starting point)
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    Graph g1(2);
    g1.addEdge(0, 1);
    g1.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(2, 4);
    g2.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g3(3);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g4(5);
    g4.addEdge(1, 0);
    g4.addEdge(0, 2);
    g4.addEdge(2, 1);
    g4.addEdge(0, 3);
    g4.addEdge(3, 4);
    g4.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g5(3);
    g5.addEdge(0, 1);
    g5.addEdge(1, 2);
    g5.addEdge(2, 0);
    g5.isBC()? cout << "Yes\n" : cout << "No\n";

    return 0;
}

```

Java

```

// A Java program to find if a given undirected graph is
// biconnected
import java.io.*;
import java.util.*;

```

```

import java.util.LinkedList;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); //Note that the graph is undirected.
        adj[w].add(v);
    }

    // A recursive function that returns true if there is an articulation
    // point in given graph, otherwise returns false.
    // This function is almost same as isAPUtil() @ http://goo.gl/Me9Fw
    // u --> The vertex to be visited next
    // visited[] --> keeps tract of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    boolean isBCUtil(int u, boolean visited[], int disc[], int low[],
                     int parent[])
    {

        // Count of children in DFS Tree
        int children = 0;

        // Mark the current node as visited
        visited[u] = true;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;

        // Go through all vertices adjacent to this
        Iterator<Integer> i = adj[u].iterator();
        while (i.hasNext())
        {
            int v = i.next(); // v is current adjacent of u

            // If v is not visited yet, then make it a child of u
            // in DFS tree and recur for it
            if (!visited[v])
            {
                children++;
                parent[v] = u;

                // check if subgraph rooted with v has an articulation point
                if (isBCUtil(v, visited, disc, low, parent))
                    return true;

                // Check if the subtree rooted with v has a connection to
                // one of the ancestors of u
                low[u] = Math.min(low[u], low[v]);
            }
        }

        // u is an articulation point in following cases

        // (1) u is root of DFS tree and has two or more children.
        if (parent[u] == NIL && children > 1)
            return true;

        // (2) If u is not root and low value of one of its
        // child is more than discovery value of u.
        if (parent[u] != NIL && low[v] >= disc[u])
            return true;
    }
}

```

```

    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = Math.min(low[u], disc[v]);
}
return false;
}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
boolean isBC()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find if there is an
    // articulation/ point in given graph. We do DFS traversal
    // starring from vertex 0
    if (isBCUtil(0, visited, disc, low, parent) == true)
        return false;

    // Now check whether the given graph is connected or not.
    // An undirected graph is connected if all vertices are
    // reachable from any starting point (we have taken 0 as
    // starting point)
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

// Driver method
public static void main(String args[])
{
    // Create graphs given in above diagrams
    Graph g1 =new Graph(2);
    g1.addEdge(0, 1);
    if (g1.isBC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g2 =new Graph(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(2, 4);
    if (g2.isBC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g3 = new Graph(3);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    if (g3.isBC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g4 = new Graph(5);
    g4.addEdge(1, 0);
    g4.addEdge(0, 2);
    g4.addEdge(2, 1);
    g4.addEdge(0, 3);
    g4.addEdge(3, 4);
}

```

```

    if (g4.isBC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g5= new Graph(3);
    g5.addEdge(0, 1);
    g5.addEdge(1, 2);
    g5.addEdge(2, 0);
    if (g5.isBC())
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

// This code is contributed by Aakash Hasija

```

Yes
Yes
No
No
Yes

Time Complexity: The above function is a simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

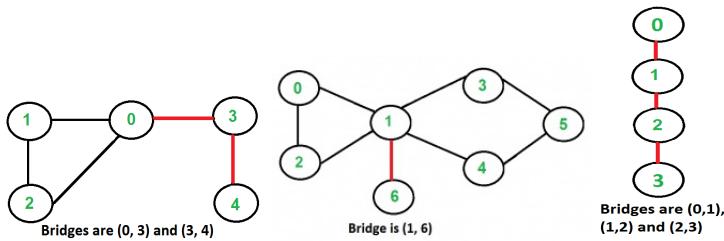
<http://www.cs.purdue.edu/homes/ayg/CS251/slides/chap9d.pdf>

Bridges in a graph

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of connected components.

Like [Articulation Points](#), bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.



How to find all bridges in a given graph?

A simple approach is to one by one remove all edges and see if removal of a edge causes disconnected graph. Following are steps of simple approach for connected graph.

- 1) For every edge (u, v) , do following
 - ..a) Remove (u, v) from graph
 - ..b) See if the graph remains connected (We can either use BFS or DFS)
 - ..c) Add (u, v) back to the graph.

Time complexity of above method is $O(E^*(V+E))$ for a graph represented using adjacency list. Can we do better?

A $O(V+E)$ algorithm to find all Bridges

The idea is similar to [O\(V+E\) algorithm for Articulation Points](#). We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v . As discussed in the [previous post](#), the value $\text{low}[v]$ indicates earliest visited vertex reachable from subtree rooted with v . *The condition for an edge (u, v) to be a bridge is, $\text{low}[v] > \text{disc}[u]$.*

Following are C++ and Java implementations of above approach.

C++

```
// A C++ program to find bridges in a given undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void bridgeUtil(int v, bool visited[], int disc[], int low[],
                    int parent[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    void bridge();    // prints all bridges
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
```

```

// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                      int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is below u in DFS tree, then u-v
            // is a bridge
            if (low[v] > disc[u])
                cout << u << " " << v << endl;
        }
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = min(low[u], disc[v]);
}
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            bridgeUtil(i, visited, disc, low, parent);
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nBridges in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();
}

```

```

cout << "\nBridges in second graph \n";
Graph g2(4);
g2.addEdge(0, 1);
g2.addEdge(1, 2);
g2.addEdge(2, 3);
g2.bridge();

cout << "\nBridges in third graph \n";
Graph g3(7);
g3.addEdge(0, 1);
g3.addEdge(1, 2);
g3.addEdge(2, 0);
g3.addEdge(1, 3);
g3.addEdge(1, 4);
g3.addEdge(1, 6);
g3.addEdge(3, 5);
g3.addEdge(4, 5);
g3.bridge();

return 0;
}

```

Java

```

// A Java program to find bridges in a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a undirected graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
        adj[w].add(v); //Add v to w's list
    }

    // A recursive function that finds and prints bridges
    // using DFS traversal
    // u --> The vertex to be visited next
    // visited[] --> keeps tract of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    void bridgeUtil(int u, boolean visited[], int disc[],
                    int low[], int parent[])
    {

        // Count of children in DFS Tree
        int children = 0;

        // Mark the current node as visited
        visited[u] = true;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;

        // Go through all vertices aadjacent to this
        Iterator<Integer> i = adj[u].iterator();
        while (i.hasNext())
        {

```

```

        int v = i.next(); // v is current adjacent of u

        // If v is not visited yet, then make it a child
        // of u in DFS tree and recur for it.
        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = Math.min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is below u in DFS tree, then u-v is
            // a bridge
            if (low[v] > disc[u])
                System.out.println(u+" "+v);
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u] = Math.min(low[u], disc[v]);
    }
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void bridge()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            bridgeUtil(i, visited, disc, low, parent);
}

public static void main(String args[])
{
    // Create graphs given in above diagrams
    System.out.println("Bridges in first graph ");
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();
    System.out.println();

    System.out.println("Bridges in Second graph");
    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();
    System.out.println();

    System.out.println("Bridges in Third graph ");
    Graph g3 = new Graph(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
}

```

```

g3.addEdge(1, 4);
g3.addEdge(1, 6);
g3.addEdge(3, 5);
g3.addEdge(4, 5);
g3.bridge();
}
// This code is contributed by Aakash Hasija

```

Bridges in first graph

3 4
0 3

Bridges in second graph

2 3
1 2
0 1

Bridges in third graph

1 6

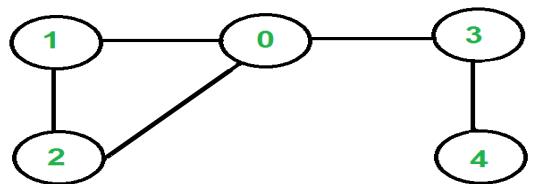
Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

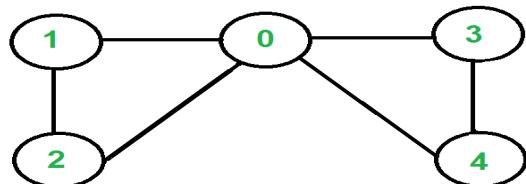
- <https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>
- <http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8>
- http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm
- <http://www.youtube.com/watch?v=bmyyxNyZKzI>

Eulerian path and circuit for undirected graph

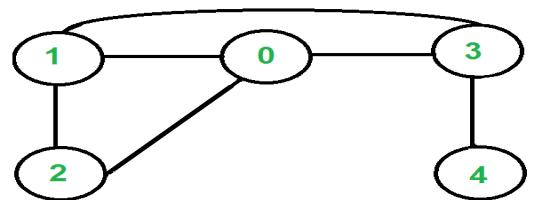
[Eulerian Path](#) is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

How to find whether a given graph is Eulerian or not?

The problem is same as following question. Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once.

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to [Hamiltonian Path](#) which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in $O(V+E)$ time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle

An undirected graph has Eulerian cycle if following two conditions are true.

- .a) All vertices with non-zero degree are connected. We dont care about vertices with zero degree because they dont belong to Eulerian Cycle or Path (we only consider all edges).
- .b) All vertices have even degree.

Eulerian Path

An undirected graph has Eulerian Path if following two conditions are true.

- .a) Same as condition (a) for Eulerian Cycle
- .b) If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

How does this work?

In Eulerian path, each time we visit a vertex v , we walk through two unvisited edges with one end point as v . Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

C++

```
// A C++ program to check if a given graph is Eulerian or not
#include<iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
```

```

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) {this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; } // To avoid memory leak

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Method to check if this graph is Eulerian or not
    int isEulerian();

    // Method to check if all non-zero degree vertices are connected
    bool isConnected();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool Graph::isConnected()
{
    // Mark all the vertices as not visited
    bool visited[V];
    int i;
    for (i = 0; i < V; i++)
        visited[i] = false;

    // Find a vertex with non-zero degree
    for (i = 0; i < V; i++)
        if (adj[i].size() != 0)
            break;

    // If there are no edges in the graph, return true
    if (i == V)
        return true;

    // Start DFS traversal from a vertex with non-zero degree
    DFSUtil(i, visited);

    // Check if all non-zero degree vertices are visited
    for (i = 0; i < V; i++)
        if (visited[i] == false && adj[i].size() > 0)
            return false;

    return true;
}

/* The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) */
int Graph::isEulerian()
{
    // Check if all non-zero degree vertices are connected
    if (isConnected() == false)
        return 0;

    // Count vertices with odd degree

```

```

int odd = 0;
for (int i = 0; i < V; i++)
    if (adj[i].size() & 1)
        odd++;

// If count is more than 2, then graph is not Eulerian
if (odd > 2)
    return 0;

// If odd count is 2, then semi-eulerian.
// If odd count is 0, then eulerian
// Note that odd count can never be 1 for undirected graph
return (odd)? 1 : 2;
}

// Function to run test cases
void test(Graph &g)
{
    int res = g.isEulerian();
    if (res == 0)
        cout << "graph is not Eulerian\n";
    else if (res == 1)
        cout << "graph has a Euler path\n";
    else
        cout << "graph has a Euler cycle\n";
}

// Driver program to test above function
int main()
{
    // Let us create and test graphs shown in above figures
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    test(g1);

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(4, 0);
    test(g2);

    Graph g3(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(1, 3);
    test(g3);

    // Let us create a graph with 3 vertices
    // connected in the form of cycle
    Graph g4(3);
    g4.addEdge(0, 1);
    g4.addEdge(1, 2);
    g4.addEdge(2, 0);
    test(g4);

    // Let us create a graph with all veritces
    // with zero degree
    Graph g5(3);
    test(g5);

    return 0;
}

```

Java

```

// A Java program to check if a given graph is Eulerian or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

```

```

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
        adj[w].add(v); //The graph is undirected
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited
        visited[v] = true;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // Method to check if all non-zero degree vertices are
    // connected. It mainly does DFS traversal starting from
    boolean isConnected()
    {
        // Mark all the vertices as not visited
        boolean visited[] = new boolean[V];
        int i;
        for (i = 0; i < V; i++)
            visited[i] = false;

        // Find a vertex with non-zero degree
        for (i = 0; i < V; i++)
            if (adj[i].size() != 0)
                break;

        // If there are no edges in the graph, return true
        if (i == V)
            return true;

        // Start DFS traversal from a vertex with non-zero degree
        DFSUtil(i, visited);

        // Check if all non-zero degree vertices are visited
        for (i = 0; i < V; i++)
            if (visited[i] == false && adj[i].size() > 0)
                return false;

        return true;
    }

    /* The function returns one of the following values
     * 0 --> If grpah is not Eulerian
     * 1 --> If graph has an Euler path (Semi-Eulerian)
     * 2 --> If graph has an Euler Circuit (Eulerian) */
    int isEulerian()
    {
        // Check if all non-zero degree vertices are connected
        if (isConnected() == false)
            return 0;

```

```

// Count vertices with odd degree
int odd = 0;
for (int i = 0; i < V; i++)
    if (adj[i].size()%2!=0)
        odd++;

// If count is more than 2, then graph is not Eulerian
if (odd > 2)
    return 0;

// If odd count is 2, then semi-eulerian.
// If odd count is 0, then eulerian
// Note that odd count can never be 1 for undirected graph
return (odd==2)? 1 : 2;
}

// Function to run test cases
void test()
{
    int res = isEulerian();
    if (res == 0)
        System.out.println("graph is not Eulerian");
    else if (res == 1)
        System.out.println("graph has a Euler path");
    else
        System.out.println("graph has a Euler cycle");
}

// Driver method
public static void main(String args[])
{
// Let us create and test graphs shown in above figures
Graph g1 = new Graph(5);
g1.addEdge(1, 0);
g1.addEdge(0, 2);
g1.addEdge(2, 1);
g1.addEdge(0, 3);
g1.addEdge(3, 4);
g1.test();

Graph g2 = new Graph(5);
g2.addEdge(1, 0);
g2.addEdge(0, 2);
g2.addEdge(2, 1);
g2.addEdge(0, 3);
g2.addEdge(3, 4);
g2.addEdge(4, 0);
g2.test();

Graph g3 = new Graph(5);
g3.addEdge(1, 0);
g3.addEdge(0, 2);
g3.addEdge(2, 1);
g3.addEdge(0, 3);
g3.addEdge(3, 4);
g3.addEdge(1, 3);
g3.test();

// Let us create a graph with 3 vertices
// connected in the form of cycle
Graph g4 = new Graph(3);
g4.addEdge(0, 1);
g4.addEdge(1, 2);
g4.addEdge(2, 0);
g4.test();

// Let us create a graph with all veritces
// with zero degree
Graph g5 = new Graph(3);
g5.test();
}
}

// This code is contributed by Aakash Hasija

```

graph has a Euler path
graph has a Euler cycle
graph is not Eulerian
graph has a Euler cycle
graph has a Euler cycle

Time Complexity: $O(V+E)$

We will soon be covering following topics on Eulerian Path and Circuit

- 1) Eulerian Path and Circuit for a Directed Graphs.
- 2) How to print a Eulerian Path or Circuit?

References:

http://en.wikipedia.org/wiki/Eulerian_path

Fleury's Algorithm for printing Eulerian Path or Circuit

Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

We strongly recommend to first read the following post on Euler Path and Circuit.

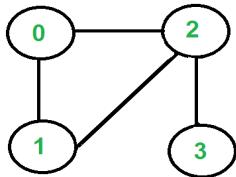
<http://www.geeksforgeeks.org/eulerian-path-and-circuit/>

In the above mentioned post, we discussed the problem of finding out whether a given graph is Eulerian or not. In this post, an algorithm to print Eulerian trail or circuit is discussed.

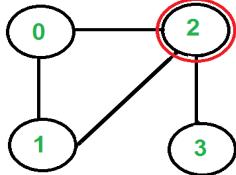
Following is Fleury's Algorithm for printing Eulerian trail or cycle (Source [Ref1](#)).

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, *always choose the non-bridge*.
4. Stop when you run out of edges.

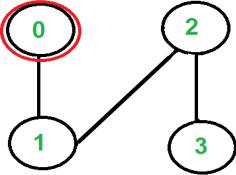
The idea is, **don't burn bridges** so that we can come back to a vertex and traverse remaining edges. For example let us consider the following graph.



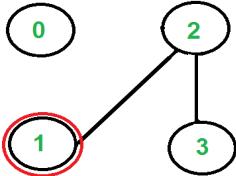
There are two vertices with odd degree, 2 and 3, we can start path from any of them. Let us start tour from vertex 2.



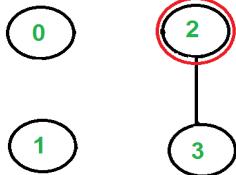
There are three edges going out from vertex 2, which one to pick? We don't pick the edge 2-3? because that is a bridge (we won't be able to come back to 3). We can pick any of the remaining two edges. Let us say we pick 2-0?. We remove this edge and move to vertex 0.



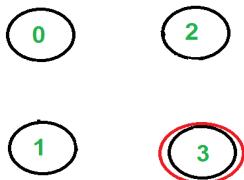
There is only one edge from vertex 0, so we pick it, remove it and move to vertex 1. Euler tour becomes 2-0 0-1?.



There is only one edge from vertex 1, so we pick it, remove it and move to vertex 2. Euler tour becomes 2-0 0-1 1-2?



Again there is only one edge from vertex 2, so we pick it, remove it and move to vertex 3. Euler tour becomes 2-0 0-1 1-2 2-3?



There are no more edges left, so we stop here. Final tour is 2-0 0-1 1-2 2-3?.

See [this](#) for and [this](#) fore more examples.

Following is C++ implementation of above algorithm. In the following code, it is assumed that the given graph has an Eulerian trail or Circuit. The main focus is to print an Eulerian trail or circuit. We can use [isEulerian\(\)](#) to first check whether there is an Eulerian Trail or Circuit in the given graph.

We first find the starting point which must be an odd vertex (if there are odd vertices) and store it in variable u. If there are zero odd vertices, we start from vertex 0. We call `printEulerUtil()` to print Euler tour starting with u. We traverse all adjacent vertices of u, if there is only one adjacent vertex, we immediately consider it. If there are more than one adjacent vertices, we consider an adjacent v only if edge u-v is not a bridge. How to find if a given is edge is bridge? We count number of vertices reachable from u. We remove edge u-v and again count number of reachable vertices from u. If number of reachable vertices are reduced, then edge u-v is a bridge. To count reachable vertices, we can either use BFS or DFS, we have used DFS in the above code. The function `DFSCount(u)` returns number of vertices reachable from u.

Once an edge is processed (included in Euler tour), we remove it from the graph. To remove the edge, we replace the vertex entry with -1 in adjacency list. Note that simply deleting the node may not work as the code is recursive and a parent call may be in middle of adjacency list.

```
// A C++ program print Eulerian Trail in a given Eulerian or Semi-Eulerian Graph
#include <iostream>
#include <string.h>
#include <algorithm>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; }

    // functions to add and remove edge
    void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
    void rmvEdge(int u, int v);

    // Methods to print Eulerian tour
    void printEulerTour();
    void printEulerUtil(int s);

    // This function returns count of vertices reachable from v. It does DFS
    int DFSCount(int v, bool visited[]);

    // Utility function to check if edge u-v is a valid next edge in
    // Eulerian trail or circuit
    bool isValidNextEdge(int u, int v);
};

/* The main function that print Eulerian Trail. It first finds an odd
   degree vertex (if there is any) and then calls printEulerUtil()
   to print the path */
void Graph::printEulerTour()
{
    // Find a vertex with odd degree
    int u = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
            { u = i; break; }

    // Print tour starting from oddv
    printEulerUtil(u);
    cout << endl;
}

// Print Euler tour starting from vertex u
void Graph::printEulerUtil(int u)
{
```

```

// Recur for all the vertices adjacent to this vertex
list<int>::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
    int v = *i;

    // If edge u-v is not removed and it's a valid next edge
    if (v != -1 && isValidNextEdge(u, v))
    {
        cout << u << "-" << v << " ";
        rmvEdge(u, v);
        printEulerUtil(v);
    }
}

// The function to check if edge u-v can be considered as next edge in
// Euler Tour
bool Graph::isValidNextEdge(int u, int v)
{
    // The edge u-v is valid in one of the following two cases:

    // 1) If v is the only adjacent vertex of u
    int count = 0; // To store count of adjacent vertices
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (*i != -1)
            count++;
    if (count == 1)
        return true;

    // 2) If there are multiple adjacents, then u-v is not a bridge
    // Do following steps to check if u-v is a bridge

    // 2.a) count of vertices reachable from u
    bool visited[V];
    memset(visited, false, V);
    int count1 = DFSCount(u, visited);

    // 2.b) Remove edge (u, v) and after removing the edge, count
    // vertices reachable from u
    rmvEdge(u, v);
    memset(visited, false, V);
    int count2 = DFSCount(u, visited);

    // 2.c) Add the edge back to the graph
    addEdge(u, v);

    // 2.d) If count1 is greater, then edge (u, v) is a bridge
    return (count1 > count2)? false: true;
}

// This function removes edge u-v from graph. It removes the edge by
// replacing adjacent vertex value with -1.
void Graph::rmvEdge(int u, int v)
{
    // Find v in adjacency list of u and replace it with -1
    list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);
    *iv = -1;

    // Find u in adjacency list of v and replace it with -1
    list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);
    *iu = -1;
}

// A DFS based function to count reachable vertices from v
int Graph::DFSCount(int v, bool visited[])
{
    // Mark the current node as visited
    visited[v] = true;
    int count = 1;

    // Recur for all vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (*i != -1 && !visited[*i])
            count += DFSCount(*i, visited);

    return count;
}

```

```

// Driver program to test above function
int main()
{
    // Let us first create and test graphs shown in above figure
    Graph g1(4);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.printEulerTour();

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 0);
    g2.printEulerTour();

    Graph g3(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(3, 2);
    g3.addEdge(3, 1);
    g3.addEdge(2, 4);
    g3.printEulerTour();

    return 0;
}

```

Output:

```

2-0  0-1  1-2  2-3
0-1  1-2  2-0
0-1  1-2  2-0  0-3  3-4  4-2  2-3  3-1

```

Note that the above code modifies given graph, we can create a copy of graph if we dont want the given graph to be modified.

Time Complexity: Time complexity of the above implementation is $O((V+E)^2)$. The function printEulerUtil() is like DFS and it calls isValidNextEdge() which also does DFS two times. Time complexity of DFS for adjacency list representation is $O(V+E)$. Therefore overall time complexity is $O((V+E)*(V+E))$ which can be written as $O(E^2)$ for a connected graph.

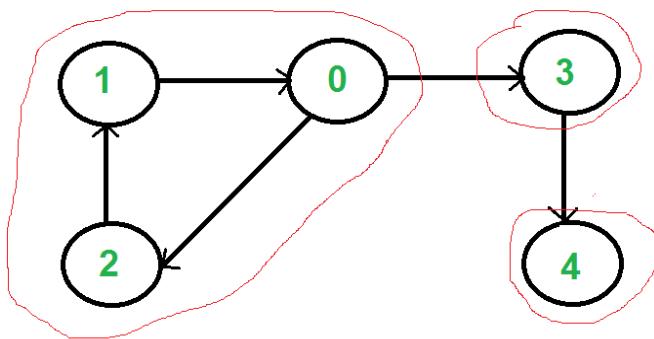
There are better algorithms to print Euler tour, we will soon be covering them as separate posts.

References:

<http://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter5-part2.pdf>
http://en.wikipedia.org/wiki/Eulerian_path#Fleury.27s_algorithm

Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.

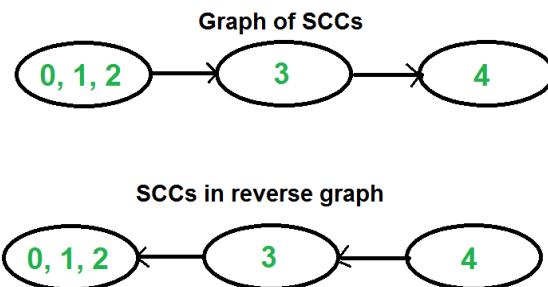


We can find all strongly connected components in $O(V+E)$ time using [Kosaraju's algorithm](#). Following is detailed Kosaraju algorithm.

- 1) Create an empty stack S and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.
- 2) Reverse directions of all arcs to obtain the transpose graph.
- 3) One by one pop a vertex from S while S is not empty. Let the popped vertex be v. Take v as source and do DFS (call [DFSUtil\(v\)](#)). The DFS starting from v prints strongly connected component of v.

How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC (See [this](#) for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4. In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC $\{0, 1, 2\}$ becomes sink and the SCC $\{4\}$ becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source. That is what we wanted to achieve and that is all needed to print SCCs one by one.



Following is C++ implementation of Kosaraju's algorithm.

C++

```
// C++ Implementation of Kosaraju's algorithm to print all SCCs
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
```

```

int V;      // No. of vertices
list<int> *adj;    // An array of adjacency lists

// Fills Stack with vertices (in increasing order of finishing
// times). The top element of stack has the maximum finishing
// time
void fillOrder(int v, bool visited[], stack<int> &Stack);

// A recursive function to print DFS starting from v
void DFSUtil(int v, bool visited[]);
public:
Graph(int V);
void addEdge(int v, int w);

// The main function that finds and prints strongly connected
// components
void printSCCs();

// Function that returns reverse (or transpose) of this graph
Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);

    // All vertices reachable from v are processed by now, push v
    Stack.push(v);
}

// The main function that finds and prints all strongly connected
// components
void Graph::printSCCs()
{

```

```

stack<int> Stack;

// Mark all the vertices as not visited (For first DFS)
bool *visited = new bool[V];
for(int i = 0; i < V; i++)
    visited[i] = false;

// Fill vertices in stack according to their finishing times
for(int i = 0; i < V; i++)
    if(visited[i] == false)
        fillOrder(i, visited, Stack);

// Create a reversed graph
Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)
for(int i = 0; i < V; i++)
    visited[i] = false;

// Now process all vertices in order defined by Stack
while (Stack.empty() == false)
{
    // Pop a vertex from stack
    int v = Stack.top();
    Stack.pop();

    // Print Strongly connected component of the popped vertex
    if (visited[v] == false)
    {
        gr.DFSUtil(v, visited);
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "Following are strongly connected components in "
          "given graph \n";
    g.printSCCs();

    return 0;
}

```

Java

```

// Java implementation of Kosaraju's algorithm to print all SCCs
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)  { adj[v].add(w); }

    // A recursive function to print DFS starting from v

```

```

void DFSUtil(int v,boolean visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    System.out.print(v + " ");

    int n;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i = adj[v].iterator();
    while (i.hasNext())
    {
        n = i.next();
        if (!visited[n])
            DFSUtil(n,visited);
    }
}

// Function that returns reverse (or transpose) of this graph
Graph getTranspose()
{
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while(i.hasNext())
            g.adj[i.next()].add(v);
    }
    return g;
}

void fillOrder(int v, boolean visited[], Stack stack)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i = adj[v].iterator();
    while (i.hasNext())
    {
        int n = i.next();
        if(!visited[n])
            fillOrder(n, visited, stack);
    }

    // All vertices reachable from v are processed by now,
    // push v to Stack
    stack.push(new Integer(v));
}

// The main function that finds and prints all strongly
// connected components
void printSCCs()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited (For first DFS)
    boolean visited[] = new boolean[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Fill vertices in stack according to their finishing
    // times
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            fillOrder(i, visited, stack);

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Now process all vertices in order defined by Stack
    while (stack.empty() == false)
    {
        // Pop a vertex from stack
        int v = (int)stack.pop();

```

```

// Print Strongly connected component of the popped vertex
if (visited[v] == false)
{
    gr.DFSUtil(v, visited);
    System.out.println();
}
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    System.out.println("Following are strongly connected components "+ 
                       "in given graph ");
    g.printSCCs();
}
}

// This code is contributed by Aakash Hasija

```

Following are strongly connected components in given graph
0 1 2
3
4

Time Complexity: The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes $O(V+E)$ for a graph represented using adjacency list. Reversing a graph also takes $O(V+E)$ time. For reversing the graph, we simply traverse all adjacency lists.

The above algorithm is asymptotically best algorithm, but there are other algorithms like [Tarfjans algorithm](#) and [path-based](#) which have same time complexity but find SCCs using single DFS. The Tarfjans algorithm is discussed in the following post.

[Tarfjans Algorithm to find Strongly Connected Components](#)

Applications:

SCC algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph.

In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

References:

http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

<https://www.youtube.com/watch?v=PZQ0Pdk15RA>

You may also like to see [Tarfjans Algorithm to find Strongly Connected Components](#).

Transitive closure of a graph

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j . The reach-ability matrix is called transitive closure of a graph.

The graph is given in the form of adjacency matrix say $\text{graph}[V][V]$ where $\text{graph}[i][j]$ is 1 if there is an edge from vertex i to vertex j or i is equal to j , otherwise $\text{graph}[i][j]$ is 0.

[Floyd Warshall Algorithm](#) can be used, we can calculate the distance matrix $\text{dist}[V][V]$ using [Floyd Warshall](#), if $\text{dist}[i][j]$ is infinite, then j is not reachable from i , otherwise j is reachable and value of $\text{dist}[i][j]$ will be less than V .

Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

1) Instead of integer resultant matrix ([dist\[V\]\[V\] in floyd warshall](#)), we can create a boolean reach-ability matrix $\text{reach}[V][V]$ (we save space). The value $\text{reach}[i][j]$ will be 1 if j is reachable from i , otherwise 0.

2) Instead of using arithmetic operations, we can use logical operations. For arithmetic operation $+$, logical and $&&$ is used, and for \min , logical or \parallel is used. (We save time by a constant factor. Time complexity is same though)

C++

```
// Program for transitive closure using Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

// A function to print the solution matrix
void printSolution(int reach[][V]);

// Prints transitive closure of graph[][] using Floyd Warshall algorithm
void transitiveClosure(int graph[][V])
{
    /* reach[][] will be the output matrix that will finally have the
       shortest distances between every pair of vertices */
    int reach[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            reach[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
       ---> Before start of a iteration, we have reachability values for
             all pairs of vertices such that the reachability values
             consider only the vertices in set {0, 1, 2, .. k-1} as
             intermediate vertices.
       ----> After the end of a iteration, vertex no. k is added to the
             set of intermediate vertices and the set becomes {0, 1, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on a path from i to j,
                // then make sure that the value of reach[i][j] is 1
                reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int reach[][V])
{
    printf ("Following matrix is transitive closure of the given graph\n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
            printf ("%d ", reach[i][j]);
    }
}
```

```

        printf("\n");
    }

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
     10
     (0)----->(3)
     |           /|\
     |           |
     5 |           | 1
     \|\/
     (1)----->(2)
     3           */
    int graph[V][V] = { {1, 1, 0, 1},
                        {0, 1, 1, 0},
                        {0, 0, 1, 1},
                        {0, 0, 0, 1}
                      };
}

// Print the solution
transitiveClosure(graph);
return 0;
}

```

Java

```

// Program for transitive closure using Floyd Warshall Algorithm
import java.util.*;
import java.lang.*;
import java.io.*;

class GraphClosure
{
    final static int V = 4; //Number of vertices in a graph

    // Prints transitive closure of graph[][] using Floyd
    // Warshall algorithm
    void transitiveClosure(int graph[][])
    {
        /* reach[][] will be the output matrix that will finally
           have the shortest distances between every pair of
           vertices */
        int reach[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph
           matrix. Or we can say the initial values of shortest
           distances are based on shortest paths considering
           no intermediate vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                reach[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
           vertices.
           ---> Before start of a iteration, we have reachability
               values for all pairs of vertices such that the
               reachability values consider only the vertices in
               set {0, 1, 2, .. k-1} as intermediate vertices.
           ----> After the end of a iteration, vertex no. k is
               added to the set of intermediate vertices and the
               set becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)
            {
                // Pick all vertices as destination for the
                // above picked source
                for (j = 0; j < V; j++)
                {
                    // If vertex k is on a path from i to j,
                    // then make sure that the value of reach[i][j] is 1
                    reach[i][j] = (reach[i][j]!=0) ||
                                  ((reach[i][k]!=0) && (reach[k][j]!=0))?1:0;
                }
            }
        }
    }
}

```

```

}

// Print the shortest distance matrix
printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int reach[][])
{
    System.out.println("Following matrix is transitive closure"+
                       " of the given graph");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
            System.out.print(reach[i][j]+" ");
        System.out.println();
    }
}

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
       |           /|\
      5 |           |
       |           | 1
      \|/           |
       (1)----->(2)
       3           */
}

/* Let us create the following weighted graph

       10
       (0)----->(3)
       |           /|\
      5 |           |
       |           | 1
      \|/           |
       (1)----->(2)
       3           */
int graph[][] = new int[][]{{1, 1, 0, 1},
                           {0, 1, 1, 0},
                           {0, 0, 1, 1},
                           {0, 0, 0, 1}};
};

// Print the solution
GraphClosure g = new GraphClosure();
g.transitiveClosure(graph);
}

// This code is contributed by Aakash Hasija
}

```

Following matrix is transitive closure of the given graph
 1 1 1 1
 0 1 1 1
 0 0 1 1
 0 0 0 1

Time Complexity: $O(V^3)$ where V is number of vertices in the given graph.

References:

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

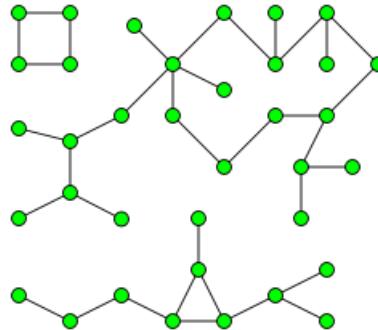
Find the number of islands

Given a boolean 2D matrix, find the number of islands.

This is an variation of the standard problem Counting number of connected components in a undirected graph.

Before we go to the problem, let us understand what is a connected component. A [connected component](#) of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.

For example, the graph shown below has three connected components.



A graph where all vertices are connected with each other, has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},  
{0, 1, 0, 0, 1},  
{1, 0, 0, 1, 1},  
{0, 0, 0, 0, 0},  
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbors. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursive call for 8 neighbors only. We keep track of the visited 1s so that they are not visited again.

C/C++

```
// Program to count islands in boolean 2D matrix  
#include <stdio.h>  
#include <string.h>  
#include <stdbool.h>  
  
#define ROW 5  
#define COL 5  
  
// A function to check if a given cell (row, col) can be included in DFS  
int isSafe(int M[][COL], int row, int col, bool visited[] [COL])  
{  
    // row number is in range, column number is in range and value is 1  
    // and not yet visited  
    return (row >= 0) && (row < ROW) &&  
        (col >= 0) && (col < COL) &&  
        (M[row][col] && !visited[row][col]);  
}  
  
// A utility function to do DFS for a 2D boolean matrix. It only considers  
// the 8 neighbours as adjacent vertices  
void DFS(int M[][] [COL], int row, int col, bool visited[] [COL])  
{  
    // These arrays are used to get row and column numbers of 8 neighbours  
    // of a given cell  
    static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};  
    static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};  
  
    // Mark this cell as visited  
    visited[row][col] = true;  
  
    // Recur for all connected neighbours  
    for (int k = 0; k < 8; ++k)  
    {  
        int r = row + rowNbr[k];  
        int c = col + colNbr[k];  
        if (isSafe(M, r, c, visited))  
            DFS(M, r, c, visited);  
    }  
}
```

```

        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
    }

// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[][])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW][COL];
    memset(visited, 0, sizeof(visited));

    // Initialize count as 0 and traverse through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j] && !visited[i][j]) // If a cell with value 1 is not
            {                                // visited yet, then new island found
                DFS(M, i, j, visited);      // Visit all cells in this island.
                ++count;                  // and increment island count
            }

    return count;
}

// Driver program to test above function
int main()
{
    int M[][][COL]= { {1, 1, 0, 0, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 0, 1, 1},
                      {0, 0, 0, 0, 0},
                      {1, 0, 1, 0, 1}
    };

    printf("Number of islands is: %d\n", countIslands(M));
    return 0;
}

```

Java

```

// Java program to count islands in boolean 2D matrix
import java.util.*;
import java.lang.*;
import java.io.*;

class Islands
{
    //No of rows and columns
    static final int ROW = 5, COL = 5;

    // A function to check if a given cell (row, col) can
    // be included in DFS
    boolean isSafe(int M[][], int row, int col,
                  boolean visited[][])
    {
        // row number is in range, column number is in range
        // and value is 1 and not yet visited
        return (row >= 0) && (row < ROW) &&
               (col >= 0) && (col < COL) &&
               (M[row][col]==1 && !visited[row][col]);
    }

    // A utility function to do DFS for a 2D boolean matrix.
    // It only considers the 8 neighbors as adjacent vertices
    void DFS(int M[][], int row, int col, boolean visited[][])
    {
        // These arrays are used to get row and column numbers
        // of 8 neighbors of a given cell
        int rowNbr[] = new int[] {-1, -1, -1, 0, 0, 1, 1, 1};
        int colNbr[] = new int[] {-1, 0, 1, -1, 1, -1, 0, 1};

        // Mark this cell as visited
        visited[row][col] = true;

        // Recur for all connected neighbours
        for (int k = 0; k < 8; ++k)

```

```

        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
    }

// The main function that returns count of islands in a given
// boolean 2D matrix
int countIslands(int M[][])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    boolean visited[][] = new boolean[ROW][COL];

    // Initialize count as 0 and traverse through the all cells
    // of given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j]==1 && !visited[i][j]) // If a cell with
            {                                // value 1 is not
                // visited yet, then new island found, Visit all
                // cells in this island and increment island count
                DFS(M, i, j, visited);
                ++count;
            }
    return count;
}

// Driver method
public static void main (String[] args) throws java.lang.Exception
{
    int M[][]= new int[][] {{1, 1, 0, 0, 0},
                           {0, 1, 0, 0, 1},
                           {1, 0, 0, 1, 1},
                           {0, 0, 0, 0, 0},
                           {1, 0, 1, 0, 1}
                           };
    Islands I = new Islands();
    System.out.println("Number of islands is: "+ I.countIslands(M));
}
} //Contributed by Aakash Hasija

```

Output:

Number of islands is: 5

Time complexity: O(ROW x COL)

Reference:

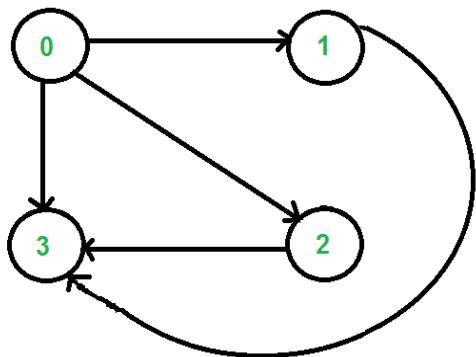
http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29

Count all possible walks from a source to a destination with exactly k edges

Given a directed graph and two vertices u and v in it, count all possible walks from u to v with exactly k edges on the walk.

The graph is given as [adjacency matrix representation](#) where value of $\text{graph}[i][j]$ as 1 indicates that there is an edge from vertex i to vertex j and a value 0 indicates no edge from i to j.

For example consider the following graph. Let source u be vertex 0, destination v be 3 and k be 2. The output should be 2 as there are two walk from 0 to 3 with exactly 2 edges. The walks are {0, 2, 3} and {0, 1, 3}



A **simple solution** is to start from u, go to all adjacent vertices and recur for adjacent vertices with k as k-1, source as adjacent vertex and destination as v. Following is C++ implementation of this simple solution.

C++

```
// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A naive recursive function to count walks from u to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    // Base cases
    if (k == 0 && u == v)      return 1;
    if (k == 1 && graph[u][v]) return 1;
    if (k <= 0)                return 0;

    // Initialize result
    int count = 0;

    // Go to all adjacents of u and recur
    for (int i = 0; i < V; i++)
        if (graph[u][i] == 1) // Check if is adjacent of u
            count += countwalks(graph, i, v, k-1);

    return count;
}
```

```
// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 1, 1, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}
```

Java

```
// Java program to count walks from u to v with exactly k edges
import java.util.*;
import java.lang.*;
```

```

import java.io.*;
class KPaths
{
    static final int V = 4; //Number of vertices

    // A naive recursive function to count walks from u
    // to v with k edges
    int countwalks(int graph[][], int u, int v, int k)
    {
        // Base cases
        if (k == 0 && u == v) return 1;
        if (k == 1 && graph[u][v] == 1) return 1;
        if (k <= 0) return 0;

        // Initialize result
        int count = 0;

        // Go to all adjacents of u and recur
        for (int i = 0; i < V; i++)
            if (graph[u][i] == 1) // Check if is adjacent of u
                count += countwalks(graph, i, v, k-1);

        return count;
    }

    // Driver method
    public static void main (String[] args) throws java.lang.Exception
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] =new int[][] { {0, 1, 1, 1},
                                    {0, 0, 0, 1},
                                    {0, 0, 0, 1},
                                    {0, 0, 0, 0}
                                };
        int u = 0, v = 3, k = 2;
        KPaths p = new KPaths();
        System.out.println(p.countwalks(graph, u, v, k));
    }
}

//Contributed by Aakash Hasija

```

2

The worst case time complexity of the above function is $O(V^k)$ where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly n children.

We can optimize the above solution using [Dynamic Programming](#). The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other [Dynamic Programming problems](#), we fill the 3D table in bottom up manner.

C++

```

// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A Dynamic programming based function to count walks from u
// to v with k edges
int countwalks(int graph[V][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value count[i][j][e] will
    // store count of possible walks from i to j with exactly k edges
    int count[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++) // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                count[i][j][e] = 0;

```

```

// from base cases
if (e == 0 && i == j)
    count[i][j][e] = 1;
if (e == 1 && graph[i][j])
    count[i][j][e] = 1;

// go to adjacent only when number of edges is more than 1
if (e > 1)
{
    for (int a = 0; a < V; a++) // adjacent of source i
        if (graph[i][a])
            count[i][j][e] += count[a][j][e-1];
}
}

return count[u][v][k];
}

// driver program to test above function
int main()
{
/* Let us create the graph shown in above diagram*/
int graph[V][V] = { {0, 1, 1, 1},
                    {0, 0, 0, 1},
                    {0, 0, 0, 1},
                    {0, 0, 0, 0}
                };
int u = 0, v = 3, k = 2;
cout << countwalks(graph, u, v, k);
return 0;
}

```

Java

```

// Java program to count walks from u to v with exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class KPaths
{
    static final int V = 4; //Number of vertices

    // A Dynamic programming based function to count walks from u
    // to v with k edges
    int countwalks(int graph[][], int u, int v, int k)
    {
        // Table to be filled up using DP. The value count[i][j][e]
        // will/ store count of possible walks from i to j with
        // exactly k edges
        int count[][][] = new int[V][V][k+1];

        // Loop for number of edges from 0 to k
        for (int e = 0; e <= k; e++)
        {
            for (int i = 0; i < V; i++) // for source
            {
                for (int j = 0; j < V; j++) // for destination
                {
                    // initialize value
                    count[i][j][e] = 0;

                    // from base cases
                    if (e == 0 && i == j)
                        count[i][j][e] = 1;
                    if (e == 1 && graph[i][j]!=0)
                        count[i][j][e] = 1;

                    // go to adjacent only when number of edges
                    // is more than 1
                    if (e > 1)
                    {
                        for (int a = 0; a < V; a++) // adjacent of i
                            if (graph[i][a]!=0)
                                count[i][j][e] += count[a][j][e-1];
                    }
                }
            }
        }
    }
}

```

```

        }
        return count[u][v][k];
    }

    // Driver method
    public static void main (String[] args) throws java.lang.Exception
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] =new int[][] { {0, 1, 1, 1},
                                    {0, 0, 0, 1},
                                    {0, 0, 0, 1},
                                    {0, 0, 0, 0}
                                };

        int u = 0, v = 3, k = 2;
        KPaths p = new KPaths();
        System.out.println(p.countwalks(graph, u, v, k));
    }
}//Contributed by Aakash Hasija

```

2

Time complexity of the above DP based solution is $O(V^3K)$ which is much better than the naive solution.

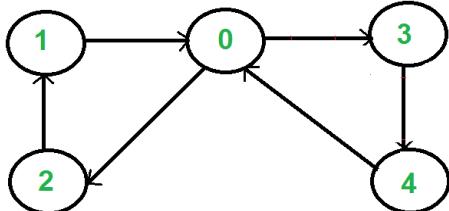
We can also use [Divide and Conquer](#) to solve the above problem in $O(V^3 \log k)$ time. The count of walks of length k from u to v is the $[u][v]$ th entry in $(\text{graph}[V][V])^k$. We can calculate power of by doing $O(\log k)$ multiplication by using the [divide and conquer technique to calculate power](#). A multiplication between two matrices of size $V \times V$ takes $O(V^3)$ time. Therefore overall time complexity of this method is $O(V^3 \log k)$.

Euler Circuit in a Directed Graph

[Eulerian Path](#) is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

A graph is said to be eulerian if it has eulerian cycle. We have discussed [eulerian circuit for an undirected graph](#). In this post, same is discussed for a directed graph.

For example, the following graph has eulerian cycle as {1, 0, 3, 4, 0, 2, 1}



How to check if a directed graph is eulerian?

A directed graph has an eulerian cycle if following conditions are true (Source: [Wiki](#))

- 1) All vertices with nonzero degree belong to a single [strongly connected component](#).
- 2) In degree and out degree of every vertex is same.

We can detect singly connected component using [Kosaraju's DFS based simple algorithm](#).

To compare in degree and out degree, we need to store in degree and out degree of every vertex. Out degree can be obtained by size of adjacency list. In degree can be stored by creating an array of size equal to number of vertices.

Following are C++ and Java implementations of above approach.

C++

```
// A C++ program to check if a given directed graph is Eulerian or not
#include<iostream>
#include <list>
#define CHARS 26
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    int *in;
public:
    // Constructor and destructor
    Graph(int V);
    ~Graph() { delete [] adj; delete [] in; }

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); (in[w])++; }

    // Method to check if this graph is Eulerian or not
    bool isEulerianCycle();

    // Method to check if all non-zero degree vertices are connected
    bool isSC();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);

    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
    in = new int[V];
    for (int i = 0; i < V; i++)
        in[i] = 0;
}

/* This function returns true if the directed graph has an eulerian
cycle, otherwise returns false */
bool Graph::isEulerianCycle()
{
```

```

// Check if all non-zero degree vertices are connected
if (isSC() == false)
    return false;

// Check if in degree and out degree of every vertex is same
for (int i = 0; i < V; i++)
    if (adj[i].size() != in[i])
        return false;

return true;
}

// A recursive function to do DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
// This function is needed in isSC()
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
            (g.in[v])++;
        }
    }
    return g;
}

// This function returns true if all non-zero degree vertices of
// graph are strongly connected (Please refer
// http://www.geeksforgeeks.org/connectivity-in-a-directed-graph/ )
bool Graph::isSC()
{
    // Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Find the first vertex with non-zero degree
    int n;
    for (n = 0; n < V; n++)
        if (adj[n].size() > 0)
            break;

    // Do DFS traversal starting from first non zero degree vertex.
    DFSUtil(n, visited);

    // If DFS traversal doesn't visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (adj[i].size() > 0 && visited[i] == false)
            return false;

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Do DFS for reversed graph starting from first vertex.
    // Starting Vertex must be same starting point of first DFS
    gr.DFSUtil(n, visited);

    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)

```

```

        if (adj[i].size() > 0 && visited[i] == false)
            return false;

    }

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    if (g.isEulerianCycle())
        cout << "Given directed graph is eulerian \n";
    else
        cout << "Given directed graph is NOT eulerian \n";
    return 0;
}

```

Java

```

// A Java program to check if a given directed graph is Eulerian or not

// A class that represents an undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
class Graph
{
    private int V;      // No. of vertices
    private LinkedList<Integer> adj[];//Adjacency List
    private int in[];   //maintaining in degree

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        in = new int[V];
        for (int i=0; i<v; ++i)
        {
            adj[i] = new LinkedList();
            in[i] = 0;
        }
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        in[w]++;
    }

    // A recursive function to print DFS starting from v
    void DFSUtil(int v,Boolean visited[])
    {
        // Mark the current node as visited
        visited[v] = true;

        int n;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i =adj[v].iterator();
        while (i.hasNext())
        {
            n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // Function that returns reverse (or transpose) of this graph

```

```

Graph getTranspose()
{
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            g.adj[i.next()].add(v);
            (g.in[v])++;
        }
    }
    return g;
}

// The main function that returns true if graph is strongly
// connected
Boolean isSC()
{
    // Step 1: Mark all the vertices as not visited (For
    // first DFS)
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from first vertex.
    // Staring Vertex must be same starting point of first DFS
    gr.DFSUtil(0, visited);

    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

/* This function returns true if the directed graph has an eulerian
cycle, otherwise returns false */
Boolean isEulerianCycle()
{
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;

    // Check if in degree and out degree of every vertex is same
    for (int i = 0; i < V; i++)
        if (adj[i].size() != in[i])
            return false;

    return true;
}

public static void main (String[] args) throws java.lang.Exception
{
    Graph g = new Graph(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    if (g.isEulerianCycle())

```

```
        System.out.println("Given directed graph is eulerian ");
    else
        System.out.println("Given directed graph is NOT eulerian ");
    }
}
//This code is contributed by Aakash Hasija
```

Given directed graph is eulerian

Time complexity of the above implementation is $O(V + E)$ as [Kosaraju's algorithm](#) takes $O(V + E)$ time. After running [Kosaraju's algorithm](#) we traverse all vertices and compare in degree with out degree which takes $O(V)$ time.

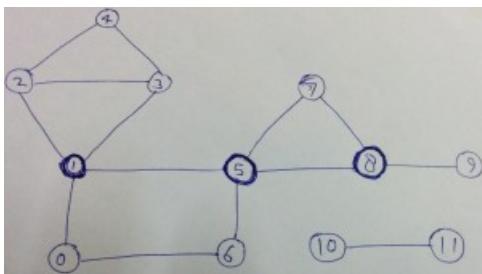
See following as an application of this.

[Find if the given array of strings can be chained to form a circle.](#)

Biconnected Components

A [biconnected component](#) is a maximal [biconnected subgraph](#).

[Biconnected Graph](#) is already discussed [here](#). In this article, we will see how to find [biconnected component](#) in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 42 34 31 23 12
- 89
- 85 78 57
- 60 56 15 01
- 1011

Algorithm is based on Disc and Low Values discussed in [Strongly Connected Components](#) Article.

Idea is to store visited edges in a stack while DFS on a graph and keep looking for [Articulation Points](#) (highlighted in above figure). As soon as an [Articulation Point](#) u is found, all edges visited while DFS from node u onwards will form one [biconnected component](#). When DFS completes for one [connected component](#), all edges present in stack will form a biconnected component.

If there is no [Articulation Point](#) in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

C++

```
// A C++ program to find biconnected components in a given undirected graph
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;
int count = 0;
class Edge
{
    public:
    int u;
    int v;
    Edge(int u, int v);
};
Edge::Edge(int u, int v)
{
    this->u = u;
    this->v = v;
}

// A class that represents an directed graph
class Graph
{
    int V;      // No. of vertices
    int E;      // No. of edges
    list<int> *adj;    // A dynamic array of adjacency lists

    // A Recursive DFS based function used by BCC()
    void BCCUtil(int u, int disc[], int low[],
                 list<Edge> *st, int parent[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void BCC();    // prints strongly connected components
};

Graph::Graph(int V)
{
    this->V = V;
    this->E = 0;
```

```

adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    E++;
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] --> earliest visited vertex (the vertex with minimum
//           discovery time) that can be reached from subtree
//           rooted with current vertex
// *st --> To store visited edges
void Graph::BCCUtil(int u, int disc[], int low[], list<Edge> *st,
                    int parent[])
{
    // A static variable is used for simplicity, we can avoid use
    // of static variable by passing a pointer.
    static int time = 0;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    int children = 0;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of 'u'

        // If v is not visited yet, then recur for it
        if (disc[v] == -1)
        {
            children++;
            parent[v] = u;
            //store the edge in stack
            st->push_back(Edge(u,v));
            BCCUtil(v, disc, low, st, parent);

            // Check if the subtree rooted with 'v' has a
            // connection to one of the ancestors of 'u'
            // Case 1 -- per Strongly Connected Components Article
            low[u] = min(low[u], low[v]);

            //If u is an articulation point,
            //pop all edges from stack till u -- v
            if( (disc[u] == 1 && children > 1) ||
                (disc[u] > 1 && low[v] >= disc[u]) )
            {
                while(st->back().u != u || st->back().v != v)
                {
                    cout << st->back().u << "--" << st->back().v << " ";
                    st->pop_back();
                }
                cout << st->back().u << "--" << st->back().v;
                st->pop_back();
                cout << endl;
            }
            count++;
        }
    }

    // Update low value of 'u' only if 'v' is still in stack
    // (i.e. it's a back edge, not cross edge).
    // Case 2 -- per Strongly Connected Components Article
    else if(v != parent[u] && disc[v] < low[u])
    {
        low[u] = min(low[u], disc[v]);
        st->push_back(Edge(u,v));
    }
}

// The function to do DFS traversal. It uses BCCUtil()
void Graph::BCC()
{
    int *disc = new int[V];
    int *low = new int[V];
}

```

```

int *parent = new int[V];
list<Edge> *st = new list<Edge>[E];

// Initialize disc and low, and parent arrays
for (int i = 0; i < V; i++)
{
    disc[i] = NIL;
    low[i] = NIL;
    parent[i] = NIL;
}

for (int i = 0; i < V; i++)
{
    if (disc[i] == NIL)
        BCCUtil(i, disc, low, st, parent);

    int j = 0;
    //If stack is not empty, pop all edges from stack
    while(st->size() > 0)
    {
        j = 1;
        cout << st->back().u << "--" << st->back().v << " ";
        st->pop_back();
    }
    if(j == 1)
    {
        cout << endl;
        count++;
    }
}
}

// Driver program to test above function
int main()
{
    Graph g(12);
    g.addEdge(0,1);g.addEdge(1,0);
    g.addEdge(1,2);g.addEdge(2,1);
    g.addEdge(1,3);g.addEdge(3,1);
    g.addEdge(2,3);g.addEdge(3,2);
    g.addEdge(2,4);g.addEdge(4,2);
    g.addEdge(3,4);g.addEdge(4,3);
    g.addEdge(1,5);g.addEdge(5,1);
    g.addEdge(0,6);g.addEdge(6,0);
    g.addEdge(5,6);g.addEdge(6,5);
    g.addEdge(5,7);g.addEdge(7,5);
    g.addEdge(5,8);g.addEdge(8,5);
    g.addEdge(7,8);g.addEdge(8,7);
    g.addEdge(8,9);g.addEdge(9,8);
    g.addEdge(10,11);g.addEdge(11,10);
    g.BCC();
    cout << "Above are " << count << " biconnected components in graph";
    return 0;
}

```

Java

```

// A Java program to find biconnected components in a given
// undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V, E; // No. of vertices & Edges respectively
    private LinkedList<Integer> adj[]; // Adjacency List

    // Count is number of biconnected components. time is
    // used to find discovery times
    static int count = 0, time = 0;

    class Edge
    {
        int u;
        int v;
        Edge(int u, int v)
        {
            this.u = u;

```

```

        this.v = v;
    }
};

//Constructor
Graph(int v)
{
    V = v;
    E = 0;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

//Function to add an edge into the graph
void addEdge(int v,int w)
{
    adj[v].add(w);
    E++;
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] --> earliest visited vertex (the vertex with minimum
//           discovery time) that can be reached from subtree
//           rooted with current vertex
// *st --> To store visited edges
void BCCUtil(int u, int disc[], int low[], LinkedList<Edge>st,
             int parent[])
{
    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    int children = 0;

    // Go through all vertices adjacent to this
    Iterator<Integer> it = adj[u].iterator();
    while (it.hasNext())
    {
        int v = it.next(); // v is current adjacent of 'u'

        // If v is not visited yet, then recur for it
        if (disc[v] == -1)
        {
            children++;
            parent[v] = u;

            // store the edge in stack
            st.add(new Edge(u,v));
            BCCUtil(v, disc, low, st, parent);

            // Check if the subtree rooted with 'v' has a
            // connection to one of the ancestors of 'u'
            // Case 1 -- per Strongly Connected Components Article
            if (low[u] > low[v])
                low[u] = low[v];

            // If u is an articulation point,
            // pop all edges from stack till u == v
            if ( (disc[u] == 1 && children > 1) ||
                 (disc[u] > 1 && low[v] >= disc[u]) )
            {
                while (st.getLast().u != u || st.getLast().v != v)
                {
                    System.out.print(st.getLast().u + "--" +
                                    st.getLast().v + " ");
                    st.removeLast();
                }
                System.out.println(st.getLast().u + "--" +
                                   st.getLast().v + " ");
                st.removeLast();

                count++;
            }
        }

        // Update low value of 'u' only if 'v' is still in stack
        // (i.e. it's a back edge, not cross edge).
        // Case 2 -- per Strongly Connected Components Article
    }
}

```

```

        else if (v != parent[u] && disc[v] < low[u])
        {
            if (low[u]>disc[v])
                low[u]=disc[v];
            st.add(new Edge(u,v));
        }
    }

// The function to do DFS traversal. It uses BCCUtil()
void BCC()
{
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
    LinkedList<Edge> st = new LinkedList<Edge>();

    // Initialize disc and low, and parent arrays
    for (int i = 0; i < V; i++)
    {
        disc[i] = -1;
        low[i] = -1;
        parent[i] = -1;
    }

    for (int i = 0; i < V; i++)
    {
        if (disc[i] == -1)
            BCCUtil(i, disc, low, st, parent);

        int j = 0;

        // If stack is not empty, pop all edges from stack
        while (st.size() > 0)
        {
            j = 1;
            System.out.print(st.getLast().u + "--" +
                            st.getLast().v + " ");
            st.removeLast();
        }
        if (j == 1)
        {
            System.out.println();
            count++;
        }
    }
}

public static void main(String args[])
{
    Graph g = new Graph(12);
    g.addEdge(0,1);
    g.addEdge(1,0);
    g.addEdge(1,2);
    g.addEdge(2,1);
    g.addEdge(1,3);
    g.addEdge(3,1);
    g.addEdge(2,3);
    g.addEdge(3,2);
    g.addEdge(2,4);
    g.addEdge(4,2);
    g.addEdge(3,4);
    g.addEdge(4,3);
    g.addEdge(1,5);
    g.addEdge(5,1);
    g.addEdge(0,6);
    g.addEdge(6,0);
    g.addEdge(5,6);
    g.addEdge(6,5);
    g.addEdge(5,7);
    g.addEdge(7,5);
    g.addEdge(5,8);
    g.addEdge(8,5);
    g.addEdge(7,8);
    g.addEdge(8,7);
    g.addEdge(8,9);
    g.addEdge(9,8);
    g.addEdge(10,11);
    g.addEdge(11,10);

    g.BCC();
}

```

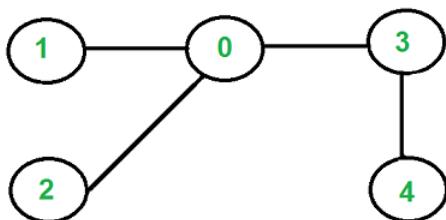
```
        System.out.println("Above are " + g.count +
                           " biconnected components in graph");
    }
}
// This code is contributed by Aakash Hasija
```

Output:

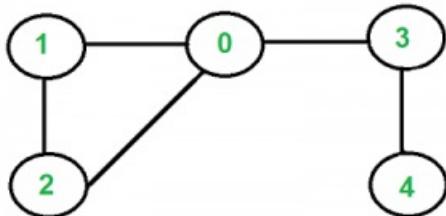
```
4--2 3--4 3--1 2--3 1--2
8--9
8--5 7--8 5--7
6--0 5--6 1--5 0--1
10--11
Above are 5 biconnected components in graph
```

Check if a given graph is tree or not

Write a function that returns true if a given undirected graph is tree and false otherwise. For example, the following graph is a tree.



But the following graph is not a tree.



An undirected graph is tree if it has following properties.

- 1) There is no cycle.
- 2) The graph is connected.

For an undirected graph we can either use [BFS](#) or [DFS](#) to detect above two properties.

How to detect cycle in an undirected graph?

We can either use BFS or DFS. For every visited vertex v, if there is an adjacent u such that u is already visited and u is not parent of v, then there is a cycle in graph. If we dont find such an adjacent for any vertex, we say that there is no cycle (See [Detect cycle in an undirected graph](#) for more details).

How to check for connectivity?

Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

C++

```
// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj; // Pointer to an array for adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isTree();    // returns true if graph is tree
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
    adj[w].push_back(v); // Add v to ws list.
}

// A recursive function that uses visited[] and parent to
// detect cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
```

```

{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for
        // that adjacent
        if (!visited[*i])
        {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of current
        // vertex, then there is a cycle.
        else if (*i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph is a tree, else false.
bool Graph::isTree()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // The call to isCyclicUtil serves multiple purposes.
    // It returns true if graph reachable from vertex 0
    // is cyclic. It also marks all vertices reachable
    // from 0.
    if (isCyclicUtil(0, visited, -1))
        return false;

    // If we find a vertex which is not reachable from 0
    // (not marked by isCyclicUtil()), then we return false
    for (int u = 0; u < V; u++)
        if (!visited[u])
            return false;

    return true;
}

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isTree() ? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.isTree() ? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    return 0;
}

```

Java

```

// A Java Program to check whether a graph is tree or not
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation

```

```

class Graph
{
    private int V;      // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    // A recursive function that uses visited[] and parent
    // to detect cycle in subgraph reachable from vertex v.
    Boolean isCyclicUtil(int v, Boolean visited[], int parent)
    {
        // Mark the current node as visited
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();

            // If an adjacent is not visited, then recur for
            // that adjacent
            if (!visited[i])
            {
                if (isCyclicUtil(i, visited, v))
                    return true;
            }

            // If an adjacent is visited and not parent of
            // current vertex, then there is a cycle.
            else if (i != parent)
                return true;
        }
        return false;
    }

    // Returns true if the graph is a tree, else false.
    Boolean isTree()
    {
        // Mark all the vertices as not visited and not part
        // of recursion stack
        Boolean visited[] = new Boolean[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // The call to isCyclicUtil serves multiple purposes
        // It returns true if graph reachable from vertex 0
        // is cyclic. It also marks all vertices reachable
        // from 0.
        if (isCyclicUtil(0, visited, -1))
            return false;

        // If we find a vertex which is not reachable from 0
        // (not marked by isCyclicUtil()), then we return false
        for (int u = 0; u < V; u++)
            if (!visited[u])
                return false;

        return true;
    }

    // Driver method
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g1 = new Graph(5);
    }
}

```

```
g1.addEdge(1, 0);
g1.addEdge(0, 2);
g1.addEdge(0, 3);
g1.addEdge(3, 4);
if (g1.isTree())
    System.out.println("Graph is Tree");
else
    System.out.println("Graph is not Tree");

Graph g2 = new Graph(5);
g2.addEdge(1, 0);
g2.addEdge(0, 2);
g2.addEdge(2, 1);
g2.addEdge(0, 3);
g2.addEdge(3, 4);

if (g2.isTree())
    System.out.println("Graph is Tree");
else
    System.out.println("Graph is not Tree");

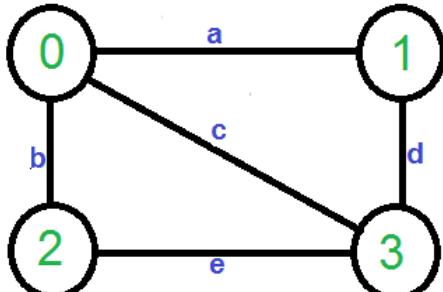
}
// This code is contributed by Aakash Hasija
```

Graph is Tree
Graph is not Tree

Kargers algorithm for Minimum Cut | Set 1 (Introduction and Implementation)

Given an undirected and unweighted graph, find the smallest cut (smallest number of edges that disconnects the graph into two components). The input graph may have parallel edges.

For example consider the following example, the smallest cut has 2 edges.



Min-Cut for above graph is either {a, d} OR {b, e}

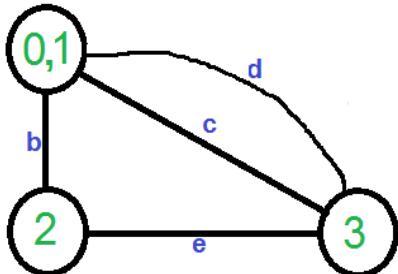
A Simple Solution use [Max-Flow based s-t cut algorithm](#) to find minimum cut. Consider every pair of vertices as source s and sink t, and call minimum s-t cut algorithm to find the s-t cut. Return minimum of all s-t cuts. Best possible time complexity of this algorithm is $O(V^5)$ for a graph. [How? there are total possible V^2 pairs and s-t cut algorithm for one pair takes $O(V^*E)$ time and $E = O(V^2)$].

Below is simple Kargers Algorithm for this purpose. Below Kargers algorithm can be implemented in $O(E) = O(V^2)$ time.

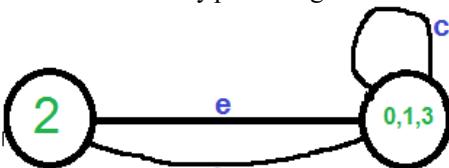
- 1) Initialize contracted graph CG as copy of original graph
- 2) While there are more than 2 vertices.
 - a) Pick a random edge (u, v) in the contracted graph.
 - b) Merge (or contract) u and v into a single vertex (update the contracted graph).
 - c) Remove self-loops
- 3) Return cut represented by two vertices.

Let us understand above algorithm through the example given.

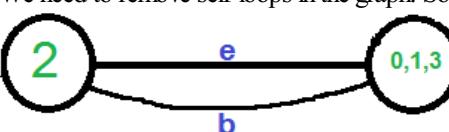
Let the first randomly picked vertex be **a** which connects vertices 0 and 1. We remove this edge and contract the graph (combine vertices 0 and 1). We get the following graph.



Let the next randomly picked edge be **d**. We remove this edge and combine vertices (0,1) and 3.

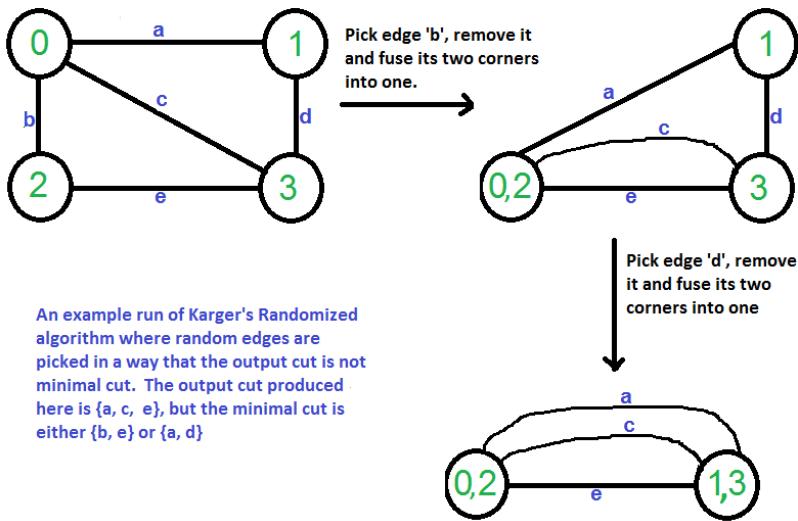


We need to remove self-loops in the graph. So we remove edge **c**.



Now graph has two vertices, so we stop. The number of edges in the resultant graph is the cut produced by Kargers algorithm.

Kargers algorithm is a Monte Carlo algorithm and cut produced by it may not be minimum. For example, the following diagram shows that a different order of picking random edges produces a min-cut of size 3.



Below is C++ implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

```

// Karger's algorithm to find Minimum Cut in an
// undirected, unweighted and connected graph.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// a structure to represent a unweighted edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a connected, undirected
// and unweighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after kargerMinCut() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// A very basic implementation of Karger's randomized
// algorithm for finding the minimum cut. Please note
// that Karger's algorithm is a Monte Carlo Randomized algo
// and the cut returned by the algorithm may not be
// minimum always
int kargerMinCut(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    struct subset *subsets = new subset[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 1;
    }

    // Perform random contractions
    while (E > 2)
    {
        // Pick a random edge
        int u = rand() % E;
        Edge e = edge[u];
        int x = find(subsets, e.src);
        int y = find(subsets, e.dest);

        if (x != y)
        {
            if (subsets[x].rank < subsets[y].rank)
                Union(subsets, x, y);
            else
                Union(subsets, y, x);
            E--;
        }
    }

    // Find the minimum cut
    int minCut = INT_MAX;
    for (int v = 0; v < V; v++)
    {
        if (subsets[v].parent == v)
        {
            int count = 0;
            for (int i = 0; i < E; i++)
            {
                Edge e = edge[i];
                if (find(subsets, e.src) == v)
                    count++;
            }
            if (count < minCut)
                minCut = count;
        }
    }

    return minCut;
}

```

```

        subsets[v].rank = 0;
    }

// Initially there are V vertices in
// contracted graph
int vertices = V;

// Keep contracting vertices until there are
// 2 vertices.
while (vertices > 2)
{
    // Pick a random edge
    int i = rand() % E;

    // Find vertices (or sets) of two corners
    // of current edge
    int subset1 = find(subsets, edge[i].src);
    int subset2 = find(subsets, edge[i].dest);

    // If two corners belong to same subset,
    // then no point considering this edge
    if (subset1 == subset2)
        continue;

    // Else contract the edge (or combine the
    // corners of edge into one vertex)
    else
    {
        printf("Contracting edge %d-%d\n",
               edge[i].src, edge[i].dest);
        vertices--;
        Union(subsets, subset1, subset2);
    }
}

// Now we have two vertices (or subsets) left in
// the contracted graph, so count the edges between
// two components and return the count.
int cutedges = 0;
for (int i=0; i<E; i++)
{
    int subset1 = find(subsets, edge[i].src);
    int subset2 = find(subsets, edge[i].dest);
    if (subset1 != subset2)
        cutedges++;
}

return cutedges;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent =
            find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {

```

```

        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// Driver program to test above functions
int main()
{
    /* Let us create following unweighted graph
       0-----1
       | \     |
       |   \   |
       |     \| |
       2-----3   */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;

    // Use a different seed value for every run.
    srand(time(NULL));

    printf("\nCut found by Karger's randomized algo is %d\n",
          kargerMinCut(graph));

    return 0;
}

```

Output:

```

Contracting edge 0-2
Contracting edge 0-3

Cut found by Karger's randomized algo is 2

```

Note that the above program is based on outcome of a random function and may produce different output.

In this post, we have discussed simple Kargers algorithm and have seen that the algorithm doesn't always produce min-cut. The above algorithm produces min-cut with probability greater or equal to that $1/(n^2)$. See next post on [Analysis and Applications of Kargers Algorithm](#), applications, proof of this probability and improvements are discussed.

References:

http://en.wikipedia.org/wiki/Karger%27s_algorithm

<https://www.youtube.com/watch?v=P0l8jMDQTEQ>

<https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf>

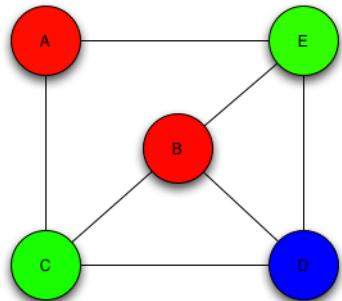
<http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/11/Small1.pdf>

Graph Coloring | Set 1 (Introduction and Applications)

[Graph coloring](#) problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like **Edge Coloring** (No vertex is incident to two edges of same color) and **Face Coloring** (Geographical Map Coloring) can be transformed into vertex coloring.

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 3 colors.



The problem to find chromatic number of a given graph is [NP Complete](#).

Applications of Graph Coloring:

The graph coloring problem has huge number of applications.

1) **Making Schedule or Time Table:** Suppose we want to make an exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). *How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams?* This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

2) **Mobile Radio Frequency Assignment:** When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

3) **Sudoku:** Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

4) **Register Allocation:** In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

5) **Bipartite Graphs:** We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See [this](#) for more details.

6) **Map Coloring:** Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See [Four Color Theorem](#))

There can be many more applications: For example the below reference video lecture has a case study at 1:18.

[Akamai](#) runs a network of thousands of servers and the servers are used to distribute content on Internet. They install a new software or update existing softwares pretty much every week. The update cannot be deployed on every server at the same time, because the server may have to be taken down for the install. Also, the update should not be done one at a time, because it will take a lot of time. There are sets of servers that cannot be taken down together, because they have certain critical functions. This is a typical scheduling application of graph coloring problem. It turned out that 8 colors were good enough to color the graph of 75000 nodes. So they could install updates in 8 passes.

We will soon be discussing different ways to solve the graph coloring problem.

References:

[Lec 6 | MIT 6.042J Mathematics for Computer Science, Fall 2010 | Video Lecture](#)

Graph Coloring | Set 2 (Greedy Algorithm)

We introduced [graph coloring and applications](#) in previous post. As discussed in the previous post, graph coloring is widely used. Unfortunately, there is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known [NP Complete problem](#). There are approximate algorithms to solve the problem though. Following is the basic Greedy Algorithm to assign colors. It doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors. The basic algorithm never uses more than $d+1$ colors where d is the maximum degree of a vertex in the given graph.

Basic Greedy Coloring Algorithm:

1. Color first vertex with first color.
2. Do following for remaining $V-1$ vertices.
 - a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v , assign a new color to it.

Following are C++ and Java implementations of the above Greedy Algorithm.

C++

```
// A C++ program to implement greedy algorithm for graph coloring
#include <iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; }

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Prints greedy coloring of the vertices
    void greedyColoring();
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// Assigns colors (starting from 0) to all vertices and prints
// the assignment of colors
void Graph::greedyColoring()
{
    int result[V];

    // Assign the first color to first vertex
    result[0] = 0;

    // Initialize remaining  $V-1$  vertices as unassigned
    for (int u = 1; u < V; u++)
        result[u] = -1; // no color is assigned to u

    // A temporary array to store the available colors. True
    // value of available[cr] would mean that the color cr is
    // assigned to one of its adjacent vertices
    bool available[V];
    for (int cr = 0; cr < V; cr++)
        available[cr] = false;

    // Assign colors to remaining  $V-1$  vertices
    for (int u = 1; u < V; u++)
    {
        // Process all adjacent vertices and flag their colors
        // as unavailable
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = true;
    }

    // Print the result
    cout << "Greedy Coloring: ";
    for (int u = 0; u < V; u++)
        cout << result[u] << " ";
}
```

```

// Find the first available color
int cr;
for (cr = 0; cr < V; cr++)
    if (available[cr] == false)
        break;

result[u] = cr; // Assign the found color

// Reset the values back to false for the next iteration
for (i = adj[u].begin(); i != adj[u].end(); ++i)
    if (result[*i] != -1)
        available[result[*i]] = false;
}

// print the result
for (int u = 0; u < V; u++)
    cout << "Vertex " << u << " ---> Color "
    << result[u] << endl;
}

// Driver program to test above function
int main()
{
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    cout << "Coloring of graph 1 \n";
    g1.greedyColoring();

    Graph g2(5);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(1, 2);
    g2.addEdge(1, 4);
    g2.addEdge(2, 4);
    g2.addEdge(4, 3);
    cout << "\nColoring of graph 2 \n";
    g2.greedyColoring();

    return 0;
}

```

Java

```

// A Java program to implement greedy algorithm for graph coloring
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        adj[w].add(v); //Graph is undirected
    }

    // Assigns colors (starting from 0) to all vertices and
    // prints the assignment of colors
    void greedyColoring()
    {

```

```

int result[] = new int[V];

// Assign the first color to first vertex
result[0] = 0;

// Initialize remaining V-1 vertices as unassigned
for (int u = 1; u < V; u++)
    result[u] = -1; // no color is assigned to u

// A temporary array to store the available colors. True
// value of available[cr] would mean that the color cr is
// assigned to one of its adjacent vertices
boolean available[] = new boolean[V];
for (int cr = 0; cr < V; cr++)
    available[cr] = false;

// Assign colors to remaining V-1 vertices
for (int u = 1; u < V; u++)
{
    // Process all adjacent vertices and flag their colors
    // as unavailable
    Iterator<Integer> it = adj[u].iterator();
    while (it.hasNext())
    {
        int i = it.next();
        if (result[i] != -1)
            available[result[i]] = true;
    }

    // Find the first available color
    int cr;
    for (cr = 0; cr < V; cr++)
        if (available[cr] == false)
            break;

    result[u] = cr; // Assign the found color

    // Reset the values back to false for the next iteration
    it = adj[u].iterator();
    while (it.hasNext())
    {
        int i = it.next();
        if (result[i] != -1)
            available[result[i]] = false;
    }
}

// print the result
for (int u = 0; u < V; u++)
    System.out.println("Vertex " + u + " ---> Color "
                      + result[u]);
}

// Driver method
public static void main(String args[])
{
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    System.out.println("Coloring of graph 1");
    g1.greedyColoring();

    System.out.println();
    Graph g2 = new Graph(5);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(1, 2);
    g2.addEdge(1, 4);
    g2.addEdge(2, 4);
    g2.addEdge(4, 3);
    System.out.println("Coloring of graph 2 ");
    g2.greedyColoring();
}
}

// This code is contributed by Aakash Hasija

```

```

Coloring of graph 1
Vertex 0 ----> Color 0
Vertex 1 ----> Color 1
Vertex 2 ----> Color 2
Vertex 3 ----> Color 0
Vertex 4 ----> Color 1

```

```

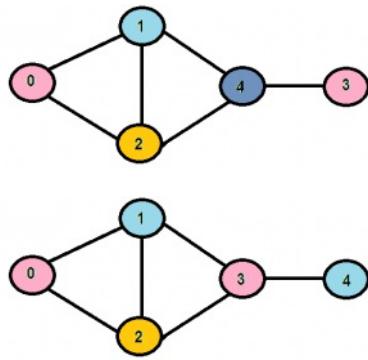
Coloring of graph 2
Vertex 0 ----> Color 0
Vertex 1 ----> Color 1
Vertex 2 ----> Color 2
Vertex 3 ----> Color 0
Vertex 4 ----> Color 3

```

Time Complexity: $O(V^2 + E)$ in worst case.

Analysis of Basic Algorithm

The above algorithm doesn't always use minimum number of colors. Also, the number of colors used sometime depend on the order in which vertices are processed. For example, consider the following two graphs. Note that in graph on right side, vertices 3 and 4 are swapped. If we consider the vertices 0, 1, 2, 3, 4 in left graph, we can color the graph using 3 colors. But if we consider the vertices 0, 1, 2, 3, 4 in right graph, we need 4 colors.



So the order in which the vertices are picked is important. Many people have suggested different ways to find an ordering that work better than the basic algorithm on average. The most common is [Welsh-Powell Algorithm](#) which considers vertices in descending order of degrees.

How does the basic algorithm guarantee an upper bound of $d+1$?

Here d is the maximum degree in the given graph. Since d is maximum degree, a vertex cannot be attached to more than d vertices. When we color a vertex, at most d colors could have already been used by its adjacent. To color this vertex, we need to pick the smallest numbered color that is not used by the adjacent vertices. If colors are numbered like 1, 2, .., then the value of such smallest number must be between 1 to $d+1$ (Note that d numbers are already picked by adjacent vertices).

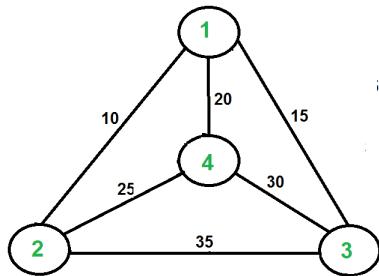
This can also be proved using induction. See [this](#) video lecture for proof.

We will soon be discussing some interesting facts about chromatic number and graph coloring.

Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between [Hamiltonian Cycle](#) and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is $10+25+30+15$ which is 80.

The problem is a famous [NP hard](#) problem. There is no polynomial time known solution for this problem.

Following are different solutions for the traveling salesman problem.

Naive Solution:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all $(n-1)!$ [Permutations](#) of cities.
- 3) Calculate cost of every permutation and keep track of minimum cost permutation.
- 4) Return the permutation with minimum cost.

Time Complexity: $?(n!)$

Dynamic Programming:

Let the given set of vertices be $\{1, 2, 3, 4, \dots, n\}$. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be $\text{cost}(i)$, the cost of corresponding Cycle would be $\text{cost}(i) + \text{dist}(i, 1)$ where $\text{dist}(i, 1)$ is the distance from i to 1. Finally, we return the minimum of all $[\text{cost}(i) + \text{dist}(i, 1)]$ values. This looks simple so far. Now the question is how to get $\text{cost}(i)$?

To calculate $\text{cost}(i)$ using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term $C(S, i)$ be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i .

We start with all subsets of size 2 and calculate $C(S, i)$ for all subsets where S is the subset, then we calculate $C(S, i)$ for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},  
C(S, i) = dist(1, i)  
Else if size of S is greater than 2.  
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

For a set of size n , we consider $n-2$ subsets each of size $n-1$ such that all subsets don't have nth in them.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than $O(n!)$, but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

We will soon be discussing approximate algorithms for travelling salesman problem.

References:

- <http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf>
- <http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf>

Travelling Salesman Problem | Set 2 (Approximate using MST)

We introduced [Travelling Salesman Problem](#) and discussed Naive and Dynamic Programming Solutions for the problem in the [previous post](#). Both of the solutions are infeasible. In fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate algorithms to solve the problem though. The approximate algorithms work only if the problem instance satisfies Triangle-Inequality.

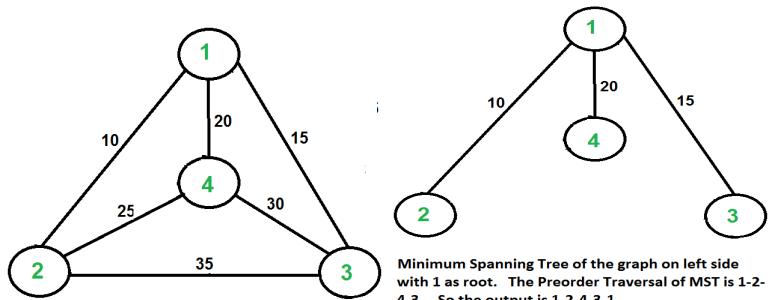
Triangle-Inequality: The least distant path to reach a vertex j from i is always to reach j directly from i , rather than through some other vertex k (or vertices), i.e., $\text{dis}(i, j) \leq \text{dis}(i, k) + \text{dist}(k, j)$. The Triangle-Inequality holds in many practical situations.

When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use Minimum Spanning Tree (MST). Following is the MST based algorithm.

Algorithm:

- 1) Let 1 be the starting and ending point for salesman.
- 2) Construct MST from with 1 as root using [Prims Algorithm](#)
- 3) List vertices visited in preorder walk of the constructed MST and add 1 at the end.

Let us consider the following example. The first diagram is the given graph. The second diagram shows MST constructed with 1 as root. The preorder traversal of MST is 1-2-4-3. Adding 1 at the end gives 1-2-4-3-1 which is the output of this algorithm.



In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

How is this algorithm 2-approximate? The cost of the output produced by the above algorithm is never more than twice the cost of best possible output. Let us see how is this guaranteed by the above algorithm.

Let us define a term **full walk** to understand this. A full walk is lists all vertices when they are first visited in preorder, it also lists vertices when they are returned after a subtree is visited in preorder. The full walk of above tree would be 1-2-1-4-1-3-1.

Following are some important facts that prove the 2-approximateneess.

- 1) The cost of best possible Travelling Salesman tour is never less than the cost of MST. (The definition of [MST](#) says, it is a minimum cost tree that connects all vertices).
- 2) The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)
- 3) The output of the above algorithm is less than the cost of full walk. In above algorithm, we print preorder walk as output. In preorder walk, two or more edges of full walk are replaced with a single edge. For example, 2-1 and 1-4 are replaced by 1 edge 2-4. So if the graph follows triangle inequality, then this is always true.

From the above three statements, we can conclude that the cost of output produced by the approximate algorithm is never more than twice the cost of best possible solution.

We have discussed a very simple 2-approximate algorithm for the travelling salesman problem. There are other better approximate algorithms for the problem. For example [Christofides algorithm](#) is 1.5 approximate algorithm. We will soon be discussing these algorithms as separate posts.

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)
<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/TSP/tsp.htm>

Backtracking | Set 6 (Hamiltonian Cycle)

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

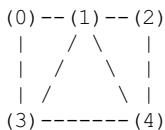
Input:

A 2D array $\text{graph}[V][V]$ where V is the number of vertices in graph and $\text{graph}[V][V]$ is adjacency matrix representation of the graph. A value $\text{graph}[i][j]$ is 1 if there is a direct edge from i to j , otherwise $\text{graph}[i][j]$ is 0.

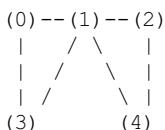
Output:

An array $\text{path}[V]$ that should contain the Hamiltonian Path. $\text{path}[i]$ should represent the i th vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is $\{0, 1, 2, 4, 3, 0\}$. There are more Hamiltonian Cycles in the graph like $\{0, 3, 4, 2, 1, 0\}$



And the following graph doesn't contain any Hamiltonian Cycle.



Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be $n!$ (n factorial) configurations.

```
while there are untried conflagrations
{
    generate the next configuration
    if ( there are edges between two consecutive vertices of this
        configuration and there is an edge from the last vertex to
        the first ).
    {
        print this configuration;
        break;
    }
}
```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

Following are implementations of the Backtracking solution.

C/C++

```
/* C/C++ program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously
       added vertex. */
    if (graph [path[pos-1]] [v] == 0)
```

```

        return false;

/* Check if the vertex has already been included.
   This step can be optimized by creating an array of size V */
for (int i = 0; i < pos; i++)
    if (path[i] == v)
        return false;

return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        if (graph[path[pos-1]][path[0]] == 1)
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil(graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,
       then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using Backtracking.
It mainly uses hamCycleUtil() to solve the problem. It returns false
if there is no Hamiltonian Cycle possible, otherwise return true and
prints the path. Please note that there may be more than one solutions,
this function prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false )
    {
        printf("\nSolution does not exist");
        return false;
    }

    printSolution(path);
    return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf ("Solution Exists:\n"
           " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);
}

```

```

// Let us print the first vertex again to show the complete cycle
printf("%d ", path[0]);
printf("\n");
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
       (0)--(1)--(2)
       |   / \   |
       |   /   \  |
       | /     \ |
       (3)-----(4) */
    bool graph1[V][V] = {{0, 1, 0, 1, 0},
                         {1, 0, 1, 1, 1},
                         {0, 1, 0, 0, 1},
                         {1, 1, 0, 0, 1},
                         {0, 1, 1, 1, 0},
                         };

    // Print the solution
    hamCycle(graph1);

    /* Let us create the following graph
       (0)--(1)--(2)
       |   / \   |
       |   /   \  |
       | /     \ |
       (3)-----(4) */
    bool graph2[V][V] = {{0, 1, 0, 1, 0},
                         {1, 0, 1, 1, 1},
                         {0, 1, 0, 0, 1},
                         {1, 1, 0, 0, 0},
                         {0, 1, 1, 0, 0},
                         };

    // Print the solution
    hamCycle(graph2);
}

return 0;
}

```

Java

```

/* Java program for solution of Hamiltonian Cycle problem
using backtracking */
class HamiltonianCycle
{
    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
       added at index 'pos' in the Hamiltonian Cycle
       constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][], int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
           the previously added vertex. */
        if (graph[path[pos - 1]][v] == 0)
            return false;

        /* Check if the vertex has already been included.
           This step can be optimized by creating an array
           of size V */
        for (int i = 0; i < pos; i++)
            if (path[i] == v)
                return false;

        return true;
    }

    /* A recursive utility function to solve hamiltonian
       cycle problem */
    boolean hamCycleUtil(int graph[][], int path[], int pos)
    {
        /* base case: If all vertices are included in
           Hamiltonian Cycle */
        if (pos == V)

```

```

{
    // And if there is an edge from the last included
    // vertex to the first vertex
    if (graph[path[pos - 1]][path[0]] == 1)
        return true;
    else
        return false;
}

// Try different vertices as a next candidate in
// Hamiltonian Cycle. We don't try for 0 as we
// included 0 as starting point in hamCycle()
for (int v = 1; v < V; v++)
{
    /* Check if this vertex can be added to Hamiltonian
     * Cycle */
    if (isSafe(v, graph, path, pos))
    {
        path[pos] = v;

        /* recur to construct rest of the path */
        if (hamCycleUtil(graph, path, pos + 1) == true)
            return true;

        /* If adding vertex v doesn't lead to a solution,
         * then remove it */
        path[pos] = -1;
    }
}

/* If no vertex can be added to Hamiltonian Cycle
 * constructed so far, then return false */
return false;
}

/* This function solves the Hamiltonian Cycle problem using
 * Backtracking. It mainly uses hamCycleUtil() to solve the
 * problem. It returns false if there is no Hamiltonian Cycle
 * possible, otherwise return true and prints the path.
 * Please note that there may be more than one solutions,
 * this function prints one of the feasible solutions. */
int hamCycle(int graph[][])
{
    path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path.
       If there is a Hamiltonian Cycle, then the path can be
       started from any point of the cycle as the graph is
       undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        System.out.println("\nSolution does not exist");
        return 0;
    }

    printSolution(path);
    return 1;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +
                       " is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");

    // Let us print the first vertex again to show the
    // complete cycle
    System.out.println(" " + path[0] + " ");
}

// driver program to test above function
public static void main(String args[])
{
    HamiltonianCycle hamiltonian =
                    new HamiltonianCycle();
    /* Let us create the following graph

```

```

(0)--(1)--(2)
|   / \   |
|   / \   |
| /   \   |
(3)-----(4)   */
int graph1[][] = {{0, 1, 0, 1, 0},
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 1},
{0, 1, 1, 1, 0},
};

// Print the solution
hamiltonian.hamCycle(graph1);

/* Let us create the following graph
(0)--(1)--(2)
|   / \   |
|   / \   |
| /   \   |
(3)      (4)   */
int graph2[][] = {{0, 1, 0, 1, 0},
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 0},
{0, 1, 1, 0, 0},
};

// Print the solution
hamiltonian.hamCycle(graph2);
}
}

// This code is contributed by Abhishek Shankhadhar

```

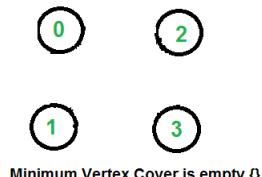
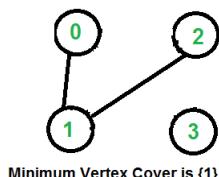
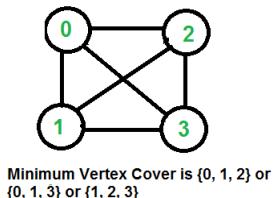
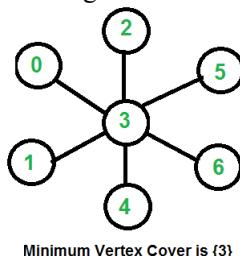
Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0

Solution does not exist

Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either u or v is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. **Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.**

Following are some examples.

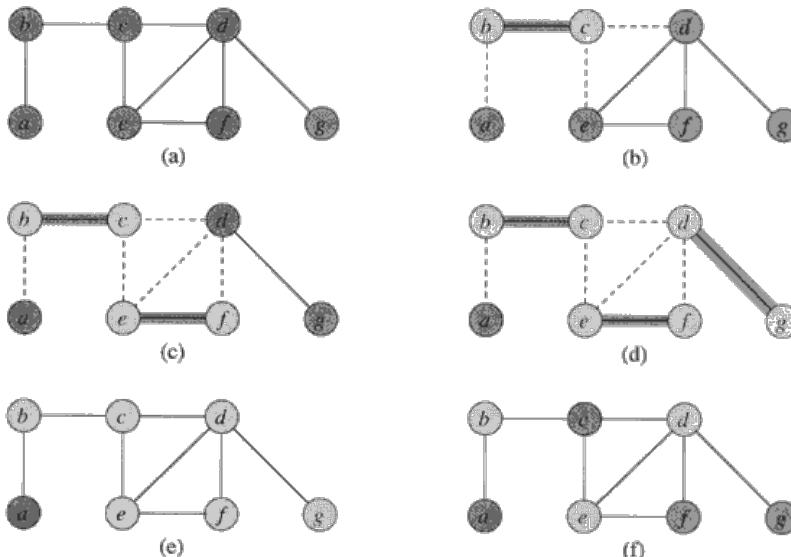


[Vertex Cover Problem](#) is a known [NP Complete problem](#), i.e., there is no polynomial time solution for this unless $P = NP$. There are approximate polynomial time algorithms to solve the problem though. Following is a simple approximate algorithm adapted from [CLRS book](#).

Approximate Algorithm for Vertex Cover:

- 1) Initialize the result as {}
- 2) Consider a set of all edges in given graph. Let the set be E .
- 3) Do following while E is not empty
 - ...a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
 - ...b) Remove all edges from E which are either incident on u or v .
- 4) Return result

Following diagram taken from [CLRS book](#) shows execution of above approximate algorithm.



How well the above algorithm perform?

It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of minimum possible vertex cover (Refer [this](#) for proof).

Implementation:

Following are C++ and Java implementations of above approximate algorithm.

C++

```
// Program to print Vertex Cover of a given undirected graph
#include<iostream>
```

```

#include <list>
using namespace std;

// This class represents a undirected graph using adjacency list
class Graph
{
    int V;      // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void printVertexCover(); // prints vertex cover
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
    adj[w].push_back(v); // Since the graph is undirected
}

// The function to print vertex cover
void Graph::printVertexCover()
{
    // Initialize all vertices as not visited.
    bool visited[V];
    for (int i=0; i<V; i++)
        visited[i] = false;

    list<int>::iterator i;

    // Consider all edges one by one
    for (int u=0; u<V; u++)
    {
        // An edge is only picked when both visited[u] and visited[v]
        // are false
        if (visited[u] == false)
        {
            // Go through all adjacents of u and pick the first not
            // yet visited vertex (We are basically picking an edge
            // (u, v) from remaining edges.
            for (i= adj[u].begin(); i != adj[u].end(); ++i)
            {
                int v = *i;
                if (visited[v] == false)
                {
                    // Add the vertices (u, v) to the result set.
                    // We make the vertex u and v visited so that
                    // all edges from/to them would be ignored
                    visited[v] = true;
                    visited[u] = true;
                    break;
                }
            }
        }
    }

    // Print the vertex cover
    for (int i=0; i<V; i++)
        if (visited[i])
            cout << i << " ";
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 5);
    g.addEdge(5, 6);

    g.printVertexCover();
}

```

```
    return 0;
}
```

Java

```
// Java Program to print Vertex Cover of a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);    // Add w to v's list.
        adj[w].add(v);    //Graph is undirected
    }

    // The function to print vertex cover
    void printVertexCover()
    {
        // Initialize all vertices as not visited.
        boolean visited[] = new boolean[V];
        for (int i=0; i<V; i++)
            visited[i] = false;

        Iterator<Integer> i;

        // Consider all edges one by one
        for (int u=0; u<V; u++)
        {
            // An edge is only picked when both visited[u]
            // and visited[v] are false
            if (visited[u] == false)
            {
                // Go through all adjacents of u and pick the
                // first not yet visited vertex (We are basically
                // picking an edge (u, v) from remaining edges.
                i = adj[u].iterator();
                while (i.hasNext())
                {
                    int v = i.next();
                    if (visited[v] == false)
                    {
                        // Add the vertices (u, v) to the result
                        // set. We make the vertex u and v visited
                        // so that all edges from/to them would
                        // be ignored
                        visited[v] = true;
                        visited[u] = true;
                        break;
                    }
                }
            }
        }

        // Print the vertex cover
        for (int j=0; j<V; j++)
            if (visited[j])
                System.out.print(j+" ");
    }

    // Driver method
}
```

```

public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 5);
    g.addEdge(5, 6);

    g.printVertexCover();
}
// This code is contributed by Aakash Hasija

```

0 1 3 4 5 6

Time Complexity of above algorithm is $O(V + E)$.

Exact Algorithms:

Although the problem is NP complete, it can be solved in polynomial time for following types of graphs.

- 1) [Bipartite Graph](#)
- 2) [Tree Graph](#)

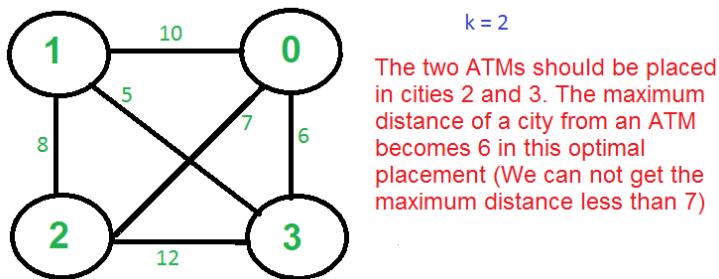
The problem to check whether there is a vertex cover of size smaller than or equal to a given number k can also be solved in polynomial time if k is bounded by $O(\log V)$ (Refer [this](#))

We will soon be discussing exact algorithms for vertex cover.

K Centers Problem | Set 1 (Greedy Approximate Algorithm)

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs) such that the maximum distance of a city to a warehouse (or ATM) is minimized.

For example consider the following four cities, 0, 1, 2 and 3 and distances between them, how do place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized.



There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a solution which is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](#) (Distance between two points is always smaller than sum of distances through a third point).

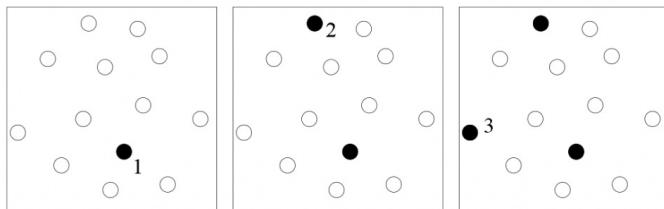
The 2-Approximate Greedy Algorithm:

1) Choose the first center arbitrarily.

2) Choose remaining $k-1$ centers using the following criteria.

Let c_1, c_2, c_3, c_i be the already chosen centers. Choose $(i+1)$ th center by picking the city which is farthest from already selected centers, i.e, the point p which has following value as maximum $\text{Min}[\text{dist}(p, c_1), \text{dist}(p, c_2), \text{dist}(p, c_3), \dots, \text{dist}(p, c_i)]$

The following diagram taken from [here](#) illustrates above algorithm



Example ($k = 3$ in the above shown Graph)

a) Let the first arbitrarily picked vertex be 0.

b) The next vertex is 1 because 1 is the farthest vertex from 0.

c) Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates following values.

Minimum of all distanced from 2 to already considered centers

$$\text{Min}[\text{dist}(2, 0), \text{dist}(2, 1)] = \text{Min}[7, 8] = 7$$

Minimum of all distanced from 3 to already considered centers

$$\text{Min}[\text{dist}(3, 0), \text{dist}(3, 1)] = \text{Min}[6, 5] = 5$$

After computing the above values, the city 2 is picked as the value corresponding to 2 is maximum.

Note that the greedy algorithm doesn't give best solution for $k = 2$ as this is just an approximate algorithm with bound as twice of optimal.

Proof that the above greedy algorithm is 2 approximate.

Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from Greedy algorithm is $2 * \text{OPT}$.

The proof can be done using contradiction.

a) Assume that the distance from the furthest point to all centers is $> 2\text{OPT}$.

- b) This means that distances between all centers are also $> 2\text{OPT}$.
- c) We have $k + 1$ points with distances $> 2\text{OPT}$ between every pair.
- d) Each point has a center of the optimal solution with distance $\leq \text{OPT}$ to it.
- e) There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, $k+1$ points)
- f) The distance between them is at most 2OPT (triangle inequality) which is a contradiction.

Source:

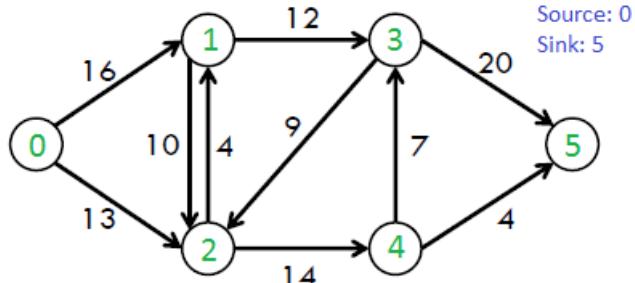
<http://algo2.iti.kit.edu/vanstee/courses/kcenter.pdf>

Ford-Fulkerson Algorithm for Maximum Flow Problem

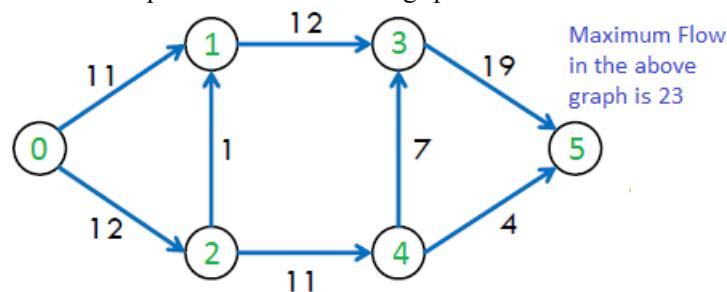
Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source s* and *sink t* in the graph, find the maximum possible flow from s to t with following constraints:

- a) Flow on an edge doesn't exceed the given capacity of the edge.
- b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



Ford-Fulkerson Algorithm

The following is simple idea of Ford-Fulkerson algorithm:

- 1) Start with initial flow as 0.
- 2) While there is a augmenting path from source to sink.
 - Add this path-flow to flow.
- 3) Return flow.

Time Complexity: Time complexity of the above algorithm is $O(\text{max_flow} * E)$. We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes $O(\text{max_flow} * E)$.

How to implement the above simple algorithm?

Let us first define the concept of Residual Graph which is needed for understanding the implementation.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called **residual capacity** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.

Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.

The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges. We need to add path flow along reverse edges because may later need to send flow in reverse direction (See following video for example).

<http://www.youtube.com/watch?v=-8MwfqB-lyM>

Following are C++ and Java implementations of Ford-Fulkerson algorithm. To keep things simple, graph is represented as a 2D matrix.

C++

```
// C++ program for implementation of Ford Fulkerson algorithm
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
```

```

#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (visited[v]==false && rGraph[u][v] > 0)
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // If we reached sink in BFS starting from source, then return
    // true, else false
    return (visited[t] == true);
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                      // residual capacity of edge from i to j (if there
                      // is an edge. If rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and reverse edges
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
    }
}

```

```

        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                    };

    cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 5);

    return 0;
}

```

Java

```

// Java program for implementation of Ford Fulkerson algorithm
import java.util.*;
import java.lang.*;
import java.io.*;
import java.util.LinkedList;

class MaxFlow
{
    static final int V = 6; //Number of vertices in graph

    /* Returns true if there is a path from source 's' to sink
     't' in residual graph. Also fills parent[] to store the
     path */
    boolean bfs(int rGraph[][], int s, int t, int parent[])
    {
        // Create a visited array and mark all vertices as not
        // visited
        boolean visited[] = new boolean[V];
        for(int i=0; i<V; ++i)
            visited[i]=false;

        // Create a queue, enqueue source vertex and mark
        // source vertex as visited
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(s);
        visited[s] = true;
        parent[s]=-1;

        // Standard BFS Loop
        while (queue.size()!=0)
        {
            int u = queue.poll();

            for (int v=0; v<V; v++)
            {
                if (visited[v]==false && rGraph[u][v] > 0)
                {
                    queue.add(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }

        // If we reached sink in BFS starting from source, then
        // return true, else false
        return (visited[t] == true);
    }

    // Returns the maximum flow from s to t in the given graph
    int fordFulkerson(int graph[][], int s, int t)
    {
        int u, v;

```

```

// Create a residual graph and fill the residual graph
// with given capacities in the original graph as
// residual capacities in residual graph

// Residual graph where rGraph[i][j] indicates
// residual capacity of edge from i to j (if there
// is an edge. If rGraph[i][j] is 0, then there is
// not)
int rGraph[][] = new int[V][V];

for (u = 0; u < V; u++)
    for (v = 0; v < V; v++)
        rGraph[u][v] = graph[u][v];

// This array is filled by BFS and to store path
int parent[] = new int[V];

int max_flow = 0; // There is no flow initially

// Augment the flow while there is path from source
// to sink
while (bfs(rGraph, s, t, parent))
{
    // Find minimum residual capacity of the edges
    // along the path filled by BFS. Or we can say
    // find the maximum flow through the path found.
    int path_flow = Integer.MAX_VALUE;
    for (v=t; v!=s; v=parent[v])
    {
        u = parent[v];
        path_flow = Math.min(path_flow, rGraph[u][v]);
    }

    // update residual capacities of the edges and
    // reverse edges along the path
    for (v=t; v != s; v=parent[v])
    {
        u = parent[v];
        rGraph[u][v] -= path_flow;
        rGraph[v][u] += path_flow;
    }

    // Add path flow to overall flow
    max_flow += path_flow;
}

// Return the overall flow
return max_flow;
}

// Driver program to test above functions
public static void main (String[] args) throws java.lang.Exception
{
    // Let us create a graph shown in the above example
    int graph[][] =new int[][] { {0, 16, 13, 0, 0, 0},
                                {0, 0, 10, 12, 0, 0},
                                {0, 4, 0, 0, 14, 0},
                                {0, 0, 9, 0, 0, 20},
                                {0, 0, 0, 7, 0, 4},
                                {0, 0, 0, 0, 0, 0}
                            };
    MaxFlow m = new MaxFlow();

    System.out.println("The maximum possible flow is " +
                       m.fordFulkerson(graph, 0, 5));
}
}

```

The maximum possible flow is 23

The above implementation of Ford Fulkerson Algorithm is called [Edmonds-Karp Algorithm](#). The idea of Edmonds-Karp is to use BFS in Ford Fulkerson implementation as BFS always picks a path with minimum number of edges. When BFS is used, the worst case time complexity can be reduced to $O(VE^2)$. The above implementation uses adjacency matrix representation though where BFS takes $O(V^2)$ time, the time complexity of the above implementation is $O(EV^3)$ (Refer [CLRS book](#) for proof of time complexity)

This is an important problem as it arises in many practical situations. Examples include, maximizing the transportation with given traffic limits,

maximizing packet flow in computer networks.

Exercise:

Modify the above implementation so that it runs in $O(VE^2)$ time.

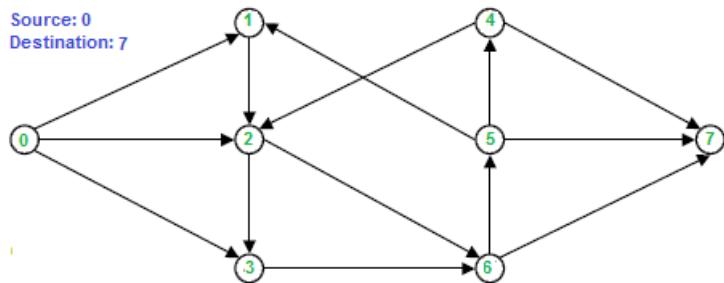
References:

<http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>

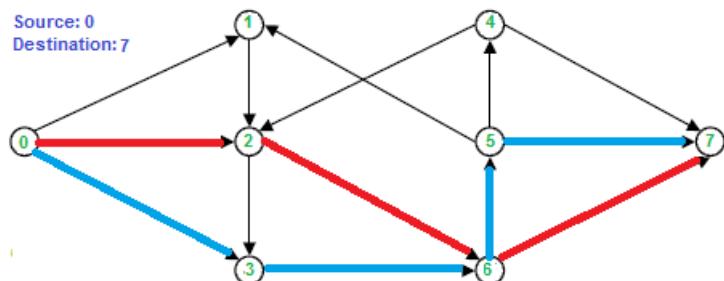
[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

Find maximum number of edge disjoint paths between two vertices

Given a directed graph and two vertices in it, source s and destination t, find out the maximum number of edge disjoint paths from s to t. Two paths are said edge disjoint if they dont share any edge.



There can be maximum two edge disjoint paths from source 0 to destination 7 in the above graph. Two edge disjoint paths are highlighted below in red and blue colors are 0-2-6-7 and 0-3-6-5-7.



Note that the paths may be different, but the maximum number is same. For example, in the above diagram, another possible set of paths is 0-1-2-6-7 and 0-3-6-5-7 respectively.

This problem can be solved by reducing it to [maximum flow problem](#). Following are steps.

- 1) Consider the given source and destination as source and sink in flow network. Assign unit capacity to each edge.
- 2) Run Ford-Fulkerson algorithm to find the maximum flow from source to sink.
- 3) The maximum flow is equal to the maximum number of edge-disjoint paths.

When we run Ford-Fulkerson, we reduce the capacity by a unit. Therefore, the edge can not be used again. So the maximum flow is equal to the maximum number of edge-disjoint paths.

Following is C++ implementation of the above algorithm. Most of the code is taken from [here](#).

```
// C++ program to find maximum number of edge disjoint paths
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 8

/* Returns true if there is a path from source 's' to sink 't' in
residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
            if (rGraph[u][v] > 0 && !visited[v])
                {
                    parent[v] = u;
                    if (v == t)
                        return true;
                    q.push(v);
                    visited[v] = true;
                }
    }
    return false;
}
```

```

    {
        if (visited[v]==false && rGraph[u][v] > 0)
        {
            q.push(v);
            parent[v] = u;
            visited[v] = true;
        }
    }
}

// If we reached sink in BFS starting from source, then return
// true, else false
return (visited[t] == true);
}

// Returns the maximum number of edge-disjoint paths from s to t.
// This function is copy of forFulkerson() discussed at http://goo.gl/wtQ4Ks
int findDisjointPaths(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                      // residual capacity of edge from i to j (if there
                      // is an edge. If rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;

        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and reverse edges
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow (max_flow is equal to maximum
    // number of edge-disjoint paths)
    return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 1, 1, 1, 0, 0, 0, 0, 0},
                        {0, 0, 1, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 1, 0, 0, 1, 0, 0},
                        {0, 0, 0, 0, 0, 0, 1, 0, 0},
                        {0, 0, 1, 0, 0, 0, 0, 0, 1},
                        {0, 1, 0, 0, 0, 0, 0, 1, 0},
                        {0, 0, 0, 0, 0, 1, 0, 0, 1},
                        {0, 0, 0, 0, 0, 0, 0, 0, 0}
                    };
}

```

```
int s = 0;
int t = 7;
cout << "There can be maximum " << findDisjointPaths(graph, s, t)
     << " edge-disjoint paths from " << s << " to " << t ;
return 0;
}
```

Output:

There can be maximum 2 edge-disjoint paths from 0 to 7

Time Complexity: Same as time complexity of Edmonds-Karp implementation of Ford-Fulkerson (See time complexity discussed [here](#))

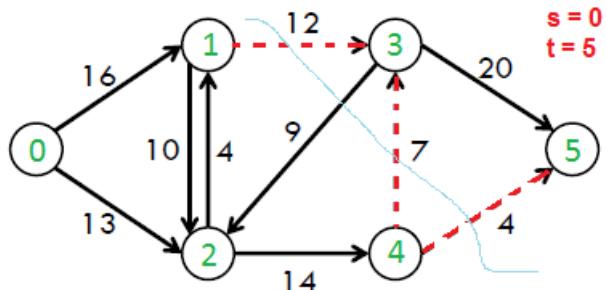
References:

<http://www.win.tue.nl/~nikhil/courses/2012/2WO08/max-flow-applications-4up.pdf>

Find minimum s-t cut in a flow network

In a flow network, an s-t cut is a cut that requires the source s and the sink t to be in different subsets, and it consists of edges going from the sources side to the sinks side. The capacity of an s-t cut is defined by the sum of capacity of each edge in the cut-set. (Source: [Wiki](#))
The problem discussed here is to find minimum capacity s-t cut of the given network. Expected output is all edges of the minimum cut.

For example, in the following flow network, example s-t cuts are $\{\{0,1\}, \{0,2\}\}$, $\{\{0,2\}, \{1,2\}\}$, $\{\{1,2\}, \{1,3\}\}$, etc. The minimum s-t cut is $\{\{1,3\}, \{4,5\}\}$ which has capacity as $12+7+4 = 23$.



We strongly recommend to read the below post first.

[Ford-Fulkerson Algorithm for Maximum Flow Problem](#)

Minimum Cut and Maximum Flow

Like [Maximum Bipartite Matching](#), this is another problem which can be solved using [Ford-Fulkerson Algorithm](#). This is based on max-flow min-cut theorem.

The [max-flow min-cut theorem](#) states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut. See [CLRS book](#) for proof of this theorem.

From Ford-Fulkerson, we get capacity of minimum cut. How to print all edges that form the minimum cut? The idea is to use [residual graph](#).

Following are steps to print all edges of minimum cut.

- 1) Run Ford-Fulkerson algorithm and consider the final [residual graph](#).
- 2) Find the set of vertices that are reachable from source in the residual graph.
- 3) All edges which are from a reachable vertex to non-reachable vertex are minimum cut edges. Print all such edges.

Following is C++ implementation of the above approach.

C++

```
// C++ program for finding minimum cut using Ford-Fulkerson
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
residual graph. Also fills parent[] to store the path */
int bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
```

```

q.pop();

for (int v=0; v<V; v++)
{
    if (visited[v]==false && rGraph[u][v] > 0)
    {
        q.push(v);
        parent[v] = u;
        visited[v] = true;
    }
}

// If we reached sink in BFS starting from source, then return
// true, else false
return (visited[t] == true);
}

// A DFS based function to find all reachable vertices from s. The function
// marks visited[i] as true if i is reachable from s. The initial values in
// visited[] must be false. We can also use BFS to find reachable vertices
void dfs(int rGraph[V][V], int s, bool visited[])
{
    visited[s] = true;
    for (int i = 0; i < V; i++)
        if (rGraph[s][i] && !visited[i])
            dfs(rGraph, i, visited);
}

// Prints the minimum s-t cut
void minCut(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // rGraph[i][j] indicates residual capacity of edge i-j
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and reverse edges
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
    }

    // Flow is maximum now, find vertices reachable from s
    bool visited[V];
    memset(visited, false, sizeof(visited));
    dfs(rGraph, s, visited);

    // Print all edges that are from a reachable vertex to
    // non-reachable vertex in the original graph
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (visited[i] && !visited[j] && graph[i][j])
                cout << i << " - " << j << endl;
    return;
}

```

```

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                    };

    minCut(graph, 0, 5);

    return 0;
}

```

Java

```

// Java program for implementation of Ford Fulkerson algorithm
import java.util.*;
import java.lang.*;
import java.io.*;
import java.util.LinkedList;

class MaxFlow
{
    static final int V=6;    //Number of vertices in graph

    /* Returns true if there is a path from source 's' to sink 't'
       in residual graph. Also fills parent[] to store the path */
    boolean bfs(int rGraph[][], int s, int t, int parent[])
    {
        // Create a visited array and mark all vertices as not visited
        boolean visited[] = new boolean[V];
        for(int i=0;i<V;++i)
            visited[i]=false;

        // Create a queue, enqueue source vertex and mark source vertex
        // as visited
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(s);
        visited[s] = true;
        parent[s]=-1;

        // Standard BFS Loop
        while (queue.size()!=0)
        {
            int u = queue.poll();

            for (int v=0; v<V; v++)
            {
                if (visited[v]==false && rGraph[u][v] > 0)
                {
                    queue.add(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }

        // If we reached sink in BFS starting from source, then return
        // true, else false
        return (visited[t] == true);
    }

    // Returns the maximum flow from s to t in the given graph
    int fordFulkerson(int graph[][], int s, int t)
    {
        int u, v;

        // Create a residual graph and fill the residual graph with
        // given capacities in the original graph as residual capacities
        // in residual graph
        // Residual graph where rGraph[i][j] indicates residual capacity
        // of edge from i to j (if there is an edge. If rGraph[i][j] is 0,
        // then there is not)
        int rGraph[][] = new int[V][V];

```

```

for (u = 0; u < V; u++)
    for (v = 0; v < V; v++)
        rGraph[u][v] = graph[u][v];

int parent[] = new int[V]; // filled by BFS and to store path

int max_flow = 0; // There is no flow initially

// Augment the flow while there is path from source to sink
while (bfs(rGraph, s, t, parent))
{
    // Find minimum residual capacity of the edges along the
    // path filled by BFS. Or we can say find the maximum flow
    // through the path found.
    int path_flow = Integer.MAX_VALUE;
    for (v=t; v!=s; v=parent[v])
    {
        u = parent[v];
        path_flow = Math.min(path_flow, rGraph[u][v]);
    }

    // update residual capacities of the edges and reverse edges
    // along the path
    for (v=t; v != s; v=parent[v])
    {
        u = parent[v];
        rGraph[u][v] -= path_flow;
        rGraph[v][u] += path_flow;
    }

    // Add path flow to overall flow
    max_flow += path_flow;
}

// Return the overall flow
return max_flow;
}

// Driver program to test above functions
public static void main (String[] args) throws java.lang.Exception
{
    // Let us create a graph shown in the above example
    int graph[][] =new int[][] { {0, 16, 13, 0, 0, 0},
                                {0, 0, 10, 12, 0, 0},
                                {0, 4, 0, 0, 14, 0},
                                {0, 0, 9, 0, 0, 20},
                                {0, 0, 0, 7, 0, 4},
                                {0, 0, 0, 0, 0, 0}
                            };
    MaxFlow m = new MaxFlow();

    System.out.println("The maximum possible flow is " +
                       m.fordFulkerson(graph, 0, 5));
}
}

```

1 - 3
4 - 3
4 - 5

References:

<http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>
<http://www.cs.princeton.edu/courses/archive/spring06/cos226/lectures/maxflow.pdf>

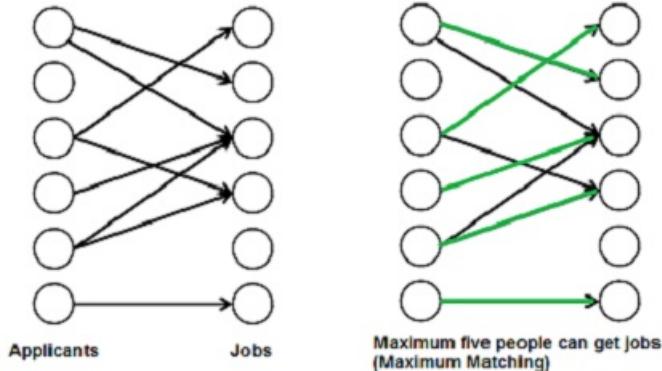
Maximum Bipartite Matching

A matching in a [Bipartite Graph](#) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

Why do we care?

There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:

There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.

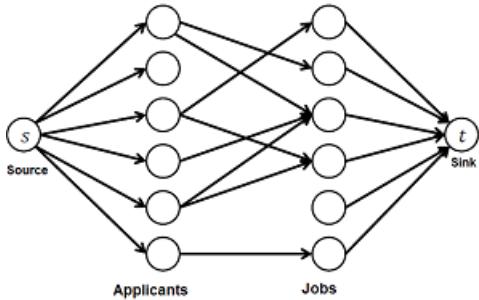


We strongly recommend to read the following post first.

[Ford-Fulkerson Algorithm for Maximum Flow Problem](#)

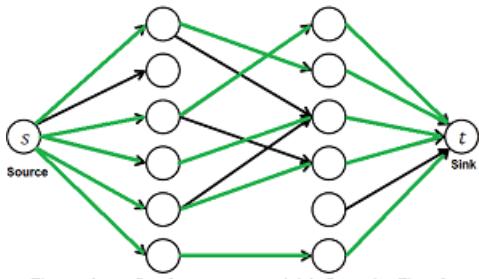
Maximum Bipartite Matching and Max Flow Problem

Maximum Bipartite Matching (MBP) problem can be solved by converting it into a flow network (See [this](#) video to know how did we arrive this conclusion). Following are the steps.



1) Build a Flow Network

There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.



The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

2) Find the maximum flow.

We use [Ford-Fulkerson algorithm](#) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.

How to implement the above approach?

Let us first define input and output forms. Input is in the form of [Edmonds matrix](#) which is a 2D array $\text{bpGraph}[M][N]$ with M rows (for M job applicants) and N columns (for N jobs). The value $\text{bpGraph}[i][j]$ is 1 if ith applicant is interested in jth job, otherwise 0.

Output is number maximum number of people that can get jobs.

A simple way to implement this is to create a matrix that represents [adjacency matrix representation](#) of a directed graph with $M+N+2$ vertices. Call the [fordFulkerson\(\)](#) for the matrix. This implementation requires $O((M+N)*(M+N))$ extra space.

Extra space can be reduced and code can be simplified using the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a job for an applicant (similar to augmenting path in Ford-Fulkerson). We call $\text{bpm}()$ for every applicant, $\text{bpm}()$ is the DFS based function that tries all possibilities to assign a job to the applicant.

In $\text{bpm}()$, we one by one try all jobs that an applicant u is interested in until we find a job, or all jobs are tried without luck. For every job we try, we do following.

If a job is not assigned to anybody, we simply assign it to the applicant and return true. If a job is assigned to somebody else say x, then we recursively check whether x can be assigned some other job. To make sure that x doesn't get the same job again, we mark the job v as seen before we make recursive call for x. If x can get other job, we change the applicant for job v and return true. We use an array $\text{maxR}[0..N-1]$ that stores the applicants assigned to different jobs.

If $\text{bpm}()$ returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in $\text{maxBPM}()$.

C++

```
// A C++ program to find maximal Bipartite matching.
#include <iostream>
#include <string.h>
using namespace std;

// M is number of applicants and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function that returns true if a
// matching for vertex u is possible
bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[])
{
    // Try every job one by one
    for (int v = 0; v < N; v++)
    {
        // If applicant u is interested in job v and v is
        // not visited
        if (bpGraph[u][v] && !seen[v])
        {
            seen[v] = true; // Mark v as visited

            // If job 'v' is not assigned to an applicant OR
            // previously assigned applicant for job v (which is matchR[v])
            // has an alternate job available.
            // Since v is marked as visited in the above line, matchR[v]
            // in the following recursive call will not get job 'v' again
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR))
            {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

// Returns maximum number of matching from M to N
int maxBPM(bool bpGraph[M][N])
{
    // An array to keep track of the applicants assigned to
    // jobs. The value of matchR[i] is the applicant number
    // assigned to job i, the value -1 indicates nobody is
    // assigned.
    int matchR[N];

    // Initially all jobs are available
    memset(matchR, -1, sizeof(matchR));
}
```

```

int result = 0; // Count of jobs assigned to applicants
for (int u = 0; u < M; u++)
{
    // Mark all jobs as not seen for next applicant.
    bool seen[N];
    memset(seen, 0, sizeof(seen));

    // Find if the applicant 'u' can get a job
    if (bpm(bpGraph, u, seen, matchR))
        result++;
}
return result;
}

// Driver program to test above functions
int main()
{
    // Let us create a bpGraph shown in the above example
    bool bpGraph[M][N] = { {0, 1, 1, 0, 0, 0},
                           {1, 0, 0, 1, 0, 0},
                           {0, 0, 1, 0, 0, 0},
                           {0, 0, 1, 1, 0, 0},
                           {0, 0, 0, 0, 0, 0},
                           {0, 0, 0, 0, 0, 1}};

    cout << "Maximum number of applicants that can get job is "
         << maxBPM(bpGraph);

    return 0;
}

```

Java

```

// A Java program to find maximal Bipartite matching.
import java.util.*;
import java.lang.*;
import java.io.*;

class MaxBipartite
{
    // M is number of applicants and N is number of jobs
    static final int M = 6;
    static final int N = 6;

    // A DFS based recursive function that returns true if a
    // matching for vertex u is possible
    boolean bpm(boolean bpGraph[][], int u, boolean seen[],
                int matchR[])
    {
        // Try every job one by one
        for (int v = 0; v < N; v++)
        {
            // If applicant u is interested in job v and v
            // is not visited
            if (bpGraph[u][v] && !seen[v])
            {
                seen[v] = true; // Mark v as visited

                // If job 'v' is not assigned to an applicant OR
                // previously assigned applicant for job v (which
                // is matchR[v]) has an alternate job available.
                // Since v is marked as visited in the above line,
                // matchR[v] in the following recursive call will
                // not get job 'v' again
                if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                           seen, matchR))
                {
                    matchR[v] = u;
                    return true;
                }
            }
        }
        return false;
    }

    // Returns maximum number of matching from M to N
    int maxBPM(boolean bpGraph[][])
    {

```

```

// An array to keep track of the applicants assigned to
// jobs. The value of matchR[i] is the applicant number
// assigned to job i, the value -1 indicates nobody is
// assigned.
int matchR[] = new int[N];

// Initially all jobs are available
for(int i=0; i<N; ++i)
    matchR[i] = -1;

int result = 0; // Count of jobs assigned to applicants
for (int u = 0; u < M; u++)
{
    // Mark all jobs as not seen for next applicant.
    boolean seen[] =new boolean[N] ;
    for(int i=0; i<N; ++i)
        seen[i] = false;

    // Find if the applicant 'u' can get a job
    if (bpm(bpGraph, u, seen, matchR))
        result++;
}
return result;
}

// Driver method
public static void main (String[] args) throws java.lang.Exception
{
    // Let us create a bpGraph shown in the above example
    boolean bpGraph[][] = new boolean[][]{
        {false, true, true, false, false, false},
        {true, false, false, true, false, false},
        {false, false, true, false, false, false},
        {false, false, true, true, false, false},
        {false, false, false, false, false, false},
        {false, false, false, false, false, true}
    };
    MaxBipartite m = new MaxBipartite();
    System.out.println( "Maximum number of applicants that can"+
        " get job is "+m.maxBPM(bpGraph));
}
}

```

Maximum number of applicants that can get job is 5

References:

- http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part5.pdf
- <http://www.youtube.com/watch?v=NlQqmEXuiC8>
- http://en.wikipedia.org/wiki/Maximum_matching
- <http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>
- <http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07NetworkFlowII-22.pdf>
- http://www.ise.ncsu.edu/fangroup/or766.dir/or766_ch7.pdf

Channel Assignment Problem

There are M transmitter and N receiver stations. Given a matrix that keeps track of the number of packets to be transmitted from a given transmitter to a receiver. If the (i, j)-th entry of the matrix is k, it means at that time the station i has k packets for transmission to station j. During a time slot, a transmitter can send only one packet and a receiver can receive only one packet. Find the channel assignments so that maximum number of packets are transferred from transmitters to receivers during the next time slot.

Example:

```
0 2 0  
3 0 1  
2 4 0
```

The above is the input format. We call the above matrix M. Each value $M[i][j]$ represents the number of packets Transmitter i has to send to Receiver j. The output should be:

```
The number of maximum packets sent in the time slot is 3  
T1 -> R2  
T2 -> R3  
T3 -> R1
```

Note that the maximum number of packets that can be transferred in any slot is $\min(M, N)$.

Algorithm:

The channel assignment problem between sender and receiver can be easily transformed into Maximum Bipartite Matching(MBP) problem that can be solved by converting it into a flow network.

Step 1: Build a Flow Network

There must be a source and sink in a flow network. So we add a dummy source and add edges from source to all senders. Similarly, add edges from all receivers to dummy sink. The capacity of all added edges is marked as 1 unit.

Step 2: Find the maximum flow.

We use [Ford-Fulkerson algorithm](#) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the maximum number of packets that can be transmitted without bandwidth interference in a time slot.

Implementation:

Let us first define input and output forms. Input is in the form of Edmonds matrix which is a 2D array table[M][N] with M rows (for M senders) and N columns (for N receivers). The value table[i][j] is the number of packets that has to be sent from transmitter i to receiver j. Output is the maximum number of packets that can be transmitted without bandwidth interference in a time slot.

A simple way to implement this is to create a matrix that represents adjacency matrix representation of a directed graph with $M+N+2$ vertices. Call the fordFulkerson() for the matrix. This implementation requires $O((M+N)*(M+N))$ extra space.

Extra space can be reduced and code can be simplified using the fact that the graph is bipartite. The idea is to use DFS traversal to find a receiver for a transmitter (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a receiver to the sender. In bpm(), we one by one try all receivers that a sender u is interested in until we find a receiver, or all receivers are tried without luck.

For every receiver we try, we do following:

If a receiver is not assigned to anybody, we simply assign it to the sender and return true. If a receiver is assigned to somebody else say x, then we recursively check whether x can be assigned some other receiver. To make sure that x doesn't get the same receiver again, we mark the receiver v as seen before we make recursive call for x. If x can get other receiver, we change the sender for receiver v and return true. We use an array maxR[0..N-1] that stores the senders assigned to different receivers.

If bpm() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

Time and space complexity analysis:

In case of bipartite matching problem, $F \leq |V|$ since there can be only $|V|$ possible edges coming out from source node. So the total running time is $O(mn) = O((m+n)n)$. The space complexity is also substantially reduced from $O((M+N)*(M+N))$ to just a single dimensional array of size M thus storing the mapping between M and N.

```
#include <iostream>  
#include <string.h>  
#include <vector>  
#define M 3  
#define N 4  
using namespace std;  
  
// A Depth First Search based recursive function that returns true  
// if a matching for vertex u is possible  
bool bpm(int table[M][N], int u, bool seen[], int matchR[]){  
    // Try every receiver one by one  
    for (int v = 0; v < N; v++)  
    {  
        // If sender u has packets to send to receiver v and  
        // receiver v is not assigned to anyone  
        if (table[u][v] > 0 && seen[v] == false)  
        {  
            seen[v] = true;  
            if (matchR[v] == -1 || bpm(table, matchR[v], seen, matchR))  
            {  
                matchR[v] = u;  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

```

// receiver v is not already mapped to any other sender
// just check if the number of packets is greater than '0'
// because only one packet can be sent in a time frame anyways
if (table[u][v]>0 && !seen[v])
{
    seen[v] = true; // Mark v as visited

    // If receiver 'v' is not assigned to any sender OR
    // previously assigned sender for receiver v (which is
    // matchR[v]) has an alternate receiver available. Since
    // v is marked as visited in the above line, matchR[v] in
    // the following recursive call will not get receiver 'v' again
    if (matchR[v] < 0 || bpm(table, matchR[v], seen, matchR))
    {
        matchR[v] = u;
        return true;
    }
}
return false;
}

// Returns maximum number of packets that can be sent parallelly in 1
// time slot from sender to receiver
int maxBPM(int table[M][N])
{
    // An array to keep track of the receivers assigned to the senders.
    // The value of matchR[i] is the sender ID assigned to receiver i.
    // the value -1 indicates nobody is assigned.
    int matchR[N];

    // Initially all receivers are not mapped to any senders
    memset(matchR, -1, sizeof(matchR));

    int result = 0; // Count of receivers assigned to senders
    for (int u = 0; u < M; u++)
    {
        // Mark all receivers as not seen for next sender
        bool seen[N];
        memset(seen, 0, sizeof(seen));

        // Find if the sender 'u' can be assigned to the receiver
        if (bpm(table, u, seen, matchR))
            result++;
    }

    cout << "The number of maximum packets sent in the time slot is "
        << result << "\n";
    for (int x=0; x<N; x++)
        if (matchR[x]+1!=0)
            cout << "T" << matchR[x]+1 << "-> R" << x+1 << "\n";
    return result;
}

// Driver program to test above function
int main()
{
    int table[M][N] = {{0, 2, 0}, {3, 0, 1}, {2, 4, 0}};
    int max_flow = maxBPM(table);
    return 0;
}

```

Output:

```

The number of maximum packets sent in the time slot is 3
T3-> R1
T1-> R2
T2-> R3

```

Memory efficient doubly linked list

Asked by Varun Bhatia.

Question:

Write a code for implementation of doubly linked list with use of single pointer in each node.

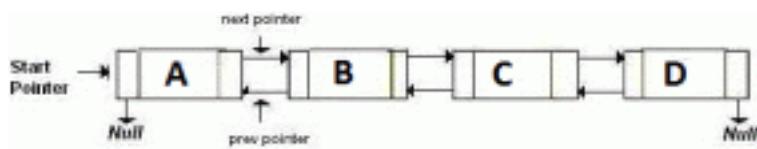
Solution:

This question is solved and very well explained at <http://www.linuxjournal.com/article/6828>.

We also recommend to read http://en.wikipedia.org/wiki/XOR_linked_list

XOR Linked List A Memory Efficient Doubly Linked List | Set 1

An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.



Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Efficient) representations of the Doubly Linked List.

Ordinary Representation:

Node A:

prev = NULL, next = add(B) // previous is NULL and next is address of B

Node B:

prev = add(A), next = add(C) // previous is address of A and next is address of C

Node C:

prev = add(B), next = add(D) // previous is address of B and next is address of D

Node D:

prev = add(C), next = NULL // previous is address of C and next is NULL

XOR List Representation:

Let us call the address variable in XOR representation npx (XOR of next and previous)

Node A:

npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:

npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C

Node C:

npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D

Node D:

npx = add(C) XOR 0 // bitwise XOR of address of C and 0

Traversal of XOR Linked List:

We can traverse the XOR list in both forward and reverse direction. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next nodes address. For example when we are at node C, we must have address of B. XOR of add(B) and npx of C gives us the add(D). The reason is simple: npx(C) is add(B) XOR add(D). If we do xor of npx(C) with add(B), we get the result as add(B) XOR add(D) XOR add(B) which is add(D) XOR 0? which is add(D). So we have the address of next node. Similarly we can traverse the list in backward direction.

We have covered more on XOR Linked List in the following post.

[XOR Linked List A Memory Efficient Doubly Linked List | Set 2](#)

References:

http://en.wikipedia.org/wiki/XOR_linked_list

<http://www.linuxjournal.com/article/6828?page=0,0>

XOR Linked List A Memory Efficient Doubly Linked List | Set 2

In the [previous post](#), we discussed how a Doubly Linked can be created using only one space for address field with every node. In this post, we will discuss implementation of memory efficient doubly linked list. We will mainly discuss following two simple functions.

- 1) A function to insert a new node at the beginning.
- 2) A function to traverse the list in forward direction.

In the following code, *insert()* function inserts a new node at the beginning. We need to change the head pointer of Linked List, that is why a double pointer is used (See [this](#)). Let us first discuss few things again that have been discussed in the [previous post](#). We store XOR of next and previous nodes with every node and we call it npx, which is the only address member we have with every node. When we insert a new node at the beginning, npx of new node will always be XOR of NULL and current head. And npx of current head must be changed to XOR of new node and node next to current head.

printList() traverses the list in forward direction. It prints data values from every node. To traverse the list, we need to get pointer to the next node at every point. We can get the address of next node by keeping track of current node and previous node. If we do XOR of curr->npx and prev, we get the address of next node.

```
/* C/C++ Implementation of Memory efficient Doubly Linked List */
#include <stdio.h>
#include <stdlib.h>

// Node structure of a memory efficient doubly linked list
struct node
{
    int data;
    struct node* npx; /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
struct node* XOR (struct node *a, struct node *b)
{
    return (struct node*) ((unsigned int) (a) ^ (unsigned int) (b));
}

/* Insert a node at the beginning of the XORed linked list and makes the
   newly inserted node as head */
void insert(struct node **head_ref, int data)
{
    // Allocate memory for new node
    struct node *new_node = (struct node *) malloc (sizeof (struct node) );
    new_node->data = data;

    /* Since new node is being inserted at the beginning, npx of new node
       will always be XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of current head node will be XOR
       of new node and node next to current head */
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next. So if we do XOR of
        // it with NULL, we get next
        struct node* next = XOR((*head_ref)->npx, NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked list in forward direction
void printList (struct node *head)
{
    struct node *curr = head;
    struct node *prev = NULL;
    struct node *next;

    printf ("Following are the nodes of Linked List: \n");

    while (curr != NULL)
    {
        // print current node
        printf ("%d ", curr->data);

        // get address of next node: curr->npx is next^prev, so curr->npx^prev
        // will be next^prev^prev which is next
    }
}
```

```
next = XOR (prev, curr->npx);

// update prev and curr for next iteration
prev = curr;
curr = next;
}

// Driver program to test above functions
int main ()
{
    /* Create following Doubly Linked List
     head-->40<-->30<-->20<-->10    */
    struct node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);

    // print the created list
    printList (head);

    return (0);
}
```

Output:

Following are the nodes of Linked List:
40 30 20 10

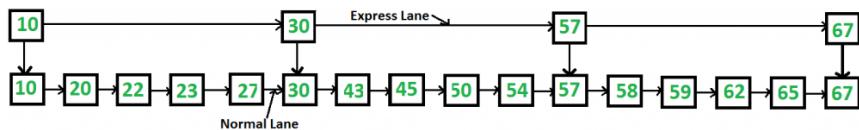
Note that XOR of pointers is not defined by C/C++ standard. So the above implementation may not work on all platforms.

Skip List | Set 1 (Introduction)

Can we search in a sorted linked list in better than O(n) time?

The worst case search time for a sorted linked list is O(n) as we can only linearly traverse the list and cannot skip nodes while searching. For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

Can we augment sorted linked lists to make the search faster? The answer is [Skip List](#). The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an express lane which connects only main outer stations, and the lower layer works as a normal lane which connects every station. Suppose we want to search for 50, we start from first node of express lane and keep moving on express lane till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on express lane, we move to normal lane using pointer from this node, and linearly search for 50 on normal lane. In following example, we start from 30 on normal lane and with linear search, we find 50.



What is the time complexity with two layers? The worst case time complexity is number of nodes on express lane plus number of nodes in a segment (A segment is number of normal lane nodes between two express lane nodes) of normal lane. So if we have n nodes on normal lane, \sqrt{n} nodes on express lane and we equally divide the normal lane, then there will be \sqrt{n} nodes in every segment of normal lane. \sqrt{n} is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be $O(\sqrt{n})$. Therefore, with $O(\sqrt{n})$ extra space, we are able to reduce the time complexity to $O(\sqrt{n})$.

Can we do better?

The time complexity of skip lists can be reduced further by adding more layers. In fact, the time complexity of search, insert and delete can become $O(\log n)$ in average case. We will soon be publishing more posts on Skip Lists.

References

- [MIT Video Lecture on Skip Lists](#)
- http://en.wikipedia.org/wiki/Skip_list

Self Organizing List | Set 1 (Introduction)

The worst case search time for a sorted linked list is $O(n)$. With a Balanced Binary Search Tree, we can skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

One idea to make search faster for Linked Lists is [Skip List](#). Another idea (which is discussed in this post) is to *place more frequently accessed items closer to head..* There can be two possibilities. offline (we know the complete search sequence in advance) and online (we dont know the search sequence).

In case of offline, we can put the nodes according to decreasing frequencies of search (The element having maximum search count is put first). For many practical applications, it may be difficult to obtain search sequence in advance. A [SelfOrganizing list](#) reorders its nodes based on searches which are done. The idea is to use locality of reference (In a typical database, 80% of the access are to 20% of the items). Following are different strategies used by SelfOrganizing Lists.

- 1) **Move-to-Front Method:** Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.
- 2) **Count Method:** Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.
- 3) **Transpose Method:** Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.

[Competitive Analysis:](#)

The worst case time complexity of all methods is $O(n)$. In worst case, the searched element is always the last element in list. For [average case analysis](#), we need probability distribution of search sequences which is not available many times.

For online strategies and algorithms like above, we have a totally different way of analyzing them called *competitive analysis* where performance of an online algorithm is compared to the performance of an optimal offline algorithm (that can view the sequence of requests in advance).

Competitive analysis is used in many practical algorithms like caching, disk paging, high performance computers. The best thing about competitive analysis is, we dont need to assume anything about probability distribution of input. The Move-to-front method is 4-competitive, means it never does more than a factor of 4 operations than offline algorithm (See [the MIT video lecture](#) for proof).

We will soon be discussing implementation and proof of the analysis given in the video lecture.

References:

- http://en.wikipedia.org/wiki/Self-organizing_list
- [MIT Video Lecture](#)
- http://www.eecs.yorku.ca/course_archive/2003-04/F/2011/2011A/DatStr_071_SOLists.pdf
- [http://en.wikipedia.org/wiki/Competitive_analysis_\(online_algorithm\)](http://en.wikipedia.org/wiki/Competitive_analysis_(online_algorithm))

Trie | Insert and Search

[Trie](#) is an efficient information retrieval data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to $M * \log N$, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in O(M) time. However the penalty is on trie storage requirements.

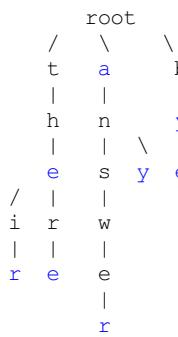
Every node of trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as leaf node. A trie node field *value* will be used to distinguish the node as leaf node (there are other uses of the *value* field). A simple structure to represent nodes of English alphabet can be as following.

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t *children[ALPHABET_SIZE];
};
```

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the *children* is an array of pointers to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the *value* field of last node is non-zero then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type *trie_node_t*, and its children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, *a* at the next level is having only one child (*n*), all other children are NULL. The leaf nodes are in blue.

Insert and search costs **O(key_length)**, however the memory requirements of trie is **O(ALPHABET_SIZE * key_length * N)** where N is number of keys in trie. There are efficient representation of trie nodes (e.g. compressed trie, [ternary search tree](#), etc.) to minimize memory requirements of trie.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)
// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
typedef struct trie_node trie_node_t;
struct trie_node
{
    int value;
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;
struct trie
{
    trie_node_t *root;
    int count;
};
```

```

// Returns new trie node (initialized to NULLs)
trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

// Initializes trie (root is dummy node)
void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = CHAR_TO_INDEX(key[level]);
        if( !pCrawl->children[index] )
        {
            pCrawl->children[index] = getNode();
        }

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->value = pTrie->count;
}

// Returns non zero, if key presents in trie
int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = CHAR_TO_INDEX(key[level]);

        if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}

```

```
// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[][8] = {"the", "a", "there", "answer", "any", "by", "bye", "their"};
    trie_t trie;

    char output[][32] = {"Not present in trie", "Present in trie"};

    initialize(&trie);

    // Construct trie
    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    // Search for different keys
    printf("%s --- %s\n", "the", output[search(&trie, "the")] );
    printf("%s --- %s\n", "these", output[search(&trie, "these")] );
    printf("%s --- %s\n", "their", output[search(&trie, "their")] );
    printf("%s --- %s\n", "thaw", output[search(&trie, "thaw")] );

    return 0;
}
```

Trie | (Delete)

In the [previous post](#) on [trie](#) we have described how to insert and search a node in trie. Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

The highlighted code presents algorithm to implement above conditions. (One may be in dilemma how a pointer passed to delete helper is reflecting changes from deleteHelper to deleteKey. Note that we are holding trie as an ADT in `trie_t` node, which is passed by reference or pointer).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)

#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')

#define FREE(p) \
    free(p); \
    p = NULL;

// forward declaration
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
    int value; // non zero if leaf
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
    trie_node_t *root;
    int count;
};

trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
```

```

int level;
int length = strlen(key);
int index;
trie_node_t *pCrawl;

pTrie->count++;
pCrawl = pTrie->root;

for( level = 0; level < length; level++ )
{
    index = INDEX(key[level]);

    if( pCrawl->children[index] )
    {
        // Skip current node
        pCrawl = pCrawl->children[index];
    }
    else
    {
        // Add new node
        pCrawl->children[index] = getNode();
        pCrawl = pCrawl->children[index];
    }
}

// mark last node as leaf (non zero)
pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}

int leafNode(trie_node_t *pNode)
{
    return (pNode->value != 0);
}

int isItFreeNode(trie_node_t *pNode)
{
    int i;
    for(i = 0; i < ALPHABET_SIZE; i++)
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
            {
                // Unmark leaf node
                pNode->value = 0;
            }
        }
    }
}

```

```

        // If empty, node to be deleted
        if( isItFreeNode(pNode) )
        {
            return true;
        }

        return false;
    }
}

else // Recursive case
{
    int index = INDEX(key[level]);

    if( deleteHelper(pNode->children[index], key, level+1, len) )
    {
        // last node marked, delete it
        FREE(pNode->children[index]);

        // recursively climb up, and delete eligible nodes
        return ( !leafNode(pNode) && isItFreeNode(pNode) );
    }
}

return false;
}

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}

int main()
{
    char keys[][8] = {"she", "sells", "sea", "shore", "the", "by", "sheer"};
    trie_t trie;

    initialize(&trie);

    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    deleteKey(&trie, keys[0]);

    printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie" : "Not present in trie");
    return 0;
}

```

Longest prefix matching A Trie based solution in Java

Given a dictionary of words and an input string, find the longest prefix of the string which is also a word in dictionary.

Examples:

Let the dictionary contains the following words:
{are, area, base, cat, cater, children, basement}

Below are some input/output examples:

| Input String | Output |
|--------------|-----------|
| caterer | cater |
| basemexy | base |
| child | < Empty > |

Solution

We build a Trie of all dictionary words. Once the Trie is built, traverse through it using characters of input string. If prefix matches a dictionary word, store current length and look for a longer match. Finally, return the longest match.

Following is Java implementation of the above solution based.

```
import java.util.HashMap;

// Trie Node, which stores a character and the children in a HashMap
class TrieNode {
    public TrieNode(char ch) {
        value = ch;
        children = new HashMap<>();
        bIsEnd = false;
    }
    public HashMap<Character, TrieNode> getChildren() { return children; }
    public char getValue() { return value; }
    public void setIsEnd(boolean val) { bIsEnd = val; }
    public boolean isEnd() { return bIsEnd; }

    private char value;
    private HashMap<Character, TrieNode> children;
    private boolean bIsEnd;
}

// Implements the actual Trie
class Trie {
    // Constructor
    public Trie() { root = new TrieNode((char)0); }

    // Method to insert a new word to Trie
    public void insert(String word) {

        // Find length of the given word
        int length = word.length();
        TrieNode crawl = root;

        // Traverse through all characters of given word
        for( int level = 0; level < length; level++ )
        {
            HashMap<Character, TrieNode> child = crawl.getChildren();
            char ch = word.charAt(level);

            // If there is already a child for current character of given word
            if( child.containsKey(ch) )
                crawl = child.get(ch);
            else // Else create a child
            {
                TrieNode temp = new TrieNode(ch);
                child.put( ch, temp );
                crawl = temp;
            }
        }

        // Set bIsEnd true for last character
        crawl.setIsEnd(true);
    }

    // The main method that finds out the longest string 'input'
    public String getMatchingPrefix(String input) {
        String result = ""; // Initialize resultant string
        int length = input.length(); // Find length of the input string
```

```

// Initialize reference to traverse through Trie
TrieNode crawl = root;

// Iterate through all characters of input string 'str' and traverse
// down the Trie
int level, prevMatch = 0;
for( level = 0 ; level < length; level++ )
{
    // Find current character of str
    char ch = input.charAt(level);

    // HashMap of current Trie node to traverse down
    HashMap<Character,TrieNode> child = crawl.getChildren();

    // See if there is a Trie edge for the current character
    if( child.containsKey(ch) )
    {
        result += ch;           //Update result
        crawl = child.get(ch); //Update crawl to move down in Trie

        // If this is end of a word, then update prevMatch
        if( crawl.isEnd() )
            prevMatch = level + 1;
    }
    else break;
}

// If the last processed character did not match end of a word,
// return the previously matching prefix
if( !crawl.isEnd() )
    return result.substring(0, prevMatch);

else return result;
}

private TrieNode root;
}

// Testing class
public class Test {
    public static void main(String[] args) {
        Trie dict = new Trie();
        dict.insert("are");
        dict.insert("area");
        dict.insert("base");
        dict.insert("cat");
        dict.insert("cater");
        dict.insert("basement");

        String input = "caterer";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basement";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "are";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "arex";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basemexz";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "xyz";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));
    }
}

```

Output:

```

caterer:  cater
basement:  basement
are:  are
arex:  are

```

```
basemexz:    base  
xyz:
```

Time Complexity: Time complexity of finding the longest prefix is $O(n)$ where n is length of the input string. Refer [this](#) for time complexity of building the Trie.

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

Input:

```
{0, 1, 0, 0, 1}
{1, 0, 1, 1, 0}
{0, 1, 0, 0, 1}
{1, 1, 1, 0, 0}
```

Output:

```
0 1 0 0 1
1 0 1 1 0
1 1 1 0 0
```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, dont print it. If the current row doesnt match with any row, print it.

Time complexity: O(ROW^2 x COL)

Auxiliary Space: O(1)

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: O(ROW x COL + ROW x log(ROW))

Auxiliary Space: O(ROW)

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, dont print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
//Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert ( &( (*root)->child[ M[row][col] ] ), M, row, col+1 );

    else // If all entries of this row are processed
```

```

    {
        // unique row found, return 1
        if ( !( *root)->isEndOfCol ) )
            return (*root)->isEndOfCol = 1;

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow( int (*M) [COL], int row )
{
    int i;
    for( i = 0; i < COL; ++i )
        printf( "%d ", M[row][i] );
    printf("\n");
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows( int (*M) [COL] )
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for ( i = 0; i < ROW; ++i )
        // insert row to TRIE
        if ( insert(&root, M, i, 0) )
            // unique row found, print it
            printRow( M, i );
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{0, 1, 0, 0, 1},
                      {1, 0, 1, 1, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 1, 0, 0}
    };

    findUniqueRows( M );

    return 0;
}

```

Time complexity: O(ROW x COL)

Auxiliary Space: O(ROW x COL)

This method has better time complexity. Also, relative order of rows is maintained while printing.

How to Implement Reverse DNS Look Up Cache?

Reverse DNS look up is using an internet IP address to find a domain name. For example, if you type 74.125.200.106 in browser, it automatically redirects to google.in.

How to implement Reverse DNS Look Up cache? Following are the operations needed from cache.

- 1) Add a IP address to URL Mapping in cache.
- 2) Find URL for a given IP address.

One solution is to use [Hashing](#).

In this post, a [Trie](#) based solution is discussed. One advantage of Trie based solutions is, worst case upper bound is $O(1)$ for Trie, for hashing, the best possible average case time complexity is $O(1)$. Also, with Trie we can implement prefix search (finding all urls for a common prefix of IP addresses).

The general disadvantage of Trie is large amount of memory requirement, this is not a major problem here as the alphabet size is only 11 here. Ten characters are needed for digits from 0 to 9 and one for dot (.) .

The idea is to store IP addresses in Trie nodes and in the last node we store the corresponding domain name. Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 11 different chars in a valid IP address
#define CHARS 11

// Maximum length of a valid IP address
#define MAX 50

// A utility function to find index of child for a given character 'c'
int getIndex(char c) { return (c == '.')? 10: (c - '0'); }

// A utility function to find character for a given child index.
char getCharFromIndex(int i) { return (i== 10)? '.' : ('0' + i); }

// Trie Node.
struct trienode
{
    bool isLeaf;
    char *URL;
    struct trienode *child[CHARS];
};

// Function to create a new trie node.
struct trienode *newTrienode(void)
{
    struct trienode *newNode = new trienode;
    newNode->isLeaf = false;
    newNode->URL = NULL;
    for (int i=0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts an ip address and the corresponding
// domain name in the trie. The last node in Trie contains the URL.
void insert(struct trienode *root, char *ipAdd, char *URL)
{
    // Length of the ip address
    int len = strlen(ipAdd);
    struct trienode *pCrawl = root;

    // Traversing over the length of the ip address.
    for (int level=0; level<len; level++)
    {
        // Get index of child node from current character
        // in ipAdd[]. Index must be from 0 to 10 where
        // 0 to 9 is used for digits and 10 for dot
        int index = getIndex(ipAdd[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrienode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }
}
```

```

}

//Below needs to be carried out for the last node.
//Save the corresponding URL of the ip address in the
//last node of trie.
pCrawl->isLeaf = true;
pCrawl->URL = new char[strlen(URL) + 1];
strcpy(pCrawl->URL, URL);
}

// This function returns URL if given IP address is present in DNS cache.
// Else returns NULL
char *searchDNSCache(struct trieNode *root, char *ipAdd)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(ipAdd);

    // Traversal over the length of ip address.
    for (int level=0; level<len; level++)
    {
        int index = getIndex(ipAdd[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address, print the URL.
    if (pCrawl!=NULL && pCrawl->isLeaf)
        return pCrawl->URL;

    return NULL;
}

//Driver function.
int main()
{
    /* Change third ipAddress for validation */
    char ipAdd[][MAX] = {"107.108.11.123", "107.109.123.255",
                         "74.125.200.106"};
    char URL[] [50] = {"www.samsung.com", "www.samsung.net",
                       "www.google.in"};
    int n = sizeof(ipAdd)/sizeof(ipAdd[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the ip address and their corresponding
    // domain name after ip address validation.
    for (int i=0; i<n; i++)
        insert(root,ipAdd[i],URL[i]);

    // If reverse DNS look up succeeds print the domain
    // name along with DNS resolved.
    char ip[] = "107.108.11.123";
    char *res_url = searchDNSCache(root, ip);
    if (res_url != NULL)
        printf("Reverse DNS look up resolved in cache:\n%s --> %s",
               ip, res_url);
    else
        printf("Reverse DNS look up not resolved in cache ");
    return 0;
}

```

Output:

```

Reverse DNS look up resolved in cache:
107.108.11.123 --> www.samsung.com

```

Note that the above implementation of Trie assumes that the given IP address does not contain characters other than {0?, 1.. 9, .}. What if a user gives an invalid IP address that contains some other characters? This problem can be resolved by [validating the input IP address](#) before inserting it into Trie. We can use the approach discussed [here](#) for IP address validation.

How to Implement Forward DNS Look Up Cache?

We have discussed [implementation of Reverse DNS Look Up Cache](#). Forward DNS look up is getting IP address for a given domain name typed in the web browser.

The cache should do the following operations :

1. Add a mapping from URL to IP address
2. Find IP address for a given URL.

There are a few changes from [reverse DNS look up cache](#) that we need to incorporate.

1. Instead of [0-9] and (.) dot we need to take care of [A-Z], [a-z] and (.) dot. As most of the domain name contains only lowercase characters we can assume that there will be [a-z] and (.) 27 children for each trie node.

2. When we type www.google.in and google.in the browser takes us to the same page. So, we need to add a domain name into trie for the words after www(.). Similarly while searching for a domain name corresponding IP address remove the www(.) if the user has provided it.

This is left as an exercise and for simplicity we have taken care of www. also.

One solution is to use [Hashing](#). In this post, a [Trie](#) based solution is discussed. One advantage of Trie based solutions is, worst case upper bound is O(1) for Trie, for hashing, the best possible average case time complexity is O(1). Also, with Trie we can implement prefix search (finding all IPs for a common prefix of URLs). The general disadvantage of Trie is large amount of memory requirement.

The idea is to store URLs in Trie nodes and store the corresponding IP address in last or leaf node.

Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 27 different chars in a valid URL
// assuming URL consists [a-z] and (.)
#define CHARS 27

// Maximum length of a valid URL
#define MAX 100

// A utility function to find index of child for a given character 'c'
int getIndex(char c)
{
    return (c == '.') ? 26 : (c - 'a');
}

// A utility function to find character for a given child index.
char getCharFromIndex(int i)
{
    return (i == 26) ? '.' : ('a' + i);
}

// Trie Node.
struct trienode
{
    bool isLeaf;
    char *ipAdd;
    struct trienode *child[CHARS];
};

// Function to create a new trie node.
struct trienode *newTrienode(void)
{
    struct trienode *newNode = new trienode;
    newNode->isLeaf = false;
    newNode->ipAdd = NULL;
    for (int i = 0; i < CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts a URL and corresponding IP address
// in the trie. The last node in Trie contains the ip address.
void insert(struct trienode *root, char *URL, char *ipAdd)
{
    // Length of the URL
    int len = strlen(URL);
    struct trienode *pCrawl = root;
```

```

// Traversing over the length of the URL.
for (int level = 0; level<len; level++)
{
    // Get index of child node from current character
    // in URL[] Index must be from 0 to 26 where
    // 0 to 25 is used for alphabets and 26 for dot
    int index = getIndex(URL[level]);

    // Create a new child if not exist already
    if (!pCrawl->child[index])
        pCrawl->child[index] = newTrieNode();

    // Move to the child
    pCrawl = pCrawl->child[index];
}

//Below needs to be carried out for the last node.
//Save the corresponding ip address of the URL in the
//last node of trie.
pCrawl->isLeaf = true;
pCrawl->ipAdd = new char[strlen(ipAdd) + 1];
strcpy(pCrawl->ipAdd, ipAdd);
}

// This function returns IP address if given URL is
// present in DNS cache. Else returns NULL
char *searchDNSCache(struct trieNode *root, char *URL)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(URL);

    // Traversal over the length of URL.
    for (int level = 0; level<len; level++)
    {
        int index = getIndex(URL[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address,
    // print the ip address.
    if (pCrawl != NULL && pCrawl->isLeaf)
        return pCrawl->ipAdd;

    return NULL;
}

// Driver function.
int main()
{
    char URL[][50] = { "www.samsung.com", "www.samsung.net",
                       "www.google.in"
                     };
    char ipAdd[][MAX] = { "107.108.11.123", "107.109.123.255",
                          "74.125.200.106"
                        };
    int n = sizeof(URL) / sizeof(URL[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the domain name and their corresponding
    // ip address
    for (int i = 0; i<n; i++)
        insert(root, URL[i], ipAdd[i]);

    // If forward DNS look up succeeds print the url along
    // with the resolved ip address.
    char url[] = "www.samsung.com";
    char *res_ip = searchDNSCache(root, url);
    if (res_ip != NULL)
        printf("Forward DNS look up resolved in cache:\n%s --> %s",
               url, res_ip);
    else
        printf("Forward DNS look up not resolved in cache ");

    return 0;
}

```

Output:

Forward DNS look up resolved in cache:
www.samsung.com --> 107.108.11.123

Suffix Array | Set 1 (Introduction)

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

[Pattern Searching | Set 8 \(Suffix Tree Introduction\)](#)

A **suffix array** is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree which is compressed trie of all suffixes of the given text](#). Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source [Wiki](#)).

A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.

Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: [Wiki](#))

Example:

Let the given string be "banana".

| | | |
|----------|-------------------|----------|
| 0 banana | 5 a | |
| 1 anana | Sort the Suffixes | 3 ana |
| 2 nana | -----> | 1 anana |
| 3 ana | alphabetically | 0 banana |
| 4 na | | 4 na |
| 5 a | | 2 nana |

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}

Naive method to build Suffix Array

A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return suffixArr;
}
```

```

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}

```

Output:

```

Following is suffix array for banana
5 3 1 0 4 2

```

The time complexity of above method to build suffix array is $O(n^2 \log n)$ if we consider a $O(n \log n)$ algorithm used for sorting. The sorting step itself takes $O(n^2 \log n)$ time as every comparison is a comparison of two strings and the comparison takes $O(n)$ time. There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, [Binary Search](#) can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only reports one of them.

```

// This code only contains search() and main. To make it a complete running
// above code or see http://ideone.com/lIo9eN

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat); // get length of pattern, needed for strncmp()

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1; // Initialize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strncmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabetically less than
        // the mid suffix
        if (res < 0) r = mid - 1;

        // Otherwise move to right half
        else l = mid + 1;
    }

    // We reach here if return statement in loop is not executed
    cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana"; // text
    char pat[] = "nan"; // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

```

```
// search pat in txt using the built suffix array  
search(pat, txt, suffArr, n);  
  
return 0;  
}
```

Output:

Pattern found at index 2

The time complexity of the above search function is $O(m\log n)$. There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a $O(m)$ suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array

Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.

- 1) Pattern Searching
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

See [this](#) for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed [a \$O\(n\log n\)\$ algorithm for Suffix Array construction here](#). We will soon be discussing more efficient suffix array algorithms.

References:

- <http://www.stanford.edu/class/cs97si/suffix-array.pdf>
- http://en.wikipedia.org/wiki/Suffix_array

Suffix Array | Set 2 (nLogn Algorithm)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree](#) which is compressed trie of all suffixes of the given text.

Let the given string be "banana".

| | | | |
|---|--------|-------------------|----------|
| 0 | banana | 5 | a |
| 1 | anana | Sort the Suffixes | 3 ana |
| 2 | nana | -----> | 1 anana |
| 3 | ana | alphabetically | 0 banana |
| 4 | na | | 4 na |
| 5 | a | | 2 nana |

The suffix array for "banana" is {5, 3, 1, 0, 4, 2}

We have discussed [Naive algorithm for construction of suffix array](#). The Naive algorithm is to consider all suffixes, sort them using a O(nLogn) sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is O($n^2 \log n$) where n is the number of characters in the input string.

In this post, a O(nLogn) algorithm for suffix array construction is discussed. Let us first discuss a O($n * \log n * \log n$) algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.

We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than $2n$. The important point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in O(nLogn) time using a nLogn sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in O(1) time (we need to compare only two values, see the below example and code).

The sort function is called O(Logn) times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes O(nLognLogn). See <http://www.stanford.edu/class/cs97si/suffix-array.pdf> for more details.

Let us build suffix array the example string banana using above algorithm.

Sort according to first two characters Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do str[i] - 'a' for ith suffix of str[]

| Index | Suffix | Rank |
|-------|--------|------|
| 0 | banana | 1 |
| 1 | anana | 0 |
| 2 | nana | 13 |
| 3 | ana | 0 |
| 4 | na | 13 |
| 5 | a | 0 |

For every character, we also store rank of next adjacent character, i.e., the rank of character at str[i + 1] (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 0 | banana | 1 | 0 |
| 1 | anana | 0 | 13 |
| 2 | nana | 13 | 0 |
| 3 | ana | 0 | 13 |
| 4 | na | 13 | 0 |
| 5 | a | 0 | -1 |

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5 | a | 0 | -1 |
| 1 | anana | 0 | 13 |
| 3 | ana | 0 | 13 |
| 0 | banana | 1 | 0 |
| 2 | nana | 13 | 0 |
| 4 | na | 13 | 0 |

Sort according to first four character

Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0 as new rank to first suffix. For assigning ranks to remaining suffixes, we consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

| Index | Suffix | Rank | Notes |
|-------|--------|------|------------------------------------|
| 5 | a | 0 | [Assign 0 to first] |
| 1 | anana | 1 | (0, 13) is different from previous |
| 3 | ana | 1 | (0, 13) is same as previous |
| 0 | banana | 2 | (1, 0) is different from previous |
| 2 | nana | 3 | (13, 0) is different from previous |

4 na 3 (13, 0) is same as previous

For every suffix str[i], also store rank of next suffix at str[i + 2]. If there is no next suffix at i + 2, we store next rank as -1

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5 | a | 0 | -1 |
| 1 | anana | 1 | 1 |
| 3 | ana | 1 | 0 |
| 0 | banana | 2 | 3 |
| 2 | nana | 3 | 3 |
| 4 | na | 3 | -1 |

Sort all Suffixes according to rank and next rank.

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5 | a | 0 | -1 |
| 3 | ana | 1 | 0 |
| 1 | anana | 1 | 1 |
| 0 | banana | 2 | 3 |
| 4 | na | 3 | -1 |
| 2 | nana | 3 | 3 |

```
// C++ program for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
        (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according to first
    // 2 characters. Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n]; // This array is needed to get the index in suffixes[]
                 // from original index. This mapping is needed to get
                 // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
        {
            if (suffixes[i].rank[0] == prev_rank)
                suffixes[i].rank[0] = rank;
            else
                suffixes[i].rank[0]++;
            ind[suffixes[i].index] = i;
            prev_rank = suffixes[i].rank[0];
        }
    }
}
```

```

// If first rank and next ranks are same as that of previous
// suffix in array, assign the same new rank to this suffix
if (suffixes[i].rank[0] == prev_rank &&
    suffixes[i].rank[1] == suffixes[i-1].rank[1])
{
    prev_rank = suffixes[i].rank[0];
    suffixes[i].rank[0] = rank;
}
else // Otherwise increment rank and assign
{
    prev_rank = suffixes[i].rank[0];
    suffixes[i].rank[0] = ++rank;
}
ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
                           suffixes[ind[nextindex]].rank[0]: -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
int *suffixArr = new int[n];
for (int i = 0; i < n; i++)
    suffixArr[i] = suffixes[i].index;

// Return the suffix array
return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}

```

Output:

Following is suffix array for banana
5 3 1 0 4 2

Note that the above algorithm uses standard sort function and therefore time complexity is $O(n \log n)$. We can use [Radix Sort](#) here to reduce the time complexity to $O(n \log n)$.

Please note that suffix arrays can be constructed in $O(n)$ time also. We will soon be discussing $O(n)$ algorithms.

References:

- <http://www.stanford.edu/class/cs97si/suffix-array.pdf>
- <http://www.cbcn.umd.edu/confcour/Fall2012/lec14b.pdf>

Pattern Searching | Set 8 (Suffix Tree Introduction)

Given a text $\text{txt}[0..n-1]$ and a pattern $\text{pat}[0..m-1]$, write a function $\text{search}(\text{char pat[], char txt[]})$ that prints all occurrences of $\text{pat}[]$ in $\text{txt}[]$. You may assume that $n > m$.

Preprocess Pattern or Preprocess Text?

We have discussed the following algorithms in the previous posts:

[KMP Algorithm](#)

[Rabin Karp Algorithm](#)

[Finite Automata based Algorithm](#)

[Boyer Moore Algorithm](#)

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is $O(n)$ where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in $O(m)$ time where m is length of the pattern.

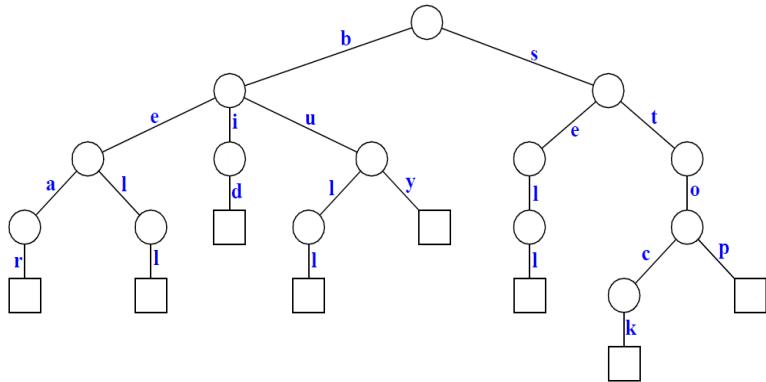
Imagine you have stored complete work of [William Shakespeare](#) and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.

Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

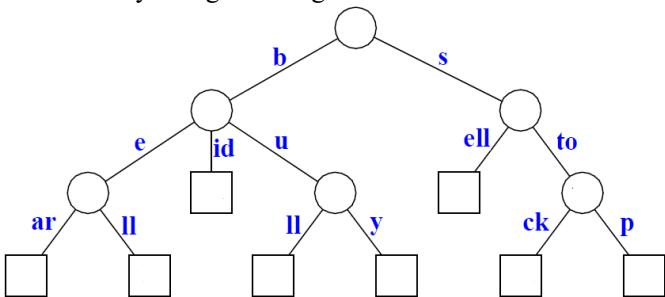
A Suffix Tree for a given text is a compressed trie for all suffixes of the given text. We have discussed [Standard Trie](#). Let us understand Compressed Trie with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.



How to build a Suffix Tree for a given text?

As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.

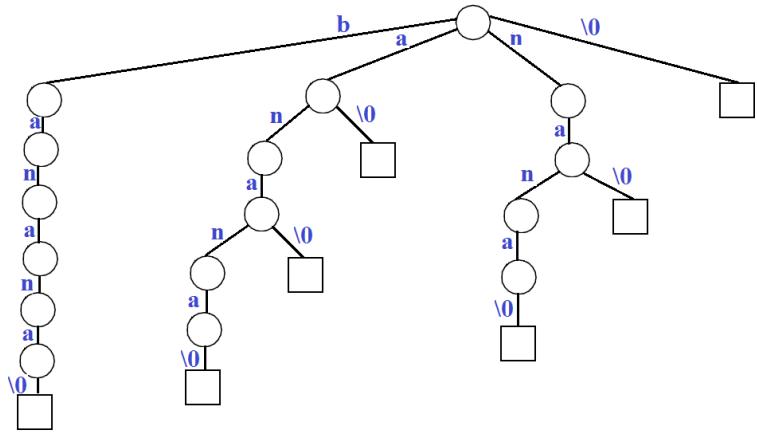
1) Generate all suffixes of given text.

2) Consider all suffixes as individual words and build a compressed trie.

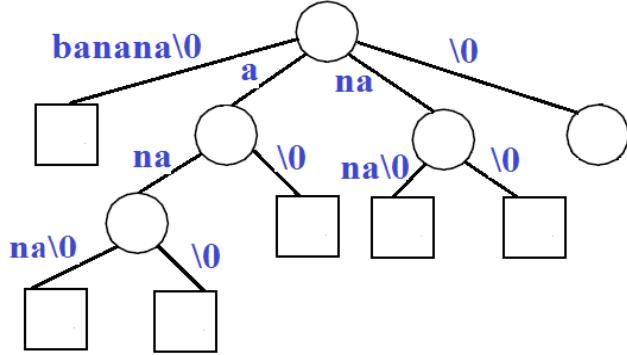
Let us consider an example text $\text{banana}\backslash 0$? where $\backslash 0$? is string termination character. Following are all suffixes of $\text{banana}\backslash 0$?

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text banana\b0?



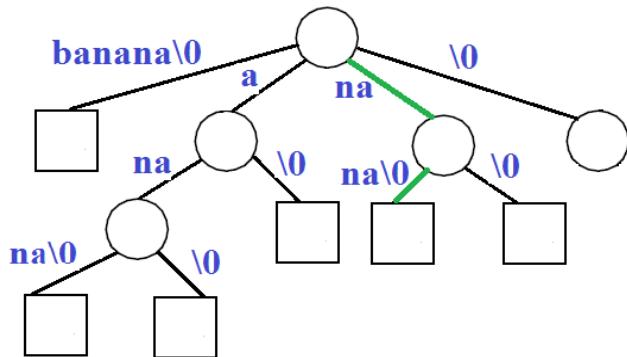
Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?

We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

- 1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
 - ..a) For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.
 - ..b) If there is no edge, print pattern doesn't exist in text and return.
- 2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print Pattern found.

Let us consider the example pattern as nan to see the searching process. Following diagram shows the path followed for searching nan or nana.



How does this work?

Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree

Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.

- 1) Pattern Searching
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

There are many more applications. See [this](#) for more details.

Ukkonens Suffix Tree Construction is discussed in following articles:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

[Ukkonens Suffix Tree Construction Part 3](#)

[Ukkonens Suffix Tree Construction Part 4](#)

[Ukkonens Suffix Tree Construction Part 5](#)

[Ukkonens Suffix Tree Construction Part 6](#)

References:

<http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf>

<http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf>

<http://www.allisons.org/ll/AlgDS/Tree/Suffix/>

Ukkonen's Suffix Tree Construction Part 1

Suffix Tree is very useful in numerous string processing and computational biology problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and its not easy to implement code to construct suffix tree and its usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.

Note: You may find some portion of the algorithm difficult to understand while 1st or 2nd reading and its perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book [Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology](#) by **Dan Gusfield** explains the concepts very well.

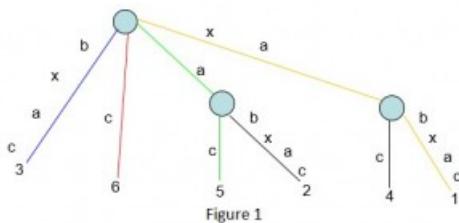
A suffix tree **T** for a m-character string **S** is a rooted directed tree with exactly m leaves numbered 1 to **m**. (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of **S**.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf **i** gives the suffix of **S** that starts at position **i**, i.e. **S[i:m]**.

Note: Position starts with 1 (its not zero indexed, but later, while code implementation, we will used zero indexed position)

For string **S** = **xabxac** with **m** = 6, suffix tree will look like following:



It has one root node and two internal nodes and 6 leaf nodes.

String Depth of red path is 1 and it represents suffix **c** starting at position 6

String Depth of blue path is 4 and it represents suffix **bxca** starting at position 3

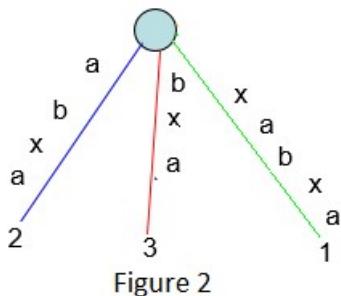
String Depth of green path is 2 and it represents suffix **ac** starting at position 5

String Depth of orange path is 6 and it represents suffix **xabxac** starting at position 1

Edges with labels a (green) and xa (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of **S** matches a prefix of another suffix of **S** (when last character is not unique in string), then path for the first suffix would not end at a leaf.

For String **S** = **xabxa**, with **m** = 5, following is the suffix tree:



Here we will have 5 suffixes: **xabxa**, **abxa**, **bxa**, **xa** and **a**.

Path for suffixes **xa** and **a** do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes (**xa** and **a**) are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use \$, # etc as termination characters.

Following is the suffix tree for string **S** = **xabxa\$** with **m** = 6 and now all 6 suffixes end at leaf.

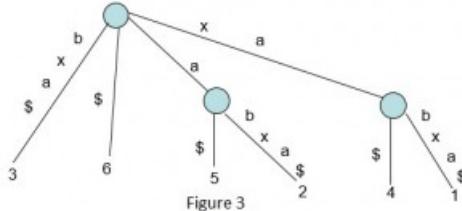


Figure 3

A naive algorithm to build a suffix tree

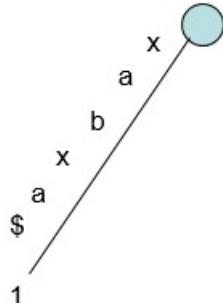
Given a string S of length m , enter a single edge for suffix $S[1..m]\$$ (the entire string) into the tree, then successively enter suffix $S[i..m]\$$ into the growing tree, for i increasing from 2 to m . Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i .

So N_{i+1} is constructed from N_i as follows:

- Start at the root of N_i
- Find the longest path from the root which matches a prefix of $S[i+1..m]\$$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v) , break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in $S[i+1..m]$ and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with $S[i+1..m]$, and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge $(w, i+1)$ from w to a new leaf labelled $i+1$ and it labels the new edge with the unmatched part of suffix $S[i+1..m]$

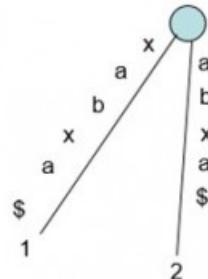
This takes $O(m^2)$ to build the suffix tree for the string S of length m .

Following are few steps to build suffix tree based for string $xabxa\$$ based on above algorithm:



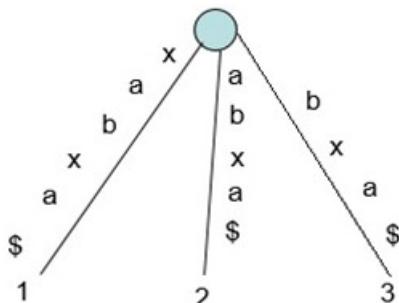
Tree with suffix $N_1, S[1..6]$

Figure 4



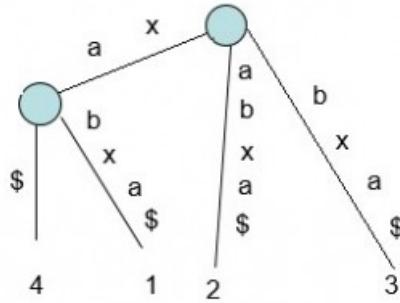
Tree with suffixes $N_1, S[1..6]$ and $N_2, S[2..6]$

Figure 5



Tree with suffixes N_1, N_2 and N_3

Figure 6



Tree with suffixes N1, N2, N3 and N4

Figure 7

Implicit suffix tree

While generating suffix tree using Ukkonens algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with \$ (or # or any other termination character) label and no internal node with only one edge going out of it.

To get implicit suffix tree from a suffix tree SS,

- Remove all terminal symbol \$ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

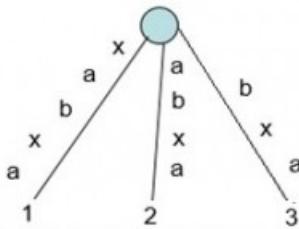


Figure 8: Implicit suffix tree for string xabxa

Sufix tree is shown in Figure 3

High Level Description of Ukkonens algorithm

Ukkonens algorithm constructs an implicit suffix tree T_i for each prefix $S[1..i]$ of S (of length m).

It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, , T_m using mth character.

Implicit suffix tree T_{i+1} is built on top of implicit suffix tree T_i .

The true suffix tree for S is built from T_m by adding \$.

At any time, Ukkonens algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.

Time taken is $O(m)$.

Ukkonens algorithm is divided into m phases (one phase for each character in the string with length m)

In phase $i+1$, tree T_{i+1} is built from tree T_i .

Each phase $i+1$ is further divided into $i+1$ extensions, one for each of the $i+1$ suffixes of $S[1..i+1]$

In extension j of phase $i+1$, the algorithm first finds the end of the path from the root labelled with substring $S[j..i]$.

It then extends the substring by adding the character $S(i+1)$ to its end (if it is not there already).

In extension 1 of phase $i+1$, we put string $S[1..i+1]$ in the tree. Here $S[1..i]$ will already be present in tree due to previous phase i. We just need to add $S[i+1]$ th character in tree (if not there already).

In extension 2 of phase $i+1$, we put string $S[2..i+1]$ in the tree. Here $S[2..i]$ will already be present in tree due to previous phase i. We just need to add $S[i+1]$ th character in tree (if not there already)

In extension 3 of phase $i+1$, we put string $S[3..i+1]$ in the tree. Here $S[3..i]$ will already be present in tree due to previous phase i. We just need to add $S[i+1]$ th character in tree (if not there already)

.

.

In extension $i+1$ of phase $i+1$, we put string $S[i+1..i+1]$ in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label $S[i+1]$.

High Level Ukkonens algorithm

Construct tree T_1

For i from 1 to m-1 do

begin {phase $i+1$ }

For j from 1 to $i+1$

```
begin {extension j}
```

Find the end of the path from the root labelled $S[j..i]$ in the current tree.

Extend that path by adding character $S[i+1]$ if it is not there already

end;

end;

Suffix extension is all about adding the next character into the suffix tree built so far.

In extension j of phase $i+1$, algorithm finds the end of $S[j..i]$ (which is already in the tree due to previous phase i) and then it extends $S[j..i]$ to be sure the suffix $S[j..i+1]$ is in the tree.

There are 3 extension rules:

Rule 1: If the path from the root labelled $S[j..i]$ ends at leaf edge (i.e. $S[i]$ is last character on leaf edge) then character $S[i+1]$ is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled $S[j..i]$ ends at non-leaf edge (i.e. there are more characters after $S[i]$ on path) and next character is not $s[i+1]$, then a new leaf edge with label $s[i+1]$ and number j is created starting from character $S[i+1]$.

A new internal node will also be created if $s[1..i]$ ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled $S[j..i]$ ends at non-leaf edge (i.e. there are more characters after $S[i]$ on path) and next character is $s[i+1]$ (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string xabxac using Ukkonens algorithm

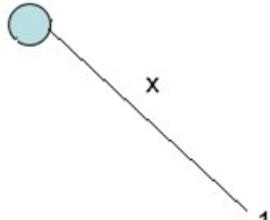


Figure 9: T1 for $S[1..1]$
Adding suffixes of x (x)
Rule 2 - A new leaf edge

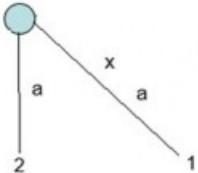


Figure 10: T2 for $S[1..2]$
Adding suffixes of xa (xa and a)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - A new leaf edge

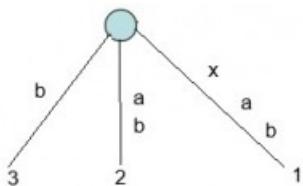


Figure 11: T3 for $S[1..3]$
Adding suffixes of xab (xab , ab and b)
Rule 1 - Extending path label in existing leaf edge
Rule 2 - A new leaf edge

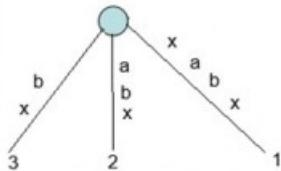


Figure 12: T4 for S[1..4]
 Adding suffixes of xabx (xabx, abx, bx and x)
 Rule 1 - Extending path label in existing leaf edge
 Rule 3: Do nothing (path with label x already present)

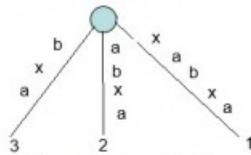


Figure 13: T5 for S[1..5]
 Adding suffixes of xabxa (xabxa, abxa, bxa, xa and x)
 Rule 1 - Extending path label in existing leaf edge
 Rule 3: Do nothing (path with label xa and a already present)

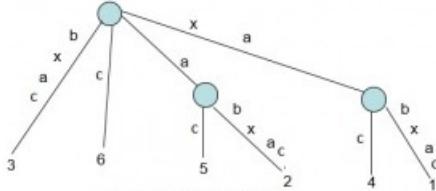


Figure 14: T6 for S[1..6]
 Adding suffixes of xabxac (xabxac, abxac, bxac, xac, ac, c)
 Rule 1 - Extending path label in existing leaf edge
 Rule 2 - Three new leaf edges and two new internal nodes

In next parts ([Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#)), we will discuss suffix links, active points, few tricks and finally code implementations ([Part 6](#)).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

Ukkonens Suffix Tree Construction Part 2

In [Ukkonens Suffix Tree Construction Part 1](#), we have seen high level Ukkonens Algorithm. This 2nd part is continuation of [Part 1](#). Please go through [Part 1](#), before looking at current article.

In Suffix Tree Construction of string S of length m , there are m phases and for a phase j ($1 \leq j \leq m$), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules (discussed in [Part 1](#)).

To do j^{th} extension of phase $i+1$ (adding character $S[i+1]$), we first need to find end of the path from the root labelled $S[j..i]$ in the current tree. One way is start from root and traverse the edges matching $S[j..i]$ string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in $O(m)$ which we will see now.

Suffix links

For an internal node v with path-label xA , where x denotes a single character and A denotes a (possibly empty) substring, if there is another node $s(v)$ with path-label A , then a pointer from v to $s(v)$ is called a suffix link.

If A is empty string, suffix link from internal node will go to root node.

There will not be any suffix link from root node (As its not considered as internal node).

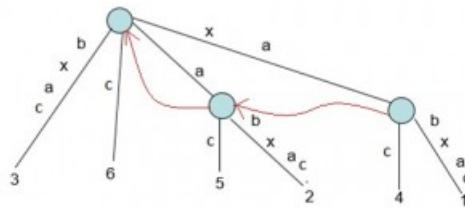


Figure 15: Suffix Links in red arrows

In extension j of some phase i , if a new internal node v with path-label xA is added, then in extension $j+1$ in the same phase i :

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension $j+1$ of same phase i , we will create a suffix link from the internal node created in j^{th} extension to the node with path labelled A .

So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension.

In any implicit suffix tree T_i after phase i , if internal node v has path-label xA , then there is a node $s(v)$ in T_i with path-label A and node v will point to node $s(v)$ using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

How suffix links are used to speed up the implementation?

In extension j of phase $i+1$, we need to find the end of the path from the root labelled $S[j..i]$ in the current tree. One way is start from root and traverse the edges matching $S[j..i]$ string. Suffix links provide a short cut to find end of the path.

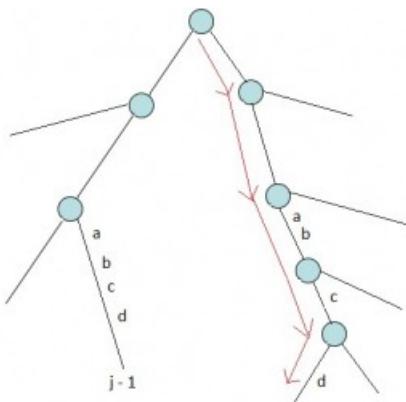


Figure 16: Traversal from root to leaf in extension j of phase $i+1$, to find end of $S[j..i]$, when suffix link is not used

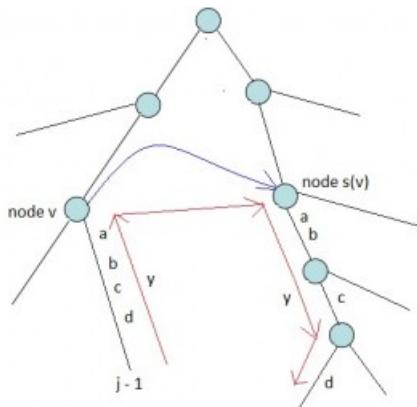


Figure 17: Traversal from root to leaf in extension j of phase $i+1$, to find end of $S[j..i]$, when suffix link (blue arrow) is used

So we can see that, to find end of path $S[j..i]$, we need not traverse from root. We can start from the end of path $S[j-1..i]$, walk up one edge to node v (i.e. go to parent node), follow the suffix link to $s(v)$, then walk down the path y (which is abcd here in Figure 17).

This shows the use of suffix link is an improvement over the process.

Note: In the next part 3, we will introduce activePoint which will help to avoid walk up. We can directly go to node $s(v)$ from node v .

When there is a suffix link from node v to node $s(v)$, then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node $s(v)$ to a leaf. In Figure 17, there is a path label abcd from node v to a leaf, then there is a path will same label abcd from node $s(v)$ to a leaf.

This fact can be used to improve the walk from $s(v)$ to leaf along the path y . This is called skip/count trick.

Skip/Count Trick

When walking down from node $s(v)$ to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.

If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

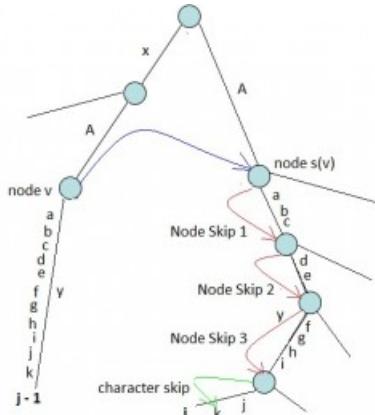


Figure 18: skip/count trick: substring y from node v has length 11. substring y from node $s(v)$ is two characters down the last node , after 3 node skips

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are m phases and each phase takes $O(m)$.

Edge-label compression

So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S . With this, suffix tree needs $O(m)$ space.

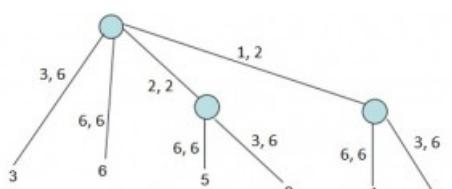


Figure 19: suffix tree for string xaxbac with edge-label compression
Figure 14 shows same suffix tree without edge-label compression

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick skip/count is seen already while walk down).

Observation 1: Rule 3 is show stopper

In a phase i , there are i extensions (1 to i) to be done.

When rule 3 applies in any extension j of phase $i+1$ (i.e. path labelled $S[j..i]$ continues with character $S[i+1]$), then it will also apply in all further extensions of same phase (i.e. extensions $j+1$ to $i+1$ in phase $i+1$). That's because if path labelled $S[j..i]$ continues with character $S[i+1]$, then path labelled $S[j+1..i]$, $S[j+2..i]$, $S[j+3..i]$, $S[i..i]$ will also continue with character $S[i+1]$.

Consider Figure 11, Figure 12 and Figure 13 in [Part 1](#) where Rule 3 is applied.

In Figure 11, xab is added in tree and in Figure 12 (Phase 4), we add next character x . In this, 3 extensions are done (which adds 3 suffixes). Last suffix x is already present in tree.

In Figure 13, we add character a in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes xa and a are already present in tree. This shows that if suffix $S[j..i]$ present in tree, then ALL the remaining suffixes $S[j+1..i]$, $S[j+2..i]$, $S[j+3..i]$, $S[i..i]$ will also be there in tree and no work needed to add those remaining suffixes.

So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension $j+1$, then we need to add suffix link from node v to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

Trick 2

Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf

Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j , extension rule 1 will always apply to extension j in all successive phases.

Consider Figure 9 to Figure 14 in [Part 1](#).

In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i , there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.

Also rule 2 creates a new leaf always (and internal node sometimes).

If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, , J_i), then $J_i \leq J_{i+1}$

J_i will be equal to J_{i+1} when there are no new leaf created in phase $i+1$ (i.e rule 3 is applied in J_{i+1} extension)

In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$

In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$

In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$

J_i will be less than J_{i+1} when few new leaves are created in phase $i+1$.

In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase $i+1$, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.

Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i , end = i for leaf edges, for phase $i+1$, end = $i+1$ for leaf edges.

Trick 3

In any phase i , leaf edges may look like (p, i) , (q, i) , (r, i) , . where p, q, r are starting position of different edges and i is end position of all. Then in phase $i+1$, these leaf edges will look like $(p, i+1)$, $(q, i+1)$, $(r, i+1)$,. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e) , (q, e) , (r, e) ,. In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree T_m could be implicit tree if a suffix is prefix of another. So we can add a $\$$ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next [Part 3](#), we will take string $S = abcabxabcd$ as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints.

We will continue to discuss the algorithm in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

Ukkonens Suffix Tree Construction Part 3

This article is continuation of following two articles:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

Please go through [Part 1](#) and [Part 2](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonens algorithm, suffix link and three implementation tricks.

Here we will take string $S = \text{abcabxabcd}$ as an example and go through all the things step by step and create the tree.

We will add $\$$ (discussed in [Part 1](#) why we do this) so string S would be $\text{abcabxabcd\$}$.

While building suffix tree for string S of length m :

- There will be m phases 1 to m (one phase for each character)
In our current example, m is 11, so there will be 11 phases.
- First phase will add first character a in the tree, second phase will add second character b in tree, third phase will add third character c in tree, , m^{th} phase will add m^{th} character in tree (This makes Ukkonens algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies) without going through all i extensions
- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being traversed)
- Phase 1 will read first character from the string, will go through 1 extension.

(In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices The Edge-label compression discussed in [Part 2](#))

Extension 1 will add suffix a in tree. We start from root and traverse path with label a . There is no path from root, going out with label a , so create a leaf edge (Rule 2).

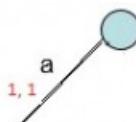


Figure 20: Phase 1, Extension 1 - Rule 2 applied
Created a leaf edge (1, 1)
Phase 1 completes here

Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)

For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions.
In our example, phase 2 will read second character b . Suffixes to be added are ab and b .
Extension 1 adds suffix ab in tree.
Path for label a ends at leaf edge, so add b at the end of this edge.
Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

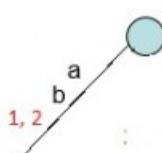


Figure 21: Phase 2, Extension 1 - Rule 1 applied
Extended the leaf edge from (1,1) to (1,2)

Extension 2 adds suffix b in tree. There is no path from root, going out with label b , so creates a leaf edge (Rule 2).

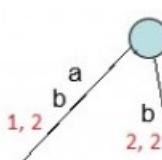


Figure 22: Phase 2, Extension 2 - Rule 2 applied
Created a leaf edge (2, 2)
Phase 2 completes here

Phase 2 completes with the completion of extension 2.

Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions.
In our example, phase 3 will read third character c. Suffixes to be added are abc, bc and c.
Extension 1 adds suffix abc in tree.
Path for label ab ends at leaf edge, so add c at the end of this edge.
Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

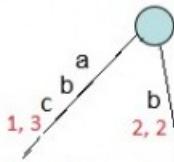


Figure 23: Phase 3, Extension 1 - Rule 1 applied
Extended the leaf edge from (1,2) to (1,3)

Extension 2 adds suffix bc in tree.

Path for label b ends at leaf edge, so add c at the end of this edge.

Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

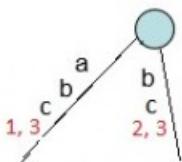


Figure 24: Phase 3, Extension 2 - Rule 1 applied
Extended the leaf edge from (2,2) to (2,3)

Extension 3 adds suffix c in tree. There is no path from root, going out with label c, so creates a leaf edge (Rule 2).

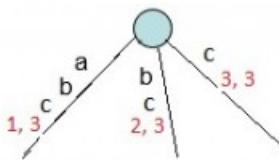


Figure 25: Phase 3, Extension 3 - Rule 2 applied
Created a leaf edge (3,3)
Phase 3 completes here

Phase 3 completes with the completion of extension 3.

Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions.
In our example, phase 4 will read fourth character a. Suffixes to be added are abca, bca, ca and a.
Extension 1 adds suffix abca in tree.
Path for label abc ends at leaf edge, so add a at the end of this edge.
Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

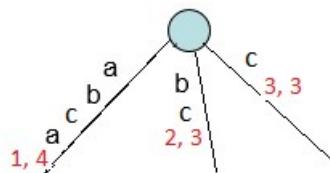


Figure 26: Phase 4, Extension 1
Rule 1 applied

Extension 2 adds suffix bca in tree.

Path for label bc ends at leaf edge, so add a at the end of this edge.

Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

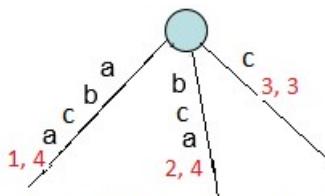


Figure 27: Phase 4, Extension 2
Rule 1 applied

Extension 3 adds suffix *ca* in tree.

Path for label *c* ends at leaf edge, so add *a* at the end of this edge.

Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

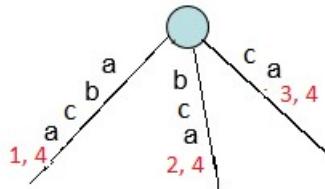


Figure 28: Phase 4, Extension 3
Rule 1 applied

Extension 4 adds suffix *a* in tree.

Path for label *a* exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix *a* is not seen explicitly (because it doesn't end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.

Phase 4 completes as soon as Rule 3 is applied while Extension 4.

Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

1. At the end of any phase *i*, there are at most *i* leaf edges (if *i*th character is not seen so far, there will be *i* leaf edges, else there will be less than *i* leaf edges).
e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).
2. After completing phase *i*, end indices of all leaf edges are *i*. How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the end index? Answer is NO.
For this, we will maintain a global variable (say END) and we will just increment this global variable END and all leaf edge end indices will point to this global variable. So this way, if we have *j* leaf edges after phase *i*, then in phase *i+1*, first *j* extensions (1 to *j*) will be done by just incrementing variable END by 1 (END will be *i+1* at the point).
Here we just implemented the trick 3 **Once a leaf, always a leaf**. This trick processes all the *j* leaf edges (i.e. extension 1 to *j*) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to *j*). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.
3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are *j* leaf edges after phase *i*, then in phase *i+1*, first *j* extensions will follow Rule 1 and will be done in constant time using trick 3. There are *i+1-j* extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase *i* is completed. If previous phase *i* went through all the *i* extensions (when *i*th character is unique so far), then in next phase *i+1*, trick 3 will take care of first *i* suffixes (the *i* leaf edges) and then extension *i+1* will start from root node and it will insert just one character [$(i+1)^{th}$] suffix in tree by creating a leaf edge using Rule 2.

If previous phase *i* completes early (and this will happen if and only if rule 3 applies when *i*th character is already seen before), say at *j*th extension (i.e. rule 3 is applied at *j*th extension), then there are *j-1* leaf edges so far.

We will state few more facts (which may be a repeat, but we want to make sure its clear to you at this point) here based on discussion so far:

- *Phase 1 starts with Rule 2, all other phases start with Rule 1*
- *Any phase ends with either Rule 2 or Rule 3*
- *Any phase *i* may go through a series of *j* extensions ($1 \leq j \leq i$). In these *j* extensions, first *p* ($0 \leq p < i$) extensions will follow Rule 1, next *q* ($0 \leq q \leq i-p$) extensions will follow Rule 2 and next *r* ($0 \leq r \leq 1$) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3*
- *In a phase *i*, $p + q + r \leq i$*
- *At the end of any phase *i*, there will be $p+q$ leaf edges and next phase *i+1* will go through Rule 1 for first $p+q$ extensions*

In the next phase $i+1$, trick 3 (Rule 1) will take care of first $j-1$ suffixes (the $j-1$ leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall algorithm will not be linear. activePoint comes to the rescue here.

In previous phase i , while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase $i+1$ will start exactly from the same point and we start matching path against $(i+1)^{\text{th}}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1st p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case APCFALZ discussed below) and at the end of current extension, we reset activePoint as appropriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

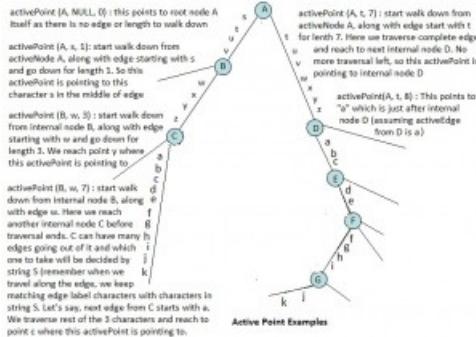
activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case APCFALZ discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).

To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.

activeNode: This could be root node or an internal node.

activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.

activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO. (*click on below image to see it clearly*)



After phase i , if there are j leaf edges then in phase $i+1$, first j extensions will be done by trick 3. activePoint will be needed for the extensions from $j+1$ to $i+1$ and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i , then before we move on to next phase $i+1$, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLength is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase $i+1$.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is $(A, s, 11)$ in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLength as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:

$(A, s, 11) \ggg (B, w, 7) - \ggg (C, a, 3)$

All above three activePoints refer to same point c

Lets take another example.

If activePoint is $(D, a, 11)$ at the start of an extension, then while walk down, below is the sequence of changes in activePoint:

$(D, a, 10) \ggg (E, d, 7) \ggg (F, f, 5) \gg (G, j, 1)$

All above activePoints refer to same point k.

If activePoints are $(A, s, 3)$, $(A, t, 5)$, $(B, w, 1)$, $(D, a, 2)$ etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.

The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Lets consider an activePoint (A, s, 0) in the above activePoint example figure. And lets say current character being processed from string S is x (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. x, because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

4. While code implementation, we will loop through all the characters of string S one by one. Each loop for i^{th} character will do processing for phase i. Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase $i+1$, we dont really have to perform all $i+1$ extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
[Ukkonens suffix tree algorithm in plain English](#)

Ukkonens Suffix Tree Construction Part 4

This article is continuation of following three articles:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

[Ukkonens Suffix Tree Construction Part 3](#)

Please go through [Part 1](#), [Part 2](#) and [Part 3](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonens algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string abcabxabcd where we went through four phases of building suffix tree.

Lets revisit those four phases we have seen already in [Part 3](#), in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding we are giving character value to activeEdge, but in code implemtation, it will be index of the character) and activeLength is ZERO.
- The global variable END and remainingSuffixCount are initialized to ZERO

*****Phase 1*****

In Phase 1, we read 1st character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be a). This is **APCFALZ**
 - Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
 - Once extension is performed, decrement the remainingSuffixCount by 1
 - At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly).

Figure 20 in [Part 3](#) is the resulting tree after phase 1.

*****Phase 2*****

In Phase 2, we read 2nd character (b) from string S

- Set END to 2 (This will do extension 1)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be b). This is **APCFALZ**
 - Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
 - Once extension is performed, decrement the remainingSuffixCount by 1
 - At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly).

Figure 22 in [Part 3](#) is the resulting tree after phase 2.

*****Phase 3*****

In Phase 3, we read 3rd character (c) from string S

- Set END to 3 (This will do extensions 1 and 2)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be c). This is **APCFALZ**
 - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
 - Once extension is performed, decrement the remainingSuffixCount by 1
 - At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly).

Figure 25 in [Part 3](#) is the resulting tree after phase 3.

*****Phase 4*****

In Phase 4, we read 4th character (a) from string S

- Set END to 4 (This will do extensions 1, 2 and 3)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be a). This is **APCFALZ**.
 - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 skip/count). In our example, edge a is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).
 - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

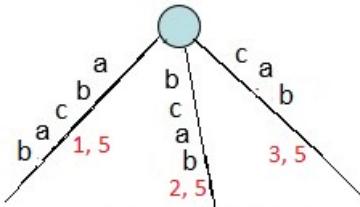
At the end of phase 4, remainingSuffixCount is 1 (One suffix a, the last one, is not added explicitly in tree, but it is there in tree implicitly). Figure 28 in [Part 3](#) is the resulting tree after phase 4.

Revisiting completed for 1st four phases, we will continue building the tree and see how it goes.

*****Phase 5*****

In phase 5, we read 5th character (b) from string S

- Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix ab and extension 5 is supposed to add suffix b in tree)
- Run a loop remainingSuffixCount times (i.e. two times) as below:
 - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge a is present going out of activeNode (i.e. root).
 - Do a walk down (The trick 1 skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)
 - Check if current character of string S (which is b) is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
 - At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)



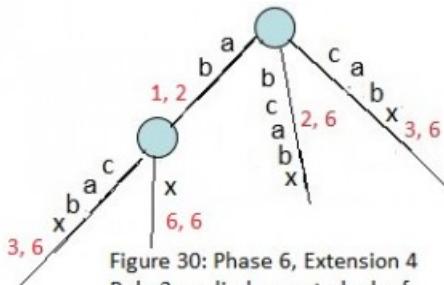


Figure 30: Phase 6, Extension 4
Rule 2 applied - created a leaf edge and also a new internal node

Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in [Part 3](#).

activePoint change for extension rule 2 (APCFER2):

Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set S[i remainingSuffixCount + 1] where i is current phase number. Can you see why this change in activePoint? Look at current extension we just discussed above for phase 6 (i=6) again where we added suffix abx. There activeLength is 2 and activeEdge is a. Now in next extension, we need to add suffix bx in the tree, i.e. path label in next extension should start with b. So b (the 5th character in string S) should be active edge for next extension and index of b will be i remainingSuffixCount + 1 (6 2 + 1 = 5). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.

What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**

Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in [Part 2](#). Lets say in phase i and extension j, suffix xAabcdedg was added in tree. At that point, lets say activePoint was (Node-V, a, 7), i.e. point g. So for next extension j+1, we would add suffix Aabcdefg and for that we need to traverse 2nd path shown in Figure 18. This can be done by following suffix link from current activeNode v. Suffix link takes us to the path to be traversed somewhere in between [Node s(v)] below which the path is exactly same as how it was below the previous activeNode v. As said earlier, activePoint gets closer to root by length 1 after every extension, this reduction in length will happen above the node s(v) but below s(v), no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

- At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
- Next suffix to be added is bx (with remainingSuffixCount 2).
- Current character x from string S doesnt match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
- Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix bx added in tree.

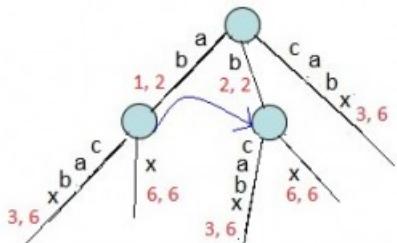


Figure 31: Phase 6, Extension 5 - Rule 2 applied
Created a leaf edge, a new internal node and
suffix link from previous internal node of extension 4
to the current newly internal node

- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is x (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous extensions internal node goes to root (as no new internal node created in current extension 6).
- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix x added in tree

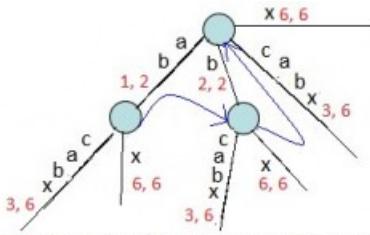


Figure 32: Phase 6, Extension 6 - Rule 2 applied
Created a leaf edge and suffix link from previous internal node of extension 5 to root node (as no new internal node created in extension 6, so suffix link goes to root)

This completes the phase 6.

Note that phase 6 has completed all its 6 extensions (Why? Because the current character c was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters abcabx read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension i, points to another internal node or root (if activeNode is root in extension i+1) by the end of extension i+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walkdown from root can be avoided.

We will go through rest of the phases (7 to 11) in [Part 5](#) and build the tree completely and after that, we will see the code for the algorithm in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
[Ukkonen's suffix tree algorithm in plain English](#)

Ukkonen's Suffix Tree Construction Part 5

This article is continuation of following four articles:

[Ukkonen's Suffix Tree Construction Part 1](#)

[Ukkonen's Suffix Tree Construction Part 2](#)

[Ukkonen's Suffix Tree Construction Part 3](#)

[Ukkonen's Suffix Tree Construction Part 4](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#) and [Part 4](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonens algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string abcabxabcd where we went through six phases of building suffix tree.

Here, we will go through rest of the phases (7 to 11) and build the tree completely.

*****Phase 7*****

In phase 7, we read 7th character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) because we have 6 leaf edges so far by the end of previous phase 6.

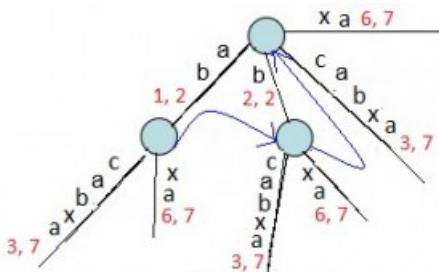


Figure 33: Phase 7, Extension 6 - Rule 1 applied

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix a)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be a). This is APCFALZ. Now activePoint becomes (root, a, 0).
 - Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge a is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (APCFER3) and stop any further processing.
 - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix a, the last one, is not added explicitly in tree, but it is there in tree implicitly).

Above Figure 33 is the resulting tree after phase 7.

*****Phase 8*****

In phase 8, we read 8th character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).

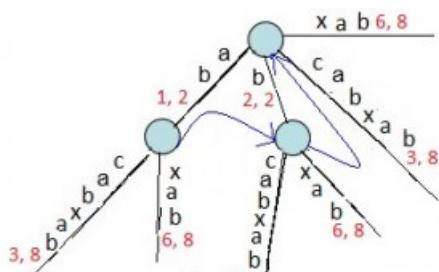


Figure 34: Phase 8, Extension 6 - Rule 1 applied

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes ab and b respectively)
- Run a loop remainingSuffixCount times (i.e. two times) as below:
 - Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge a is present going out of activeNode (i.e. root).
 - Do a walk down (The trick 1 skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
 - Check if current character of string S (which is b) is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (APCFER3) and we stop here (Rule 3).

- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, ab and b, the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

*****Phase 9*****

In phase 9, we read 9th character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) because we have 6 leaf edges so far by the end of previous phase 8.

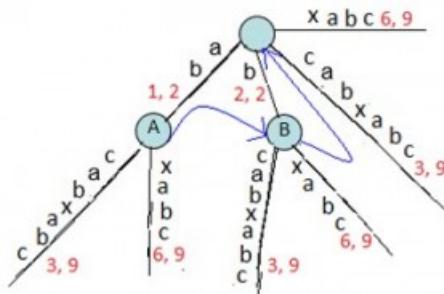


Figure 35: Phase 9, Extension 6 - Rule 1 applied

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes abc, bc and c respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
 - Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge a is present going out of activeNode (i.e. root).
 - Do a walk down (The trick 1 skip/count) if necessary. In current phase 9, walk down needed as activeLength(2) \geq edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on APCFWD (This is first time APCFWD is being applied in our example).
 - Check if current character of string S (which is c) is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (APCFER3) and we stop here (Rule 3).
 - At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, abc, bc and c, the last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

*****Phase 10*****

In phase 10, we read 10th character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) because we have 6 leaf edges so far by the end of previous phase 9.

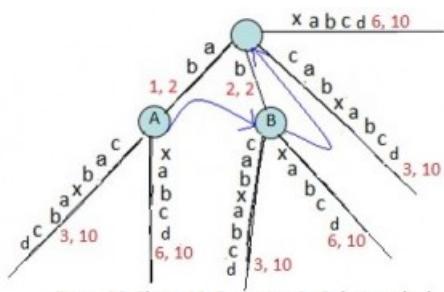


Figure 36: Phase 10, Extension 6 - Rule 1 applied

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes abcd, bcd, cd and d respectively)
- Run a loop remainingSuffixCount times (i.e. four times) as below:

*****Extension 7*****

- Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge c is present going out of activeNode (Node A).
- Do a walk down (The trick 1 skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength.
- Check if current character of string S (which is d) is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with d going out of activePoint, so we create a leaf edge with label d. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)

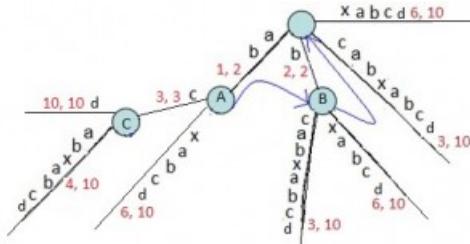


Figure 37: Phase 10, Extension 7 - Rule 2 applied
New leaf edge and new internal node created

The newly created internal node **c** (in above Figure) in current extension 7, will get its suffix link set in next extension 8 (see Figure 38 below).

- Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix abcd added in tree.
- Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that's going to be Node B. There is no change in activeEdge and activeLength (This is APCFER2C2). So new activePoint is (Node B, c, 1).

*****Extension 8*****

- Now in extension 8 (here we will add suffix bcd), while adding character d after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character d at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character d at activePoint (Node B, c, 1). So logic will be same and here we a new leaf edge with label d and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

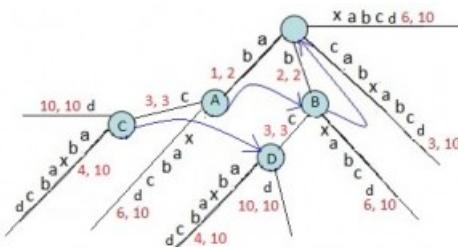


Figure 38: Phase 10, Extension 8 - Rule 2 applied
New leaf edge created and a new internal node created
Also the internal node C created in previous extension 7, pointing to the internal node D via suffix link

Please note the node C from previous extension (see Figure 37 above) got its suffix link set here and node D created in current extension will get its suffix link set in next extension. What happens if no new node created in next extensions? We have seen this before in Phase 6 (Part 4) and will see again in last extension of this Phase 10. Stay Tuned.

- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix bcd added in tree.
- Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is Root Node. There is no change in activeEdge and activeLength (This is APCFER2C2). So new activePoint is (root, c, 1).

*****Extension 9*****

- Now in extension 9 (here we will add suffix cd), while adding character d after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

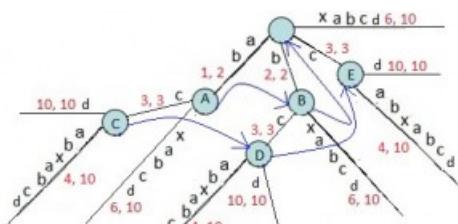


Figure 39: Phase 10, Extension 9 - Rule 2 applied
New leaf edge created and a new internal node created
Also the internal node D created in previous extension 8, pointing to the internal node E via suffix link

- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix cd added in tree.
- Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on APCFER2C1, activeNode will remain root, activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be d. So new activePoint

is (root, d, 0).

*****Extension 10*****

- Now in extension 10 (here we will add suffix d), while adding character d after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

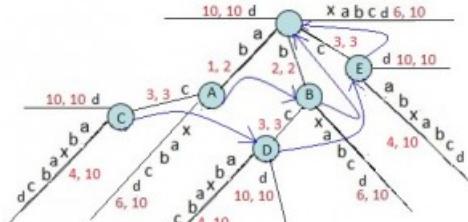


Figure 40: Phase 10, Extension 10 - Rule 2 applied
New leaf edge created. Also the internal node E created in previous extension 9, pointing to root node via suffix link

Internal Node created in previous extension, waiting for suffix link to be set in next extension, points to root if no internal node created in next extension. In code implementation, as soon as a new internal node (Say A) gets created in an extension j, we will set its suffix link to root node and in next extension j+1, if Rule 2 applies on an existing or newly created node (Say B) or Rule 3 applies with some active node (Say B), then suffix link of node A will change to the new node B, else node A will keep pointing to root

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix d added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far)
- activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them
- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extensions activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get its suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got its suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).
- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

*****Phase 11*****

In phase 11, we read 11th character (\$) from string S

- Set END to 11 (This will do extensions 1 to 10) because we have 10 leaf edges so far by the end of previous phase 10.

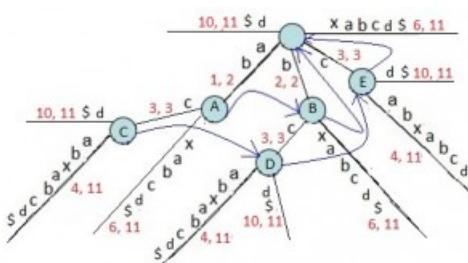


Figure 41: Phase 11, Extension 10 - Rule 1 applied

- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix \$ to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character \$ of string S being processed (**APCFALZ**).
- There is no edge going out from activeNode root, so a leaf edge with label \$ will be created (Rule 2).

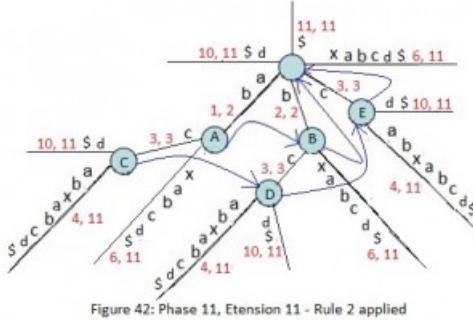


Figure 42: Phase 11, Extension 11 - Rule 2 applied

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix \$ added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character \$ was not seen before in string S so far)

Now we have added all suffixes of string abcabxabcd\$ in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as stringSize labelSize + 1. Indexed suffix tree will look like below:

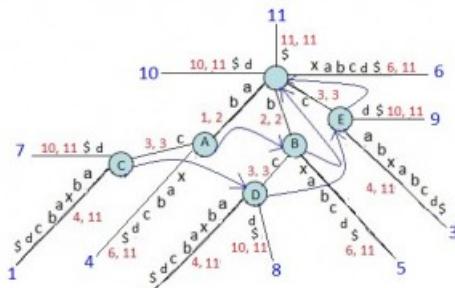


Figure 43: Final Suffix Tree for String S = abcabxabcd\$

In above Figure, suffix indices are shown as character position starting with 1 (Its not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index j (1 to m for string of length m) in above figure, in code implementation, it will be j-1 (0 to m-1)

And we are done !!!!

Data Structure to represent suffix tree

How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.

Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.

In the next [Part 6](#), we will discuss the data structure we will use in our code implementation and the code as well.

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
[Ukkonens suffix tree algorithm in plain English](#)

Ukkonens Suffix Tree Construction Part 6

This article is continuation of following five articles:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

[Ukkonens Suffix Tree Construction Part 3](#)

[Ukkonens Suffix Tree Construction Part 4](#)

[Ukkonens Suffix Tree Construction Part 5](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonens algorithm, suffix link and three implementation tricks and activePoints along with an example string abcabxabcd where we went through all phases of building suffix tree.

Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of [Part 5](#) article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.

There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** This will point to other node where current node should point via suffix link.
- **start, end** These two will store the edge label details from parent node to current node. (start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Lets say there are two nodes A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.
- **suffixIndex** This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1.

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? Root is a special node, with no parent and so its start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf? If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     *node is connected to its parent node. Each edge will
     *connect two nodes, one parent and one child, and
     *(start, end) interval of a given edge will be stored
     *in the child node. Lets say there are two nodes A and B
     *connected by an edge with indices (5, 8) then this
     *indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
```

```

    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase.*/
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
       actual suffix index will be set later for leaves
       at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;
}

```

```

/*Increment remainingSuffixCount indicating that a
new suffix added to the list of suffixes yet to be
added in tree*/
remainingSuffixCount++;

/*set lastNewNode to NULL while starting a new phase,
indicating there is no internal node waiting for
it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children[text[activeEdge]] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children[text[activeEdge]] =
            newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from an existing node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
         is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to current active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
             and move on to next phase*/
            break;
        }

        /*We will be here when activePoint is in middle of
         the edge being traversed and current character
         being processed is not on the edge (we fall off
         the tree). In this case, we add a new internal node
         and a new leaf edge going out of that new node. This
         is Extension Rule 2, where a new leaf edge and a new
         internal node get created*/
        splitEnd = (int*) malloc(sizeof(int));
        *splitEnd = next->start + activeLength - 1;

        //New internal node
        Node *split = newNode(next->start, splitEnd);
    }
}

```

```

activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {

```

```

        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
// strcpy(text, "abc"); buildSuffixTree();
// strcpy(text, "xabxac#"); buildSuffixTree();
// strcpy(text, "xabxa"); buildSuffixTree();
// strcpy(text, "xabxa$"); buildSuffixTree();
strcpy(text, "abcabxabcd$"); buildSuffixTree();
// strcpy(text, "geeksforgeeks$"); buildSuffixTree();
// strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
// strcpy(text, "AABAACAAADAABAAABAA$"); buildSuffixTree();
    return 0;
}

```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous [Part 5](#) article):

```

$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
xabcd$ [5]

```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way.

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, [KMP](#) otherwise)
- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- [Linear Time Suffix Array Creation](#)

The above basic problems can be solved by DFS traversal on suffix tree.

We will soon post articles on above problems and others like below:

- Build [Generalized suffix tree](#)
- Linear Time [Longest common substring problem](#)
- Linear Time [Longest palindromic substring](#)

And [More](#).

Test your understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string AABAACAAADAABAAABAA\$ on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3.
Following are the rules applied on five consecutive extensions in some Phase i ($i > 5$), which ones are valid:
 A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
 B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
 C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
 D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
 E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
 F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have its suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not? Can it happen that a new node created in extension j, may not get its right suffix link in next extension $j+1$ and get the right one in later extensions like $j+2, j+3$ etc?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- [Suffix Tree Application 1 Substring Check](#)
- [Suffix Tree Application 2 Searching All Patterns](#)
- [Suffix Tree Application 3 Longest Repeated Substring](#)
- [Suffix Tree Application 4 Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Suffix Tree Application 6 Longest Palindromic Substring](#)

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
[Ukkonen's suffix tree algorithm in plain English](#)

Generalized Suffix Tree 1

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for [substring check](#), [searching all patterns](#), [longest repeated substring](#) and [built suffix array](#) (All linear time operations).

There are lots of other problems where multiple strings are involved.

e.g. pattern searching in a text file or dictionary, spell checker, phone book, [Autocomplete](#), [Longest common substring problem](#), [Longest palindromic substring](#) and [More](#).

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.

A suffix tree made of a set of strings is known as [Generalized Suffix Tree](#).

We will discuss a simple way to build [Generalized Suffix Tree](#) here for **two strings only**.

Later, we will discuss another approach to build [Generalized Suffix Tree](#) for **two or more strings**.

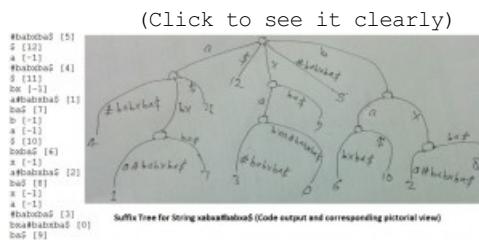
Here we will use the [suffix tree implementation](#) for one string discussed already and modify that a bit to build [generalized suffix tree](#).

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y\$ where # and \$ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y\$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

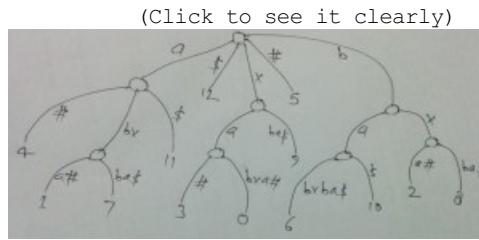
Lets say X = xabxa, and Y = babxba, then

X#Y\$ = xabxa#babxba\$

If we run the code implemented at [Ukkonen's Suffix Tree Construction Part 6](#) for string xabxa#babxba\$, we get following output:



We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels #babxba\$, a#babxa\$ and bxa#babxa\$, we can remove babxa\$ (belongs to 2nd input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:



Below implementation is built on top of [original implementation](#). Here we are removing unwanted characters on path labels. If a path label has # character in it, then we are trimming all characters after the # in that path label.

Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y\$

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nods A and B
```

```

connected by an edge with indices (5, 8) then this
indices (5, 8) will be stored in node B. */
int start;
int *end;

/*for leaf nodes, it stores the index of suffix for
 the path from root to leaf*/
int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
 same phase. lastNewNode will be set to NULL when last
 newly created internal node (if there is any) got it's
 suffix link reset to new internal node created in next
 extension of same phase.*/
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
     actual suffix index will be set later for leaves
     at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

```

```

}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
            from an existng node (the current activeNode), and
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
            is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to curent active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

                //APCFER3
                activeLength++;
                /*STOP all further processing in this phase
                and move on to next phase*/
                break;
            }
        }
        /*We will be here when activePoint is in middle of
        the edge being traversed and current character
        being processed is not on the edge (we fall off
        the tree). In this case, we add a new internal node
    }
}

```

```

        and a new leaf edge going out of that new node. This
        is Extension Rule 2, where a new leaf edge and a new
        internal node get created*/
        splitEnd = (int*) malloc(sizeof(int));
        *splitEnd = next->start + activeLength - 1;

        //New internal node
        Node *split = newNode(next->start, splitEnd);
        activeNode->children[text[activeEdge]] = split;

        //New leaf coming out of new internal node
        split->children[text[pos]] = newNode(pos, &leafEnd);
        next->start += activeLength;
        split->children[text[next->start]] = next;

        /*We got a new internal node here. If there is any
         internal node created in last extensions of same
         phase which is still waiting for it's suffix link
         reset, do it now.*/
        if (lastNewNode != NULL)
        {
            /*suffixLink of lastNewNode points to current newly
             created internal node*/
            lastNewNode->suffixLink = split;
        }

        /*Make the current newly created internal node waiting
         for it's suffix link reset (which is pointing to root
         at present). If we come across any other internal node
         (existing or newly created) in next extension of same
         phase, when a new leaf edge gets added (i.e. when
         Extension Rule 2 applies is any of the next extension
         of same phase) at that point, suffixLink of this node
         will point to that internal node.*/
        lastNewNode = split;
    }

    /* One suffix got added in tree, decrement the count of
     suffixes yet to be added.*/
    remainingSuffixCount--;
    if (activeNode == root && activeLength > 0) //APCFER2C1
    {
        activeLength--;
        activeEdge = pos - remainingSuffixCount + 1;
    }
    else if (activeNode != root) //APCFER2C2
    {
        activeNode = activeNode->suffixLink;
    }
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

```

```

        //Current node is not a leaf as it has outgoing
        //edges from it.
        leaf = 0;
        setSuffixIndexByDFS(n->children[i], labelHeight +
                           edgeLength(n->children[i]));
    }
}
if (leaf == 1)
{
for(i= n->start; i<= *n->end; i++)
{
    if(text[i] == '#') //Trim unwanted characters
    {
        n->end = (int*) malloc(sizeof(int));
        *(n->end) = i;
    }
}
    n->suffixIndex = size - labelHeight;
    printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
// strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
// strcpy(text, "xabxa#babxba$"); buildSuffixTree();
    return 0;
}

```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```

# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]

```

```
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]
```

If two strings are of size M and N, this implementation will take $O(M+N)$ time and space.

If input strings are not concatenated already, then it will take $2(M+N)$ space in total, $M+N$ space to store the generalized suffix tree and another $M+N$ space to store concatenated string.

Followup:

Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.

We will discuss another approach to build generalized suffix tree soon where we will need only one unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 Substring Check](#)
- [Suffix Tree Application 2 Searching All Patterns](#)
- [Suffix Tree Application 3 Longest Repeated Substring](#)
- [Suffix Tree Application 4 Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Suffix Tree Application 6 Longest Palindromic Substring](#)

Suffix Tree Application 4 Build Linear Time Suffix Array

Given a string, build its [Suffix Array](#)

We have already discussed following two ways of building suffix array:

- [Naive O\(n²Logn\) algorithm](#)
- [Enhanced O\(nLogn\) algorithm](#)

Please go through these to have the basic understanding.

Here we will see how to build suffix array in linear time using suffix tree.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

[Ukkonen's Suffix Tree Construction Part 1](#)

[Ukkonen's Suffix Tree Construction Part 2](#)

[Ukkonen's Suffix Tree Construction Part 3](#)

[Ukkonen's Suffix Tree Construction Part 4](#)

[Ukkonen's Suffix Tree Construction Part 5](#)

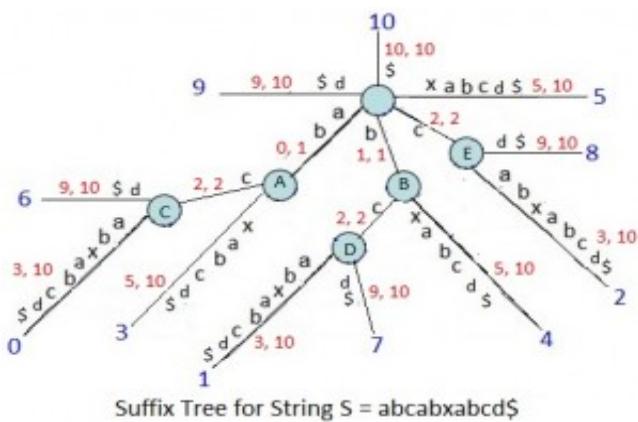
[Ukkonen's Suffix Tree Construction Part 6](#)

Lets consider string abcabxabcd.

Its suffix array would be:

0 6 3 1 7 4 2 8 9 5

Lets look at following figure:



This is suffix tree for String abcabxabcd\$

If we do a DFS traversal, visiting edges in lexicographic order (we have been doing the same traversal in other Suffix Tree Application articles as well) and print suffix indices on leaves, we will get following:

10 0 6 3 1 7 4 2 8 9 5

\$ is lexicographically lesser than [a-zA-Z].

The suffix index 10 corresponds to edge with \$ label.

Except this 1st suffix index, the sequence of all other numbers gives the suffix array of the string.

So if we have a suffix tree of the string, then to get its suffix array, we just need to do a lexicographic order DFS traversal and store all the suffix indices in resultant suffix array, except the very 1st suffix index.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And and then create suffix array in linear time
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
```

```

(start, end) interval of a given edge will be stored
in the child node. Lets say there are two nodes A and B
connected by an edge with indices (5, 8) then this
indices (5, 8) will be stored in node B. */
int start;
int *end;

/*for leaf nodes, it stores the index of suffix for
the path from root to leaf*/
int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
}

```

```

    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from an existing node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
             is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to current active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

                //APCFER3
                activeLength++;
                /*STOP all further processing in this phase
                and move on to next phase*/
                break;
            }
        }
        /*We will be here when activePoint is in middle of
         the edge being traversed and current character
    }
}

```

```

being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index

```

```

// if (leaf == 1 && n->start != -1)
//   printf("%d\n", n->suffixIndex);

//Current node is not a leaf as it has outgoing
//edges from it.
leaf = 0;
setSuffixIndexByDFS(n->children[i], labelHeight +
                     edgeLength(n->children[i]));
}
}

if (leaf == 1)
{
    n->suffixIndex = size - labelHeight;
//Uncomment below line to print suffix index
//printf("%d\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
        suffixArray[(*idx)++] = n->suffixIndex;
    }
}

void buildSuffixArray(int suffixArray[])

```

```

{
    int i = 0;
    for(i=0; i< size; i++)
        suffixArray[i] = -1;
    int idx = 0;
    doTraversal(root, suffixArray, &idx);
    printf("Suffix Array for String ");
    for(i=0; i<size; i++)
        printf("%c", text[i]);
    printf(" is: ");
    for(i=0; i<size; i++)
        printf("%d ", suffixArray[i]);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKSS");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "AAAAAAA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "abcabxabcd$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "CCAAACCCGATTAS");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);
}

```

```
    return 0;  
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2  
Suffix Array for String GEEKSFORGEEKS is: 9 1 10 2 5 8 0 11 3 6 7 12 4  
Suffix Array for String AAAAAAAA is: 9 8 7 6 5 4 3 2 1 0  
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6  
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1  
Suffix Array for String abcabxabcd is: 0 6 3 1 7 4 2 8 9 5  
Suffix Array for String CCAAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

Ukkonens Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal of tree take $O(N)$ to build suffix array.

So overall, its linear in time and space.

Can you see why traversal is $O(N)$?? Because a suffix tree of string of length N will have at most $N-1$ internal nodes and N leaves. Traversal of these nodes can be done in $O(N)$.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 Substring Check](#)
- [Suffix Tree Application 2 Searching All Patterns](#)
- [Suffix Tree Application 3 Longest Repeated Substring](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Suffix Tree Application 6 Longest Palindromic Substring](#)

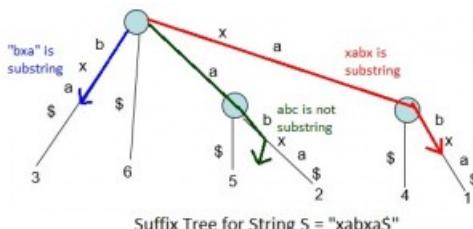
Suffix Tree Application 1 Substring Check

Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms ([KMP](#), [Rabin-Karp](#), [Naive Algorithm](#), [Finite Automata](#)) are already discussed, which can be used for this check. Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.



Here we will build suffix tree using Ukkonens Algorithm, discussed already as below:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

[Ukkonens Suffix Tree Construction Part 3](#)

[Ukkonens Suffix Tree Construction Part 4](#)

[Ukkonens Suffix Tree Construction Part 5](#)

[Ukkonens Suffix Tree Construction Part 6](#)

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

```
// A C program for substring check using Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
 same phase. lastNewNode will be set to NULL when last
 newly created internal node (if there is any) got it's
 suffix link reset to new internal node created in next
 extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;
```

```

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
     actual suffix index will be set later for leaves
     at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)

```

```

activeNode->children[text[activeEdge]] =
    newNode(pos, &leafEnd);

/*A new leaf edge is created in above line starting
 from an existng node (the current activeNode), and
 if there is any internal node waiting for it's suffix
 link get reset, point the suffix link from that last
 internal node to current activeNode. Then set lastNewNode
 to NULL indicating no more node waiting for suffix link
 reset.*/
if (lastNewNode != NULL)
{
    lastNewNode->suffixLink = activeNode;
    lastNewNode = NULL;
}
}

// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
     is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
         and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
     the edge being traversed and current character
     being processed is not on the edge (we fall off
     the tree). In this case, we add a new internal node
     and a new leaf edge going out of that new node. This
     is Extension Rule 2, where a new leaf edge and a new
     internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
     internal node created in last extensions of same
     phase which is still waiting for it's suffix link
     reset, do it now.*/
    if (lastNewNode != NULL)
    {
        /*suffixLink of lastNewNode points to current newly
         created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
     for it's suffix link reset (which is pointing to root
     at present). If we come across any other internal node

```

```

        (existing or newly created) in next extension of same
        phase, when a new leaf edge gets added (i.e. when
        Extension Rule 2 applies is any of the next extension
        of same phase) at that point, suffixLink of this node
        will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
   suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)

```

```

        free(n->end);
        free(n);
    }

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1; // no match
    }
    if(str[idx] == '\0')
        return 1; // match
    return 0; // more characters yet to match
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res != 0)
            return res; // match (res = 1) or no match (res = -1)
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1; // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("Pattern <%s> is a Substring\n", str);
    else
        printf("Pattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "THIS IS A TEST TEXT$");
    buildSuffixTree();

    checkForSubString("TEST");
    checkForSubString("A");
}

```

```

checkForSubString(" ");
checkForSubString("IS A");
checkForSubString(" IS A ");
checkForSubString("TEST1");
checkForSubString("THIS IS GOOD");
checkForSubString("TES");
checkForSubString("TESA");
checkForSubString("ISB");

//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}

```

Output:

```

Pattern <TEST> is a Substring
Pattern <A> is a Substring
Pattern <> is a Substring
Pattern <IS A> is a Substring
Pattern < IS A > is a Substring
Pattern <TEST1> is NOT a Substring
Pattern <THIS IS GOOD> is NOT a Substring
Pattern <TES> is a Substring
Pattern <TESA> is NOT a Substring
Pattern <ISB> is NOT a Substring

```

Ukkonens Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal for substring check takes $O(M)$ for a pattern of length M .

With slight modification in traversal algorithm discussed here, we can answer following:

1. Find all occurrences of a given pattern P present in text T .
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 2 Searching All Patterns](#)
- [Suffix Tree Application 3 Longest Repeated Substring](#)
- [Suffix Tree Application 4 Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Suffix Tree Application 6 Longest Palindromic Substring](#)

Suffix Tree Application 2 Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms ([KMP](#), [Rabin-Karp](#), [Naive Algorithm](#), [Finite Automata](#)) are already discussed, which can be used for this check. Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application ([Substring Check](#)), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through [Substring Check 1st](#).

In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonens Algorithm, discussed already as below:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

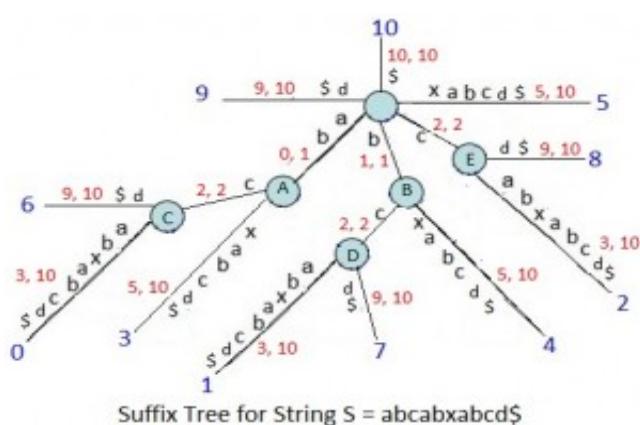
[Ukkonens Suffix Tree Construction Part 3](#)

[Ukkonens Suffix Tree Construction Part 4](#)

[Ukkonens Suffix Tree Construction Part 5](#)

[Ukkonens Suffix Tree Construction Part 6](#)

Lets look at following figure:



This is suffix tree for String $abcabxabcd\$$, showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.

Suffix Index of a path tells the index of a substring (starting from root) on that path.

Consider a path $bcd\$$ in above tree with suffix index 7. It tells that substrings b , bc , bcd , $bcd\$$ are at index 7 in string.

Similarly path $bxabcd\$$ with suffix index 4 tells that substrings b , bx , $bxab$, $bxabc$, $bxabcd$, $bxabcd\$$ are at index 4.

Similarly path $bcabxabcd\$$ with suffix index 1 tells that substrings b , bc , bca , $bcab$, $bcabx$, $bcabxa$, $bcabxabc$, $bcabxabcd$, $bcabxabcd\$$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring b is at indices 1, 4 and 7
- Substring bc is at indices 1 and 7

With above explanation, we should be able to see following:

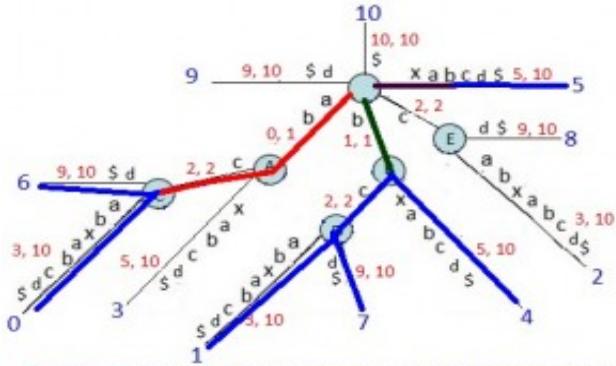
- Substring ab is at indices 0, 3 and 6
- Substring abc is at indices 0 and 6
- Substring c is at indices 2 and 8
- Substring xab is at index 5
- Substring d is at index 9
- Substring cd is at index 8

..

..

Can you see how to find all the occurrences of a pattern in a string?

- 1st of all, check if the given pattern really exists in string or not (As we did in [Substring Check](#)). For this, traverse the suffix tree against the pattern.
- If you find pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string



1. Substring abc is found, subtree traversal shows that it is at indices 0 and 6.
2. Substring b is found, subtree traversal shows that it is at indices 1, 4 and 7.
3. Substring xab is found, subtree traversal shows that it is at index 5.

```

// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;
    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

```

```

/*For root node, suffixLink will be set to NULL
For internal nodes, suffixLink will be set to root
by default in current extension and may change in
next extension*/
node->suffixLink = root;
node->start = start;
node->end = end;

/*suffixIndex will be set to -1 by default and
actual suffix index will be set later for leaves
at the end of all phases*/
node->suffixIndex = -1;
return node;
}

int edgeLength(Node *n) {
if(n == root)
    return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
/*activePoint change for walk down (APCFWD) using
Skip/Count Trick (Trick 1). If activeLength is greater
than current edge length, set next internal node as
activeNode and adjust activeEdge and activeLength
accordingly to represent same activePoint*/
if (activeLength >= edgeLength(currNode))
{
    activeEdge += edgeLength(currNode);
    activeLength -= edgeLength(currNode);
    activeNode = currNode;
    return 1;
}
return 0;
}

void extendSuffixTree(int pos)
{
/*Extension Rule 1, this takes care of extending all
leaves created so far in tree*/
leafEnd = pos;

/*Increment remainingSuffixCount indicating that a
new suffix added to the list of suffixes yet to be
added in tree*/
remainingSuffixCount++;

/*set lastNewNode to NULL while starting a new phase,
indicating there is no internal node waiting for
it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children[text[activeEdge]] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children[text[activeEdge]] =
            newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
        from an existing node (the current activeNode), and
        if there is any internal node waiting for it's suffix
        link get reset, point the suffix link from that last
        internal node to current activeNode. Then set lastNewNode
        to NULL indicating no more node waiting for suffix link
        reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
}

```

```

// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next)) //Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
     is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to current active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
         and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
     the edge being traversed and current character
     being processed is not on the edge (we fall off
     the tree). In this case, we add a new internal node
     and a new leaf edge going out of that new node. This
     is Extension Rule 2, where a new leaf edge and a new
     internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
     internal node created in last extensions of same
     phase which is still waiting for it's suffix link
     reset, do it now.*/
    if (lastNewNode != NULL)
    {
        /*suffixLink of lastNewNode points to current newly
         created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
     for it's suffix link reset (which is pointing to root
     at present). If we come across any other internal node
     (existing or newly created) in next extension of same
     phase, when a new leaf edge gets added (i.e. when
     Extension Rule 2 applies to any of the next extension
     of same phase) at that point, suffixLink of this node
     will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
   suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}

```

```

        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // printf(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
}

```

```

root = newNode(-1, rootEnd);
activeNode = root; //First activeNode will be root
for (i=0; i<size; i++)
    extendSuffixTree(i);
int labelHeight = 0;
setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
int k = 0;
//Traverse the edge with character by character matching
for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
{
if(text[k] != str[idx])
    return -1; // no match
}
if(str[idx] == '\0')
    return 1; // match
return 0; // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)
{
if(n == NULL)
    return 0;
if(n->suffixIndex > -1)
{
printf("\nFound at position: %d", n->suffixIndex);
return 1;
}
int count = 0;
int i = 0;
for (i = 0; i < MAX_CHAR; i++)
{
if(n->children[i] != NULL)
{
    count += doTraversalToCountLeaf(n->children[i]);
}
}
return count;
}

int countLeaf(Node *n)
{
if(n == NULL)
    return 0;
return doTraversalToCountLeaf(n);
}

int doTraversal(Node *n, char* str, int idx)
{
if(n == NULL)
{
    return -1; // no match
}
int res = -1;
//If node n is not root node, then traverse edge
//from node n's parent to node n.
if(n->start != -1)
{
    res = traverseEdge(str, idx, n->start, *(n->end));
    if(res == -1) //no match
        return -1;
    if(res == 1) //match
    {
        if(n->suffixIndex > -1)
            printf("\nsubstring count: 1 and position: %d",
                   n->suffixIndex);
        else
            printf("\nsubstring count: %d", countLeaf(n));
        return 1;
    }
}
//Get the character index to search
idx = idx + edgeLength(n);
//If there is an edge from node n going out
//with current character str[idx], travrse that edge
if(n->children[str[idx]] != NULL)
    return doTraversal(n->children[str[idx]], str, idx);
}

```

```

else
    return -1; // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}

```

Output:

Text: GEEKSFORGEEKS, Pattern to search: GEEKS
 Found at position: 8
 Found at position: 0
 substring count: 2
 Pattern <GEEKS> is a Substring

Text: GEEKSFORGEEKS, Pattern to search: GEEK1
 Pattern <GEEK1> is NOT a Substring

Text: GEEKSFORGEEKS, Pattern to search: FOR
 substring count: 1 and position: 5
 Pattern <FOR> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AABA
 Found at position: 13
 Found at position: 9
 Found at position: 0
 substring count: 3
 Pattern <AABA> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AA

```
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring
```

```
Text: AABAACAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring
```

```
Text: AAAAAAAA, Pattern to search: AAAA
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring
```

```
Text: AAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 8
Pattern <AA> is a Substring
```

```
Text: AAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring
```

```
Text: AAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring
```

Ukkonens Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal for substring check takes $O(M)$ for a pattern of length M and then if there are Z occurrences of the pattern, it will take $O(Z)$ to find indices of all those Z occurrences.

Overall pattern complexity is linear: $O(M + Z)$.

A bit more detailed analysis

How many internal nodes will there be in a suffix tree of string of length N ??

Answer: $N-1$ (Why ??)

There will be N suffixes in a string of length N .

Each suffix will have one leaf.

So a suffix tree of string of length N will have N leaves.

As each internal node has at least 2 children, an N -leaf suffix tree has at most $N-1$ internal nodes.

If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have $Z-1$ internal nodes. A tree with Z leaves can be traversed in $O(Z)$ time.

Overall pattern complexity is linear: $O(M + Z)$.

For a given pattern, Z (the number of occurrences) can be atmost N .

So worst case complexity can be: $O(M + N)$ if Z is close/equal to N (A tree traversal with N nodes take $O(N)$ time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 Substring Check](#)
- [Suffix Tree Application 3 Longest Repeated Substring](#)
- [Suffix Tree Application 4 Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Suffix Tree Application 6 Longest Palindromic Substring](#)

Suffix Tree Application 3 Longest Repeated Substring

Given a text string, find Longest Repeated Substring in the text. If there are more than one Longest Repeated Substrings, get any one of them.

Longest Repeated Substring in GEEKSFORGEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAA is: AAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabpq is: ab (pq is another LRS here)

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3rd Suffix Tree Application). Other approaches will be discussed soon.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonens Algorithm, discussed already as below:

[Ukkonens Suffix Tree Construction Part 1](#)

[Ukkonens Suffix Tree Construction Part 2](#)

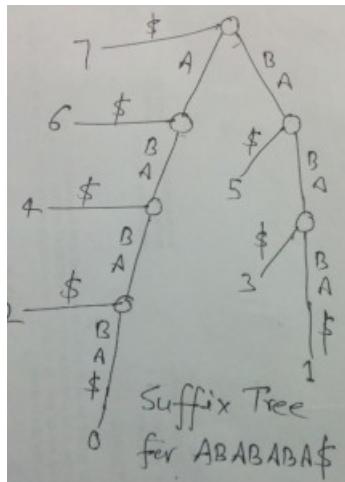
[Ukkonens Suffix Tree Construction Part 3](#)

[Ukkonens Suffix Tree Construction Part 4](#)

[Ukkonens Suffix Tree Construction Part 5](#)

[Ukkonens Suffix Tree Construction Part 6](#)

Lets look at following figure:



This is suffix tree for string ABABABA\$.

In this string, following substrings are repeated:

A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA

And Longest Repeated Substring is ABABA.

In a suffix tree, one node cant have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).

In above figure, we can see that

- Path with Substring A has three internal nodes down the tree
- Path with Substring AB has two internal nodes down the tree
- Path with Substring ABA has two internal nodes down the tree
- Path with Substring ABAB has one internal node down the tree
- Path with Substring ABABA has one internal node down the tree
- Path with Substring B has two internal nodes down the tree
- Path with Substring BA has two internal nodes down the tree
- Path with Substring BAB has one internal node down the tree
- Path with Substring BABA has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??

We can see in figure that, longest repeated substring will end at the internal node which is farthest from the root (i.e. deepest node in the tree),

because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nodes A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;
    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
 same phase. lastNewNode will be set to NULL when last
 newly created internal node (if there is any) got it's
 suffix link reset to new internal node created in next
 extension of same phase.*/
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
     For internal nodes, suffixLink will be set to root
     by default in current extension and may change in
     next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
     actual suffix index will be set later for leaves
     at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}
```

```

}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
        }
        /*Extension Rule 3 (current character being processed

```

```

        is already on the edge) */
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to current active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies to any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

```

```

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)    return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringstartIndex)
{
    if (n == NULL)

```

```

{
    return;
}
int i=0;
if(n->suffixIndex == -1) //If it is internal node
{
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            doTraversal(n->children[i], labelHeight +
                        edgeLength(n->children[i]), maxHeight,
                        substringstartIndex);
        }
    }
}
else if(n->suffixIndex > -1 &&
        (*maxHeight < labelHeight - edgeLength(n)))
{
    *maxHeight = labelHeight - edgeLength(n);
    *substringstartIndex = n->suffixIndex;
}
}

void getLongestRepeatedSubstring()
{
    int maxHeight = 0;
    int substringstartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringstartIndex);
// printf("maxHeight %d, substringstartIndex %d\n", maxHeight,
//         substringstartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringstartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ATCGATCGA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "banana$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "abcpqrabpqpq$");
    buildSuffixTree();
}

```

```

getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "pqrpqpqabab$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}

```

Output:

```

Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAA$ is: AAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
Longest Repeated Substring in ATCGATCGA$ is: ATCGA
Longest Repeated Substring in banana$ is: ana
Longest Repeated Substring in abcPqrabPpq$ is: ab
Longest Repeated Substring in pqrpqpqabab$ is: ab

```

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1st lexicographically.

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that finding deepest node will take O(N).

So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text
2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 Substring Check](#)
- [Suffix Tree Application 2 Searching All Patterns](#)
- [Suffix Tree Application 4 Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Suffix Tree Application 6 Longest Palindromic Substring](#)

Suffix Tree Application 6 Longest Palindromic Substring

Given a string, find the longest substring which is palindrome.

We have already discussed Nave [$O(n^3)$], quadratic [$O(n^2)$] and linear [$O(n)$] approaches in [Set 1](#), [Set 2](#) and [Manachers Algorithm](#).

In this article, we will discuss another linear time approach based on suffix tree.

If given string is S, then approach is following:

- Reverse the string S (say reversed string is R)
- Get [Longest Common Substring](#) of S and R **given that LCS in S and R must be from same position in S**

Can you see why we say that **LCS in R and S must be from same position in S** ?

Lets look at following examples:

- For $S = xababaz$ and $R = zybababax$, LCS and LPS both are ababa (SAME)
- For $S = abacdfgdcaba$ and $R = abacdgcaba$, LCS is abacd and LPS is aba (DIFFERENT)
- For $S = pqrqpabcdfgdcba$ and $R = abcdgfcbapqrqp$, LCS and LPS both are pqrqp (SAME)
- For $S = pqqpabcdfghfdcba$ and $R = abcdfhgfdcbapqqp$, LCS is abcdf and LPS is pqqp (DIFFERENT)

We can see that LCS and LPS are not same always. When they are different ?

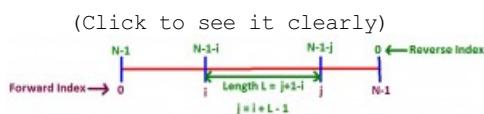
When S has a reversed copy of a non-palindromic substring in it which is of same or longer length than LPS in S, then LCS and LPS will be different.

In 2nd example above ($S = abacdfgdcaba$), for substring abacd, there exists a reverse copy dcaba in S, which is of longer length than LPS aba and so LPS and LCS are different here. Same is the scenario in 4th example.

To handle this scenario we say that LPS in S is same as LCS in S and R **given that LCS in R and S must be from same position in S**.

If we look at 2nd example again, substring aba in R comes from exactly same position in S as substring aba in S which is ZERO (0th index) and so this is LPS.

The Position Constraint:



We will refer string S index as forward index (S_i) and string R index as reverse index (R_j).

Based on above figure, a character with index i (forward index) in a string S of length N, will be at index $N-1-i$ (reverse index) in its reversed string R.

If we take a substring of length L in string S with starting index i and ending index j ($j = i+L-1$), then in its reversed string R, the reversed substring of the same will start at index $N-1-j$ and will end at index $N-1-i$.

If there is a common substring of length L at indices S_i (forward index) and R_j (reverse index) in S and R, then these will come from same position in S if $R_j = (N-1)(S_i + L-1)$ where N is string length.

So to find LPS of string S, we find longest common string of S and R where both substrings satisfy above constraint, i.e. if substring in S is at index S_i , then same substring should be in R at index $(N-1)(S_i + L-1)$. If this is not the case, then this substring is not LPS candidate.

Naive [$O(N*M^2)$] and Dynamic Programming [$O(N*M)$] approaches to find LCS of two strings are already discussed [here](#) which can be extended to add position constraint to give LPS of a given string.

Now we will discuss suffix tree approach which is nothing but an extension to [Suffix Tree LCS approach](#) where we will add the position constraint.

While finding LCS of two strings X and Y, we just take deepest node marked as XY (i.e. the node which has suffixes from both strings as its children).

While finding LPS of string S, we will again find LCS of S and R with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in S). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

In [Generalized Suffix Tree](#) of $S\#R\$$, a substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings S and R. The index of the common substring in S and R can be found by looking at suffix index at respective leaf node.

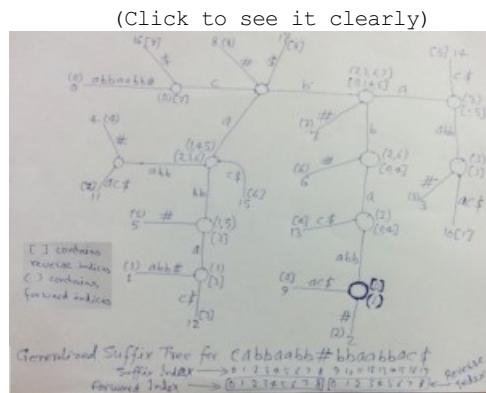
If string $S\#$ is of length N then:

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than N, then that suffix belongs to R and reverse index for all ancestor nodes will be **N suffix index**

Lets take string $S = cabbaabb$. The figure below is [Generalized Suffix Tree](#) for $cabbaabb\#bbaabbac\$$ where we have shown forward and

reverse indices of all children suffixes on all internal nodes (except root).

Forward indices are in Parentheses () and reverse indices are in square bracket [].



In above figure, all leaf nodes will have one forward or reverse index depending on which string (S or R) they belong to. Then childrens forward or reverse indices propagate to the parent.

Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in S and leaves with suffix indices 9 to 17 will get reverse index in R from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in S and so its forward index is 2 and shown in (). Leaf with suffix index 9 is from position 0 in R and so its reverse index is 0 and shown in []. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is [0,4]. And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying position constraint. For this, we need to check if there is a forward index S_i on a node, then there must be a reverse index R_i with value $(N - 2)(S_i + L - 1)$ where N is length of string $S\#$ and L is node depth (or substring length). If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn't represent any common substring (In code implementation also, forward and reverse indices will not be calculated on root node)

How to implement this approach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need to know if reverse index $R_i = (N - 2)(S_i + L - 1)$ also present on same node.
- Keep track of deepest internal node satisfying above condition.

One way to do above is:

While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices.

What data structure is suitable here to do all these in quickest way?

- If we store indices in array, it will require linear search which will make overall approach non-linear in time.
- If we store indices in tree (set in C++, TreeSet in Java), we may use binary search but still overall approach will be non-linear in time.
- If we store indices in hash function based set (unordered_set in C++, HashSet in Java), it will provide a constant search on average and this will make overall approach linear in time. *A hash function based set may take more space depending on values being stored.*

We will use two unordered_set (one for forward and other from reverse indices) in our implementation, added as a member variable in SuffixTreeNode structure.

```
// A C++ program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for given string S
// and it's reverse R, then we find
// longest palindromic substring of given string S
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <unordered_set>
#define MAX_CHAR 256
using namespace std;

struct SuffixTreeNode {
```

```

struct SuffixTreeNode *children[MAX_CHAR];

//pointer to other node via suffix link
struct SuffixTreeNode *suffixLink;

/*(start, end) interval specifies the edge, by which the
node is connected to its parent node. Each edge will
connect two nodes, one parent and one child, and
(start, end) interval of a given edge will be stored
in the child node. Lets say there are two nods A and B
connected by an edge with indices (5, 8) then this
indices (5, 8) will be stored in node B. */
int start;
int *end;

/*for leaf nodes, it stores the index of suffix for
the path from root to leaf*/
int suffixIndex;

//To store indices of children suffixes in given string
unordered_set<int> *forwardIndices;

//To store indices of children suffixes in reversed string
unordered_set<int> *reverseIndices;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase.*/
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    node->forwardIndices = new unordered_set<int>;
    node->reverseIndices = new unordered_set<int>;
    return node;
}

```

```

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from an existing node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]] ;
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
             is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])

```

```

{
    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node

```

```

{
    //Print the label on edge from parent to current node
//Uncomment below line to print suffix tree
    //printf(n->start, *(n->end));
}
int leaf = 1;
int i;
for (i = 0; i < MAX_CHAR; i++)
{
    if (n->children[i] != NULL)
    {
//Uncomment below two lines to print suffix index
        // if (leaf == 1 && n->start != -1)
        //     printf(" [%d]\n", n->suffixIndex);

        //Current node is not a leaf as it has outgoing
        //edges from it.
        leaf = 0;
        setSuffixIndexByDFS(n->children[i], labelHeight +
                           edgeLength(n->children[i]));
    }
    if (n != root)
    {
        //Add children's suffix indices in parent
        n->forwardIndices->insert(
            n->children[i]->forwardIndices->begin(),
            n->children[i]->forwardIndices->end());
        n->reverseIndices->insert(
            n->children[i]->reverseIndices->begin(),
            n->children[i]->reverseIndices->end());
    }
}
if (leaf == 1)
{
for(i= n->start; i<= * (n->end); i++)
{
    if(text[i] == '#')
    {
        n->end = (int*) malloc(sizeof(int));
        *(n->end) = i;
    }
}
n->suffixIndex = size - labelHeight;

if(n->suffixIndex < size) //Suffix of Given String
    n->forwardIndices->insert(n->suffixIndex);
else //Suffix of Reversed String
    n->reverseIndices->insert(n->suffixIndex - size);

//Uncomment below line to print suffix index
    // printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;
}

```

```

/*Root is a special node with start and end indices as -1,
as it has no parent from where an edge comes to root*/
root = newNode(-1, rootEnd);

activeNode = root; //First activeNode will be root
for (i=0; i<size; i++)
    extendSuffixTree(i);
int labelHeight = 0;
setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringstartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                edgeLength(n->children[i]),
                maxHeight, substringstartIndex);

                if(*maxHeight < labelHeight
                && n->forwardIndices->size() > 0 &&
                n->reverseIndices->size() > 0)
                {
                    for (forwardIndex=n->forwardIndices->begin();
                        forwardIndex!=n->forwardIndices->end();
                        ++forwardIndex)
                    {
                        reverseIndex = (size1 - 2) -
                        (*forwardIndex + labelHeight - 1);
                        //If reverse suffix comes from
                        //SAME position in given string
                        //Keep track of deepest node
                        if(n->reverseIndices->find(reverseIndex) !=
                        n->reverseIndices->end())
                        {
                            *maxHeight = labelHeight;
                            *substringstartIndex = *(n->end) -
                            labelHeight + 1;
                            break;
                        }
                    }
                }
            }
        }
    }
}

void getLongestPalindromicSubstring()
{
    int maxHeight = 0;
    int substringstartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringstartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringstartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d", maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$");
    buildSuffixTree();
    getLongestPalindromicSubstring();
}

```

```

//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 17;
printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
strcpy(text, "forgeeksskeegfor#rofgeeksskeegrof$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abcde is: ");
strcpy(text, "abcde#edcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 7;
printf("Longest Palindromic Substring in abcdae is: ");
strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abacd is: ");
strcpy(text, "abacd#dcaba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abcdc is: ");
strcpy(text, "abcdc#cdcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 13;
printf("Longest Palindromic Substring in abacdfgdcab is: ");
strcpy(text, "abacdfgdcab#abacdgcab$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 15;
printf("Longest Palindromic Substring in xyabacdfgdcab is: ");
strcpy(text, "xyabacdfgdcab#abacdgcabayx$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 9;
printf("Longest Palindromic Substring in xababayz is: ");
strcpy(text, "xababayz#zybabax$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in xabax is: ");
strcpy(text, "xabax#xabax$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}

```

Output:

```

Longest Palindromic Substring in cabbaabb is: bbaabb, of length: 6
Longest Palindromic Substring in forgeeksskeegfor is: geeksskeeg, of length: 10
Longest Palindromic Substring in abcde is: a, of length: 1
Longest Palindromic Substring in abcdae is: a, of length: 1
Longest Palindromic Substring in abacd is: aba, of length: 3
Longest Palindromic Substring in abcdc is: cdc, of length: 3
Longest Palindromic Substring in abacdfgdcab is: aba, of length: 3
Longest Palindromic Substring in xyabacdfgdcab is: aba, of length: 3
Longest Palindromic Substring in xababayz is: ababa, of length: 5
Longest Palindromic Substring in xabax is: xabax, of length: 5

```

Followup:

Detect ALL palindromes in a given string.

e.g. For string abcdccbefgf, all possible palindromes are a, b, c, d, e, f, g, dd, fgf, cddc, bcddcb.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 Substring Check](#)
- [Suffix Tree Application 2 Searching All Patterns](#)
- [Suffix Tree Application 3 Longest Repeated Substring](#)
- [Suffix Tree Application 4 Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 Longest Common Substring](#)
- [Generalized Suffix Tree 1](#)

AVL Tree | Set 1 (Insertion)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

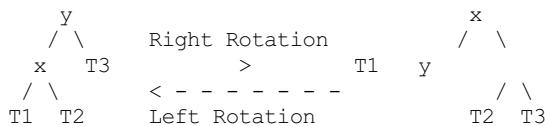
Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of an AVL tree is always $O(\log n)$ where n is the number of nodes in the tree (See [this](#) video lecture for proof).

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys(left)} < \text{key(root)} < \text{keys(right)}$). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)



Keys in both of the above trees follow the following order
 $\text{keys(T1)} < \text{key(x)} < \text{keys(T2)} < \text{key(y)} < \text{keys(T3)}$
So BST property is not violated anywhere.

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

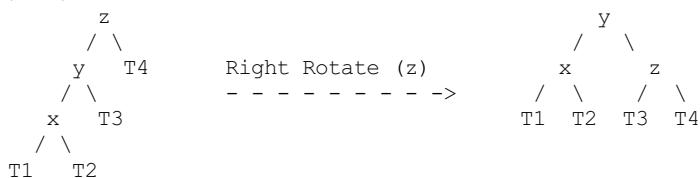
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

- a) y is left child of z and x is left child of y (Left Left Case)
- b) y is left child of z and x is right child of y (Left Right Case)
- c) y is right child of z and x is right child of y (Right Right Case)
- d) y is right child of z and x is left child of y (Right Left Case)

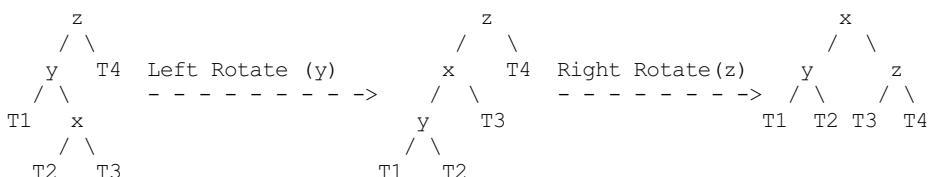
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See [this](#) video lecture for proof)

a) Left Left Case

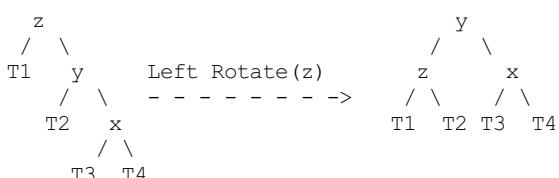
T1, T2, T3 and T4 are subtrees.



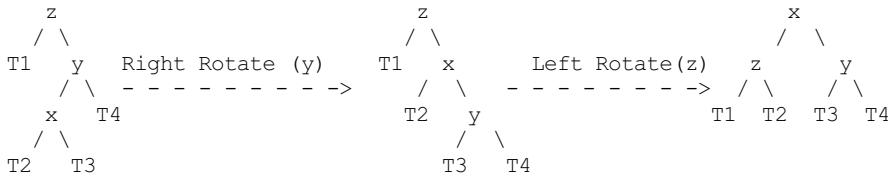
b) Left Right Case



c) Right Right Case



d) Right Left Case



C implementation

Following is the C implementation for AVL Tree Insertion. The following C implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in bottom up manner. So we dont need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

- 1) Perform the normal BST insertion.
- 2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

```
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;
}
```

```

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return (newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)

```

```

{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
            30
           /   \
          20   40
         / \   \
        10 25  50
    */
    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    return 0;
}

```

Output:

Pre order traversal of the constructed AVL tree is
 30 20 10 25 40 50

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL insert is $O(\log n)$.

The AVL tree and other self balancing search trees like Red Black are useful to get all basic operations done in $O(\log n)$ time. The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

Following is the post for delete.

[AVL Tree | Set 2 \(Deletion\)](#)

Following are some previous posts that have used self-balancing search trees.

[Median in a stream of integers \(running integers\)](#)

[Maximum of all subarrays of size k](#)

[Count smaller elements on right side](#)

References:

[IITD Video Lecture on AVL Tree Introduction](#)

[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

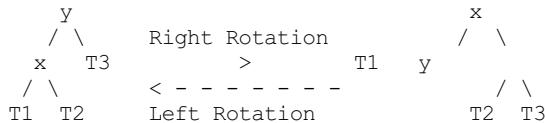
AVL Tree | Set 2 (Deletion)

We have discussed AVL insertion in the [previous post](#). In this post, we will follow a similar approach for deletion.

Steps to follow for deletion.

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys(left)} < \text{key(root)} < \text{keys(right)}$). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)



Keys in both of the above trees follow the following order

$\text{keys(T1)} < \text{key(x)} < \text{keys(T2)} < \text{key(y)} < \text{keys(T3)}$

So BST property is not violated anywhere.

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from [insertion](#) here.

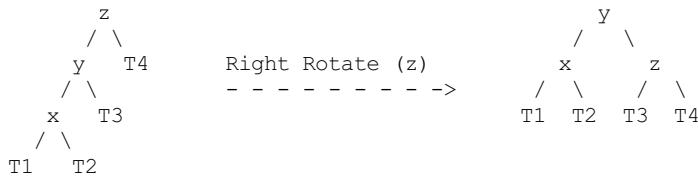
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

- a) y is left child of z and x is left child of y (Left Left Case)
- b) y is left child of z and x is right child of y (Left Right Case)
- c) y is right child of z and x is right child of y (Right Right Case)
- d) y is right child of z and x is left child of y (Right Left Case)

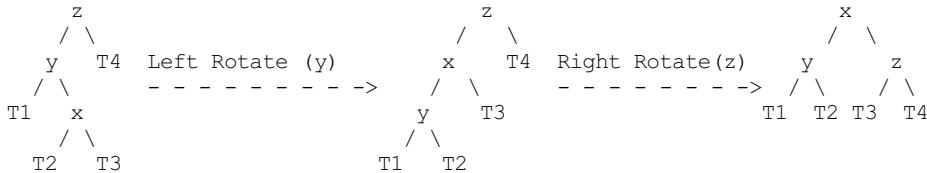
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z wont fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See [this video lecture](#) for proof)

a) Left Left Case

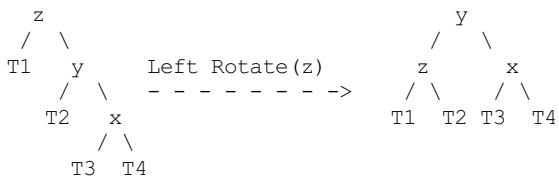
T1, T2, T3 and T4 are subtrees.



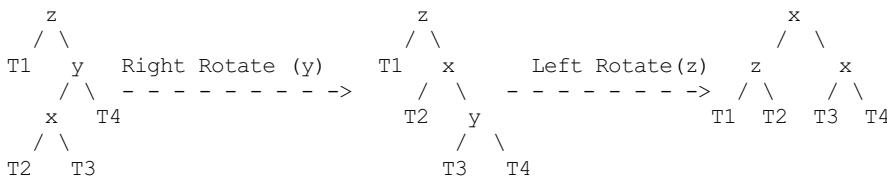
b) Left Right Case



c) Right Right Case



d) Right Left Case



Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

C implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we dont need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

- 1) Perform the normal BST deletion.
- 2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

C

```
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;
}
```

```

// Return new root
return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)

```

```

{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) )
        {
            struct node *temp = root->left ? root->left : root->right;

            // No child case
            if(temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        }
        else
        {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = max(height(root->left), height(root->right)) + 1;

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
    // this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {

```

```

        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    /* The constructed AVL Tree would be
         9
        / \
       1   10
      / \   \
     0   5   11
    / \   \
   -1   2   6
    */
    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    root = deleteNode(root, 10);

    /* The AVL Tree after deletion of 10
         1
        / \
       0   9
      / \   \
     -1   5   11
        / \
       2   6
    */
    printf("\nPre order traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}

```

Java

```
// Java program for deletion in AVL Tree
```

```

class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}

class AVLTree {
    static Node root;

    // A utility function to get height of the tree
    int height(Node N) {
        if (N == null) {
            return 0;
        }
        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b) {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        // Return new root
        return y;
    }

    // Get Balance factor of node N
    int getBalance(Node N) {
        if (N == null) {
            return 0;
        }
        return height(N.left) - height(N.right);
    }

    Node insert(Node node, int key) {
        /* 1. Perform the normal BST rotation */
        if (node == null) {
            return (new Node(key));
        }

        if (key < node.key) {

```

```

        node.left = insert(node.left, key);
    } else {
        node.right = insert(node.right, key);
    }

    /* 2. Update height of this ancestor node */
    node.height = max(height(node.left), height(node.right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases
    // Left Left Case
    if (balance > 1 && key < node.left.key) {
        return rightRotate(node);
    }

    // Right Right Case
    if (balance < -1 && key > node.right.key) {
        return leftRotate(node);
    }

    // Left Right Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
Node minValueNode(Node node) {
    Node current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null) {
        current = current.left;
    }

    return current;
}

Node deleteNode(Node root, int key) {

    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == null) {
        return root;
    }

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root.key) {
        root.left = deleteNode(root.left, key);
    }

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root.key) {
        root.right = deleteNode(root.right, key);
    }

    // if key is same as root's key, then this is the node
    // to be deleted
    else {

        // node with only one child or no child
        if ((root.left == null) || (root.right == null)) {
            Node temp = null;
            if (temp == root.left) {
                temp = root.right;
            }
        }
    }
}

```

```

        } else {
            temp = root.left;
        }

        // No child case
        if (temp == null) {
            temp = root;
            root = null;
        } else // One child case
        {
            root = temp; // Copy the contents of the non-empty child
        }
    } else {

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        Node temp = minValueNode(root.right);

        // Copy the inorder successor's data to this node
        root.key = temp.key;

        // Delete the inorder successor
        root.right = deleteNode(root.right, temp.key);
    }
}

// If the tree had only one node then return
if (root == null) {
    return root;
}

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root.height = max(height(root.left), height(root.right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
// this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases
// Left Left Case
if (balance > 1 && getBalance(root.left) >= 0) {
    return rightRotate(root);
}

// Left Right Case
if (balance > 1 && getBalance(root.left) < 0) {
    root.left = leftRotate(root.left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalance(root.right) <= 0) {
    return leftRotate(root);
}

// Right Left Case
if (balance < -1 && getBalance(root.right) > 0) {
    root.right = rightRotate(root.right);
    return leftRotate(root);
}

return root;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

public static void main(String[] args) {
    AVLTree tree = new AVLTree();

    /* Constructing tree given in the above figure */
    root = tree.insert(root, 9);
    root = tree.insert(root, 5);
    root = tree.insert(root, 10);
}

```

```

root = tree.insert(root, 0);
root = tree.insert(root, 6);
root = tree.insert(root, 11);
root = tree.insert(root, -1);
root = tree.insert(root, 1);
root = tree.insert(root, 2);

/* The constructed AVL Tree would be
   9
   / \
  1   10
  / \   \
 0   5   11
 / \   /
-1   2   6
*/
System.out.println("The preorder traversal of constructed tree is : ");
tree.preOrder(root);

root = tree.deleteNode(root, 10);

/* The AVL Tree after deletion of 10
   1
   / \
  0   9
  /   / \
-1   5   11
 / \
2   6
*/
System.out.println("");
System.out.println("Preorder traversal after deletion of 10 :");
tree.preOrder(root);
}

}

// This code has been contributed by Mayank Jaiswal

```

Output:

```

Pre order traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Pre order traversal after deletion of 10
1 0 -1 9 5 2 6 11

```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Logn).

References:

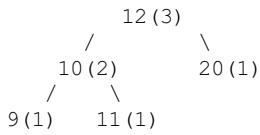
[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

AVL with duplicate keys

Please refer below post before reading about AVL tree handling of duplicates.

How to handle duplicates in Binary Search Tree?

The is to augment [AVL tree](#) node to store count together with regular fields like key, left and right pointers. Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.



Count of a key is shown in bracket

Below is C implementation of normal AVL Tree with count with every key. This code basically is taken from [code for insert and delete in AVL tree](#). The changes made for handling duplicates are highlighted, rest of the code is same.

The important thing to note is changes are very similar to simple Binary Search Tree changes.

```
// AVL tree that handles duplicates
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
    int count;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    node->count = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;
```

```

// Update heights
y->height = max(height(y->left), height(y->right))+1;
x->height = max(height(x->left), height(x->right))+1;

// Return new root
return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return (newNode(key));

    // If key already exists in BST, increment count and return
    if (key == node->key)
    {
        (node->count)++;
        return node;
    }

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {

```

```

        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

/* return the (unchanged) node pointer */
return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
        {
            (root->count)--;
            return;
        }
        // ELSE, delete the node

        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) )
        {
            struct node *temp = root->left ? root->left : root->right;

            // No child case
            if(temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        }
        else
        {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }
}

```

```

// If the tree had only one node then return
if (root == NULL)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = max(height(root->left), height(root->right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
// this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root->left) < 0)
{
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root->right) > 0)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}

return root;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d(%d) ", root->key, root->count);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 9);
    root = insert(root, 7);
    root = insert(root, 17);

    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    root = deleteNode(root, 9);

    printf("\nPre order traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}

```

Output:

```

Pre order traversal of the constructed AVL tree is
9(2) 5(2) 7(1) 10(1) 17(1)
Pre order traversal after deletion of 10

```

9(1) 5(2) 7(1) 10(1) 17(1)

[GATE Corner](#)[Quiz Corner](#)

Splay Tree | Set 1 (Search)

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with [AVL](#) and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?

Like [AVL](#) and Red-Black Trees, Splay tree is also [self-balancing BST](#). The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

Search Operation

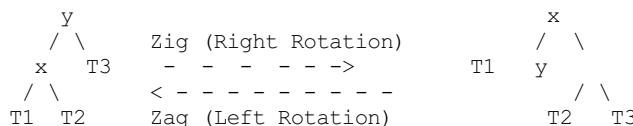
The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

1) Node is root We simply return the root, dont do anything else as the accessed node is already root.

2) Zig: Node is child of root (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).

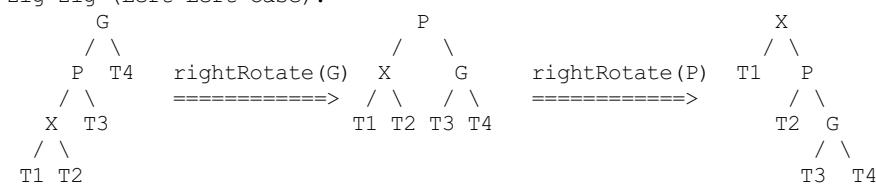
T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)



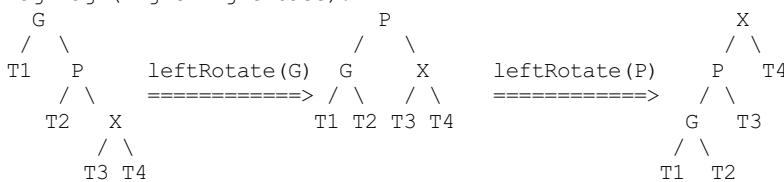
3) Node has both parent and grandparent. There can be following subcases.

.....**3.a) Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

Zig-Zig (Left Left Case) :

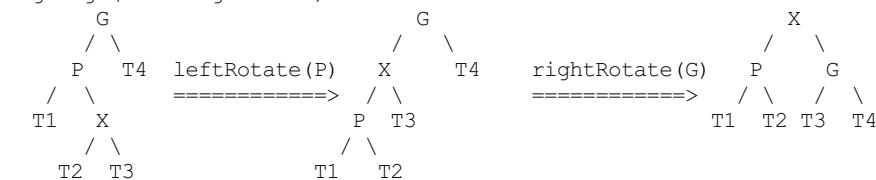


Zag-Zag (Right Right Case) :

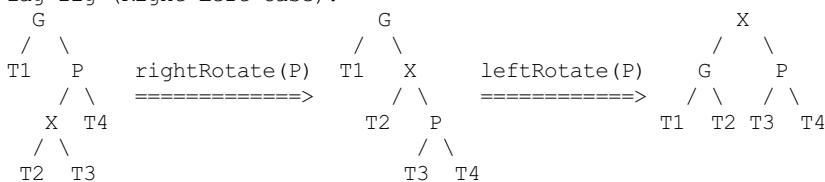


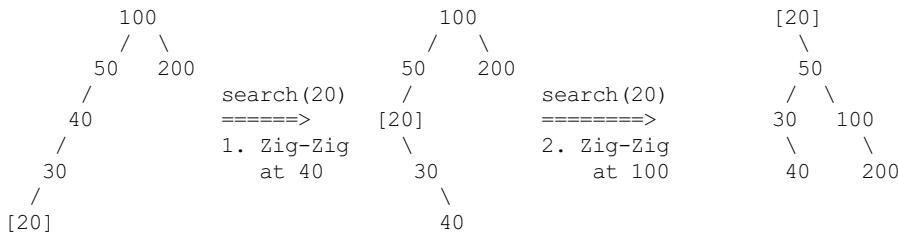
.....**3.b) Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

Zig-Zag (Left Right Case) :



Zag-Zig (Right Left Case) :



Example:

The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Implementation:

```
// The code is adopted from http://goo.gl/SDH9hH
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key    = key;
    node->left   = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);
        }
    }
}
```

```

        // Do first rotation for root, second rotation is done after else
        root = rightRotate(root);
    }
    else if (root->left->key < key) // Zig-Zag (Left Right)
    {
        // First recursively bring the key as root of left-right
        root->left->right = splay(root->left->right, key);

        // Do first rotation for root->left
        if (root->left->right != NULL)
            root->left = leftRotate(root->left);
    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}
else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;

    // Zag-Zig (Right Left)
    if (root->right->key > key)
    {
        // Bring the key as root of right-left
        root->right->left = splay(root->right->left, key);

        // Do first rotation for root->right
        if (root->right->left != NULL)
            root->right = rightRotate(root->right);
    }
    else if (root->right->key < key)// Zag-Zag (Right Right)
    {
        // Bring the key as root of right-right and do first rotation
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}
}

// The search function for Splay tree. Note that this function
// returns the new root of Splay Tree. If key is present in tree
// then, it is moved to root.
struct node *search(struct node *root, int key)
{
    return splay(root, key);
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);

    root = search(root, 20);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}

```

Output:

Preorder traversal of the modified Splay tree is
20 50 30 40 100 200

Summary

- 1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.
- 2) All splay tree operations take $O(\log n)$ time on average. Splay trees can be rigorously shown to run in $O(\log n)$ average time per operation, over any sequence of operations (assuming we start from an empty tree)
- 3) Splay trees are simpler compared to [AVL](#) and Red-Black Trees as no extra field is required in every tree node.
- 4) Unlike [AVL tree](#), a splay tree can change even with read-only operations like search.

Applications of Splay Trees

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.

Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software (Source: <http://www.cs.berkeley.edu/~jrs/61b/lec/36>)

See [Splay Tree | Set 2 \(Insert\)](#) for splay tree insertion.

References:

- <http://www.cs.berkeley.edu/~jrs/61b/lec/36>
- <http://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html>
- <http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>

Splay Tree | Set 2 (Insert)

It is recommended to refer following post as prerequisite of this post.

[Splay Tree | Set 1 \(Search\)](#)

As discussed in the [previous post](#), Splay tree is a self-balancing data structure where the last accessed key is always at root. The insert operation is similar to Binary Search Tree insert with additional steps to make sure that the newly inserted key becomes the new root.

Following are different cases to insert a key k in splay tree.

1) Root is NULL: We simply allocate a new node and return it as root.

2) Splay the given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.

3) If new roots key is same as k, dont do anything as k is already present.

4) Else allocate memory for new node and compare roots key with k.

.4.a) If k is smaller than roots key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.

.4.b) If k is greater than roots key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.

5) Return new node as new root of tree.

Example:

```
          100          [20]          25
         / \           \             / \
        50  200         50           20  50
       /   \           / \           /   \
      40   30         30  100       30  100
     /     \           \   \           \   \
    30     200         40  200       40  200
   /
[20]

// This code is adopted from http://algs4.cs.princeton.edu/33balanced/SplayBST.java.html
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key    = key;
    node->left   = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
```

```

}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}
else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;

    // Zig-Zag (Right Left)
    if (root->right->key > key)
    {
        // Bring the key as root of right-left
        root->right->left = splay(root->right->left, key);

        // Do first rotation for root->right
        if (root->right->left != NULL)
            root->right = rightRotate(root->right);
    }
    else if (root->right->key < key)// Zag-Zag (Right Right)
    {
        // Bring the key as root of right-right and do first rotation
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}
}

// Function to insert a new key k in splay tree with given root
struct node *insert(struct node *root, int k)
{
    // Simple Case: If tree is empty
    if (root == NULL) return newNode(k);

    // Bring the closest leaf node to root
    root = splay(root, k);

    // If key is already present, then return
    if (root->key == k) return root;

    // Otherwise allocate memory for new node
    struct node *newnode = newNode(k);
}

```

```

// If root's key is greater, make root as right child
// of newnode and copy the left child of root to newnode
if (root->key > k)
{
    newnode->right = root;
    newnode->left = root->left;
    root->left = NULL;
}

// If root's key is smaller, make root as left child
// of newnode and copy the right child of root to newnode
else
{
    newnode->left = root;
    newnode->right = root->right;
    root->right = NULL;
}

return newnode; // newnode becomes new root
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);
    root = insert(root, 25);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}

```

Output:

Preorder traversal of the modified Splay tree is
25 20 50 30 40 100 200

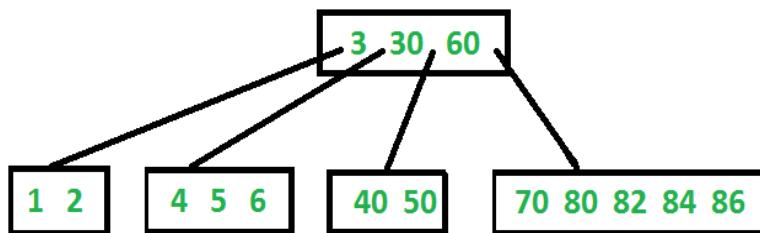
B-Tree | Set 1 (Introduction)

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like [AVL](#) and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree

- 1) All leaves are at same level.
- 2) A B-Tree is defined by the term *minimum degree t*. The value of t depends upon disk block size.
- 3) Every node except root must contain at least $t-1$ keys. Root may contain minimum 1 key.
- 4) All nodes (including root) may contain at most $2t-1$ keys.
- 5) Number of children of a node is equal to the number of keys in it plus 1.
- 6) All keys of a node are sorted in increasing order. The child between two keys k_1 and k_2 contains all keys in range from k_1 and k_2 .
- 7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- 8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is $O(\log n)$.

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k . We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

```
// C++ implementation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTREE node
class BTREENode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTREENode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTREENode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTREENode *search(int k); // returns NULL if k is not present.

    // Make BTREE friend of this so that we can access private members of this
    // class in BTREE functions
    friend class BTREE;
};

// A BTREE
class BTREE
{
    BTREENode *root; // Pointer to root node
    int t; // Minimum degree
```

```

public:
    // Constructor (Initializes tree as empty)
    BTTree(int _t)
    {   root = NULL;  t = _t; }

    // function to traverse the tree
    void traverse()
    {   if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTTreeNode* search(int k)
    {   return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTTreeNode class
BTTreeNode::BTTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTTreeNode *BTTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

```

The above code doesn't contain driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in [Cormen book](#)), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion

- [B-Tree Insertion](#)
- [B-Tree Deletion](#)

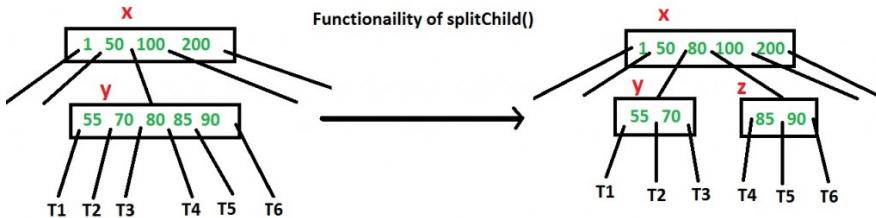
References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

B-Tree | Set 2 (Insert)

In the [previous post](#), we introduced B-Tree. We also discussed search() and traverse() functions.

In this post, insert() operation is discussed. A new key is always inserted at leaf node. Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space. *How to make sure that a node has space available for key before the key is inserted?* We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is complete algorithm.

Insertion

- 1) Initialize x as root.
- 2) While x is not leaf, do following
 - ..a) Find the child of x that is going to be traversed next. Let the child be y.
 - ..b) If y is not full, change x to point to y.
 - ..c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
- 3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we dont split a node before going down to it and split it only if new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree t as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Initially root is NULL. Let us first insert 10.

Insert 10

10

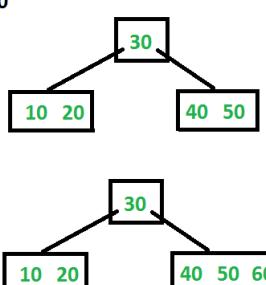
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is $2*t - 1$ which is 5.

Insert 20, 30, 40 and 50

10 20 30 40 50

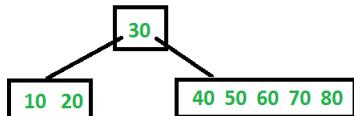
Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

Insert 60



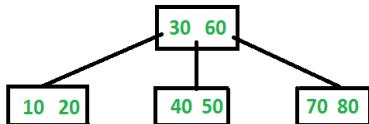
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90



See [this](#) for more examples.

Following is C++ implementation of the above proactive algorithm.

```
// C++ program for B-Tree insertion
#include<iostream>
using namespace std;

// A BTREE node
class BTREENode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTREENode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTREENode(int _t, bool _leaf); // Constructor

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of y in
    // child array C[]. The Child y must be full when this function is called
    void splitChild(int i, BTREENode *y);

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTREENode *search(int k); // returns NULL if k is not present.

    // Make BTREE friend of this so that we can access private members of this
    // class in BTREE functions
    friend class BTREE;
};

// A BTREE
class BTREE
{
    BTREENode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTREE(int _t)
    {   root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    {   if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTREENode* search(int k)
    {   return (root == NULL)? NULL : root->search(k); }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);
};

// Constructor for BTREENode class
BTREENode::BTREENode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
```

```

t = t1;
leaf = leaf1;

// Allocate memory for maximum number of possible keys
// and child pointers
keys = new int[2*t-1];
C = new BTreenode *[2*t];

// Initialize the number of keys as 0
n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreenode::traverse()
{
    // There are n keys and n+1 children, traverses through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreenode *BTreenode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTreenode::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreenode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreenode *s = new BTreenode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);

            // New root has two children now. Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)

```

```

        i++;
        s->C[i]->insertNonFull(k);

        // Change root
        root = s;
    }
    else // If root is not full, call insertNonFull for root
        root->insertNonFull(k);
}
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreenode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two. See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreenode::splitChild(int i, BTreenode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreenode *z = new BTreenode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
}

```

```

// create space of new child
for (int j = n; j >= i+1; j--)
    C[j+1] = C[j];

// Link the new child to this node
C[i+1] = z;

// A key of y will move to this node. Find location of
// new key and move all greater keys one space ahead
for (int j = n-1; j >= i; j--)
    keys[j+1] = keys[j];

// Copy the middle key of y to this node
keys[i] = y->keys[t-1];

// Increment count of keys in this node
n = n + 1;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
    t.insert(17);

    cout << "Traversal of the constucted tree is ";
    t.traverse();

    int k = 6;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    k = 15;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    return 0;
}

```

Output:

```

Traversal of the constucted tree is  5 6 7 10 12 17 20 30
Present
Not Present

```

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)
<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture17.html>

B-Tree | Set 3 (Delete)

It is recommended to refer following posts as prerequisite of this post.

[B-Tree | Set 1 \(Introduction\)](#)

[B-Tree | Set 2 \(Insert\)](#)

B-Tree is a type of a multi-way search tree. So, if you are not familiar with multi-way search trees in general, it is better to take a look at [this video lecture from IIT-Delhi](#), before proceeding further. Once you get the basics of a multi-way search tree clear, B-Tree operations will be easier to understand.

Source of the following explanation and algorithm is [Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

Deletion process:

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node-not just a leaf and when we delete a key from an internal node, we will have to rearrange the nodes children.

As in insertion, we must make sure the deletion doesn't violate the [B-tree properties](#). Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $t-1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x . This procedure guarantees that whenever it calls itself recursively on a node x , the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to back up (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b), then we delete x , and its only child $x.c_1$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

We sketch how deletion works with various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x .
2. If the key k is in node x and x is an internal node, do the following.

- a) If the child y that precedes k in node x has at least t keys, then find the predecessor k_0 of k in the sub-tree rooted at y . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)
- b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x . If z has at least t keys, then find the successor k_0 of k in the sub-tree rooted at z . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)
- c) Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y , so that x loses both k and the pointer to z , and y now contains $2t-1$ keys. Then free z and recursively delete k from y .

3. If the key k is not present in internal node x , determine the root $x.c(i)$ of the appropriate subtree that must contain k , if k is in the tree at all. If $x.c(i)$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x .

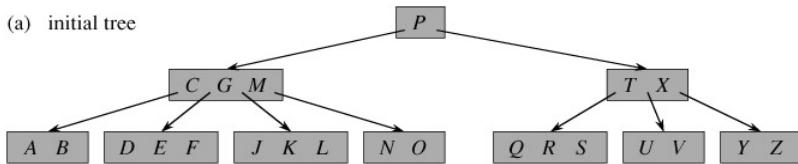
- a) If $x.c(i)$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c(i)$ an extra key by moving a key from x down into $x.c(i)$, moving a key from $x.c(i)$'s immediate left or right sibling up into x , and moving the appropriate child pointer from the sibling into $x.c(i)$.

- b) If $x.c(i)$ and both of $x.c(i)$'s immediate siblings have $t-1$ keys, merge $x.c(i)$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

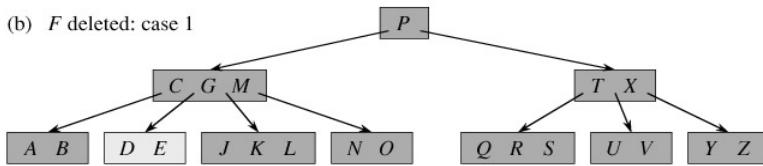
Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

The following figures from [CLRS book](#) explain the deletion process.

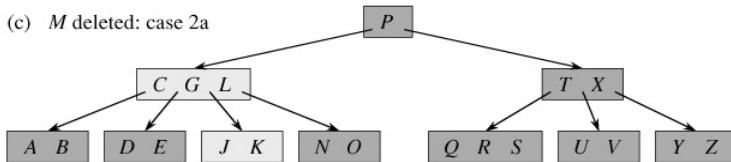
(a) initial tree



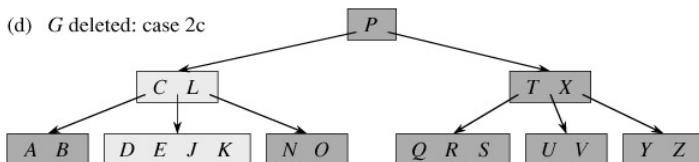
(b) F deleted: case 1



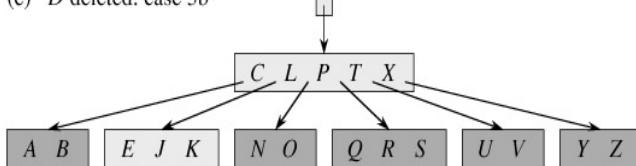
(c) M deleted: case 2a



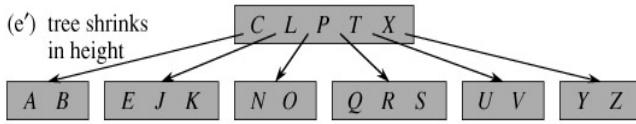
(d) G deleted: case 2c



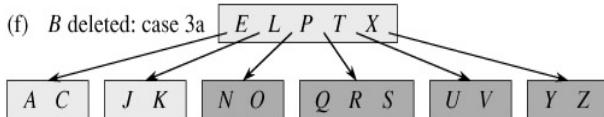
(e) D deleted: case 3b



(e') tree shrinks in height



(f) B deleted: case 3a



Implementation:

Following is C++ implementation of deletion process.

```
/*
The following program performs deletion on a B-Tree. It contains functions
specific for deletion along with all the other functions provided in the
previous articles on B-Trees. See http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/
for previous article.
```

The deletion function has been compartmentalized into 8 functions for ease of understanding and clarity

The following functions are exclusive for deletion

In class BTreenode:

- 1) remove
- 2) removeFromLeaf
- 3) removeFromNonLeaf
- 4) getPred
- 5) getSucc
- 6) borrowFromPrev
- 7) borrowFromNext
- 8) merge
- 9) findKey

In class BTree:

- 1) remove

The removal of a key from a B-Tree is a fairly complicated process. The program handles all the 6 different cases that might arise while removing a key.

Testing: The code has been tested using the B-Tree provided in the CLRS book(included in the main function) along with other cases.

Reference: CLRS3 - Chapter 18 - (499-502)
It is advised to read the material in CLRS before taking a look at the code. */

```
#include<iostream>
using namespace std;

// A BTree node
class BTreenode
{
    int *keys; // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTreenode **C; // An array of child pointers
    int n;      // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false

public:
    BTreenode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreenode *search(int k); // returns NULL if k is not present.

    // A function that returns the index of the first key that is greater
    // or equal to k
    int findKey(int k);

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index
    // of y in child array C[]. The Child y must be full when this
    // function is called
    void splitChild(int i, BTreenode *y);

    // A wrapper function to remove the key k in subtree rooted with
    // this node.
    void remove(int k);

    // A function to remove the key present in idx-th position in
    // this node which is a leaf
    void removeFromLeaf(int idx);

    // A function to remove the key present in idx-th position in
    // this node which is a non-leaf node
    void removeFromNonLeaf(int idx);

    // A function to get the predecessor of the key- where the key
    // is present in the idx-th position in the node
    int getPred(int idx);

    // A function to get the successor of the key- where the key
    // is present in the idx-th position in the node
    int getSucc(int idx);

    // A function to fill up the child node present in the idx-th
    // position in the C[] array if that child has less than t-1 keys
    void fill(int idx);

    // A function to borrow a key from the C[idx-1]-th node and place
    // it in C[idx]th node
    void borrowFromPrev(int idx);

    // A function to borrow a key from the C[idx+1]-th node and place it
    // in C[idx]th node
    void borrowFromNext(int idx);

    // A function to merge idx-th child of the node with (idx+1)th child of
    // the node
    void merge(int idx);
```

```

// Make BTree friend of this so that we can access private members of
// this class in BTree functions
friend class BTree;
};

class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:

    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {
        root = NULL;
        t = _t;
    }

    void traverse()
    {
        if (root != NULL) root->traverse();
    }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {
        return (root == NULL)? NULL : root->search(k);
    }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);

    // The main function that removes a new key in thie B-Tree
    void remove(int k);

};

BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreeNode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

// A function to remove the key k from the sub-tree rooted with this node
void BTreeNode::remove(int k)
{
    int idx = findKey(k);

    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {

        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else
    {

```

```

// If this node is a leaf node, then the key is not present in tree
if (leaf)
{
    cout << "The key "<< k << " is does not exist in the tree\n";
    return;
}

// The key to be removed is present in the sub-tree rooted with this node
// The flag indicates whether the key is present in the sub-tree rooted
// with the last child of this node
bool flag = ( (idx==n)? true : false );

// If the child where the key is supposed to exist has less than t keys,
// we fill that child
if (C[idx]->n < t)
    fill(idx);

// If the last child has been merged, it must have merged with the previous
// child and so we recurse on the (idx-1)th child. Else, we recurse on the
// (idx)th child which now has atleast t keys
if (flag && idx > n)
    C[idx-1]->remove(k);
else
    C[idx]->remove(k);
}
return;
}

// A function to remove the idx-th key from this node - which is a leaf node
void BTreenode::removeFromLeaf (int idx)
{
    // Move all the keys after the idx-th pos one place backward
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Reduce the count of keys
    n--;

    return;
}

// A function to remove the idx-th key from this node - which is a non-leaf node
void BTreenode::removeFromNonLeaf(int idx)
{
    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]->n >= t)
    {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    }

    // If the child C[idx] has less than t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in
    // the subtree rooted at C[idx+1]
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if (C[idx+1]->n >= t)
    {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx+1]->remove(succ);
    }

    // If both C[idx] and C[idx+1] has less than t keys, merge k and all of C[idx+1]
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else
    {
        merge(idx);
        C[idx]->remove(k);
    }
}

```

```

    return;
}

// A function to get predecessor of keys[idx]
int BTreenode::getPred(int idx)
{
    // Keep moving to the right most node until we reach a leaf
    BTreenode *cur=C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];

    // Return the last key of the leaf
    return cur->keys[cur->n-1];
}

int BTreenode::getSucc(int idx)
{
    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreenode *cur = C[idx+1];
    while (!cur->leaf)
        cur = cur->C[0];

    // Return the first key of the leaf
    return cur->keys[0];
}

// A function to fill child C[idx] which has less than t-1 keys
void BTreenode::fill(int idx)
{
    // If the previous child(C[idx-1]) has more than t-1 keys, borrow a key
    // from that child
    if (idx!=0 && C[idx-1]->n>=t)
        borrowFromPrev(idx);

    // If the next child(C[idx+1]) has more than t-1 keys, borrow a key
    // from that child
    else if (idx!=n && C[idx+1]->n>=t)
        borrowFromNext(idx);

    // Merge C[idx] with its sibling
    // If C[idx] is the last child, merge it with with its previous sibling
    // Otherwise merge it with its next sibling
    else
    {
        if (idx != n)
            merge(idx);
        else
            merge(idx-1);
    }
    return;
}

// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
void BTreenode::borrowFromPrev(int idx)
{
    BTreenode *child=C[idx];
    BTreenode *sibling=C[idx-1];

    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus, the loses
    // sibling one key and child gains one key

    // Moving all key in C[idx] one step ahead
    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];

    // If C[idx] is not a leaf, move all its child pointers one step ahead
    if (!child->leaf)
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

    // Setting child's first key equal to keys[idx-1] from the current node
    child->keys[0] = keys[idx-1];

    // Moving sibling's last child as C[idx]'s first child
}

```

```

if (!leaf)
    child->C[0] = sibling->C[sibling->n];

// Moving the key from the sibling to the parent
// This reduces the number of keys in the sibling
keys[idx-1] = sibling->keys[sibling->n-1];

child->n += 1;
sibling->n -= 1;

return;
}

// A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreenode::borrowFromNext(int idx)
{

BTreenode *child=C[idx];
BTreenode *sibling=C[idx+1];

// keys[idx] is inserted as the last key in C[idx]
child->keys[(child->n)] = keys[idx];

// Sibling's first child is inserted as the last child
// into C[idx]
if (!(child->leaf))
    child->C[(child->n)+1] = sibling->C[0];

//The first key from sibling is inserted into keys[idx]
keys[idx] = sibling->keys[0];

// Moving all keys in sibling one step behind
for (int i=1; i<sibling->n; ++i)
    sibling->keys[i-1] = sibling->keys[i];

// Moving the child pointers one step behind
if (!sibling->leaf)
{
    for(int i=1; i<=sibling->n; ++i)
        sibling->C[i-1] = sibling->C[i];
}

// Increasing and decreasing the key count of C[idx] and C[idx+1]
// respectively
child->n += 1;
sibling->n -= 1;

return;
}

// A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreenode::merge(int idx)
{
    BTreenode *child = C[idx];
    BTreenode *sibling = C[idx+1];

    // Pulling a key from the current node and inserting it into (t-1)th
    // position of C[idx]
    child->keys[t-1] = keys[idx];

    // Copying the keys from C[idx+1] to C[idx] at the end
    for (int i=0; i<sibling->n; ++i)
        child->keys[i+t] = sibling->keys[i];

    // Copying the child pointers from C[idx+1] to C[idx]
    if (!child->leaf)
    {
        for(int i=0; i<=sibling->n; ++i)
            child->C[i+t] = sibling->C[i];
    }

    // Moving all keys after idx in the current node one step before -
    // to fill the gap created by moving keys[idx] to C[idx]
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Moving the child pointers after (idx+1) in the current node one
    // step before
    for (int i=idx+2; i<=n; ++i)

```

```

C[i-1] = C[i];

// Updating the key count of child and the current node
child->n += sibling->n+1;
n--;

// Freeing the memory occupied by sibling
delete(sibling);
return;
}

// The main function that inserts a new key in this B-Tree
void BTREE::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTREENode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTREENode *s = new BTREENode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);

            // New root has two children now. Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);

            // Change root
            root = s;
        }
        else // If root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTREENode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;
    }
}

```

```

// See if the found child is full
if (C[i+1]->n == 2*t-1)
{
    // If the child is full, then split it
    splitChild(i+1, C[i+1]);

    // After split, the middle key of C[i] goes up and
    // C[i] is splitted into two. See which of the two
    // is going to have the new key
    if (keys[i+1] < k)
        i++;
}
C[i+1]->insertNonFull(k);
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreenode::splitChild(int i, BTreenode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreenode *z = new BTreenode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node
    n = n + 1;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreenode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreenode *BTreenode::search(int k)

```

```

{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

void BTTree::remove(int k)
{
    if (!root)
    {
        cout << "The tree is empty\n";
        return;
    }

    // Call the remove function for root
    root->remove(k);

    // If the root node has 0 keys, make its first child as the new root
    // if it has a child, otherwise set root as NULL
    if (root->n==0)
    {
        BTTreeNode *tmp = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];

        // Free the old root
        delete tmp;
    }
    return;
}

// Driver program to test above functions
int main()
{
    BTTree t(3); // A B-Tree with minium degree 3

    t.insert(1);
    t.insert(3);
    t.insert(7);
    t.insert(10);
    t.insert(11);
    t.insert(13);
    t.insert(14);
    t.insert(15);
    t.insert(18);
    t.insert(16);
    t.insert(19);
    t.insert(24);
    t.insert(25);
    t.insert(26);
    t.insert(21);
    t.insert(4);
    t.insert(5);
    t.insert(20);
    t.insert(22);
    t.insert(2);
    t.insert(17);
    t.insert(12);
    t.insert(6);

    cout << "Traversal of tree constructed is\n";
    t.traverse();
    cout << endl;

    t.remove(6);
    cout << "Traversal of tree after removing 6\n";
    t.traverse();
}

```

```

cout << endl;

t.remove(13);
cout << "Traversal of tree after removing 13\n";
t.traverse();
cout << endl;

t.remove(7);
cout << "Traversal of tree after removing 7\n";
t.traverse();
cout << endl;

t.remove(4);
cout << "Traversal of tree after removing 4\n";
t.traverse();
cout << endl;

t.remove(2);
cout << "Traversal of tree after removing 2\n";
t.traverse();
cout << endl;

t.remove(16);
cout << "Traversal of tree after removing 16\n";
t.traverse();
cout << endl;

return 0;
}

```

Output:

```

Traversal of tree constructed is
1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 6
1 2 3 4 5 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 13
1 2 3 4 5 7 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 7
1 2 3 4 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 4
1 2 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 2
1 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 16
1 3 5 10 11 12 14 15 17 18 19 20 21 22 24 25 26

```

Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

- 1 Find the sum of elements from index l to r where $0 \leq l \leq r \leq n-1$
- 2 Change value of a specified element of the array $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time.

Another solution is to create another array and store sum from start to i at the ith index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

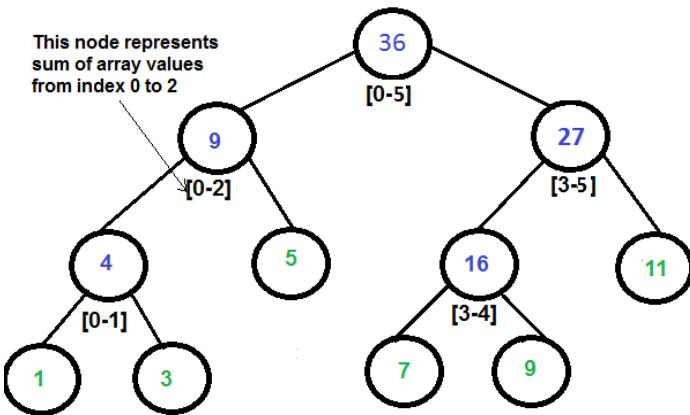
What if the number of query and updates are equal? **Can we perform both the operations in $O(\log n)$ time once given the array?** We can use a Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.

2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $\lfloor (i - 1)/2 \rfloor$.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Construction of Segment Tree from given array

We start with a segment $\text{arr}[0 \dots n-1]$. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a [Full Binary Tree](#) because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is algorithm to get the sum of elements.

```
int getSum(node, l, r)
{
    if range of node is within l and r
        return value in node
    else if range of node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
}
```

Update a value

Like tree construction and query operations, update can also be done recursively. We are given an index which needs to updated. Let $diff$ be the value to be added. We start from root of the segment tree, and add $diff$ to all nodes which have given index in their range. If a node doesn't have

given index in its range, we dont make any changes to that node.

Implementation:

Following is implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

C

```
// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the sum of values in given range
of the array. The following are parameters for this function.

st    --> Pointer to segment tree
si    --> Index of current node in the segment tree. Initially
        0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment represented
            by current node, i.e., st[si]
qs & qe  --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the sum of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
st, si, ss and se are same as getSumUtil()
i    --> index of the element to be updated. This index is
        in input array.
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];
```

```

// Update the value in array
arr[i] = new_val;

// Update the values of nodes in segment tree
updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start)
// to qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
              constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n",
           getSum(st, n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);
}

```

```

// Find sum after the value is updated
printf("Updated sum of values in given range = %d\n",
       getSum(st, n, 1, 3));
return 0;
}



## Java



// Java Program to show segment tree operations like construction,
// query and update
class SegmentTree
{
    int st[]; // The array that stores segment tree nodes

    /* Constructor to construct segment tree from given array. This
       constructor allocates memory for segment tree and calls
       constructSTUtil() to fill the allocated memory */
    SegmentTree(int arr[], int n)
    {
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

        //Maximum size of segment tree
        int max_size = 2 * (int) Math.pow(2, x) - 1;

        st = new int[max_size]; // Memory allocation

        constructSTUtil(arr, 0, n - 1, 0);
    }

    // A utility function to get the middle index from corner indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the sum of values in given range
       of the array. The following are parameters for this function.

       st      --> Pointer to segment tree
       si      --> Index of current node in the segment tree. Initially
                   0 is passed as root is always at index 0
       ss & se  --> Starting and ending indexes of the segment represented
                     by current node, i.e., st[si]
       qs & qe  --> Starting and ending indexes of query range */
    int getSumUtil(int ss, int se, int qs, int qe, int si)
    {
        // If segment of this node is a part of given range, then return
        // the sum of the segment
        if (qs <= ss && qe >= se)
            return st[si];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return 0;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
               getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
    }

    /* A recursive function to update the nodes which have the given
       index in their range. The following are parameters
       st, si, ss and se are same as getSumUtil()
       i      --> index of the element to be updated. This index is in
                   input array.
       diff --> Value to be added to all nodes which have i in range */
    void updateValueUtil(int ss, int se, int i, int diff, int si)
    {
        // Base Case: If the input index lies outside the range of
        // this segment
        if (i < ss || i > se)
            return;

        // If the input index is in range of this node, then update the
        // value of the node and its children
        st[si] = st[si] + diff;
        if (se != ss) {

```

```

        int mid = getMid(ss, se);
        updateValueUtil(ss, mid, i, diff, 2 * si + 1);
        updateValueUtil(mid + 1, se, i, diff, 2 * si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        System.out.println("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(0, n - 1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }
    return getSumUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, si * 2 + 1) +
             constructSTUtil(arr, mid + 1, se, si * 2 + 2);
    return st[si];
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    SegmentTree tree = new SegmentTree(arr, n);

    // Build segment tree from given array

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    tree.updateValue(arr, n, 1, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
                       tree.getSum(n, 1, 3));
}
}

//This code is contributed by Ankur Narain Verma

```

Sum of values in given range = 15
Updated sum of values in given range = 22

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\log n)$. To query a sum, we process at most four nodes at every level and number of levels is $O(\log n)$.

The time complexity of update is also $O(\log n)$. To update a leaf value, we process one node at every level and number of levels is $O(\log n)$.

Segment Tree | Set 2 (Range Minimum Query)

References:

<http://www.cse.iitk.ac.in/users/aca/lop12/slides/06.pdf>

Segment Tree | Set 2 (Range Minimum Query)

We have introduced [segment tree with a simple example](#) in the previous post. In this post, [Range Minimum Query](#) problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to efficiently find the minimum value from index qs (query start) to qe (query end) where $0 \leq qs \leq qe \leq n-1$. The array is static (elements are not deleted and inserted during the series of queries).

A **simple solution** is to run a loop from qs to qe and find minimum element in given range. This solution takes $O(n)$ time in worst case.

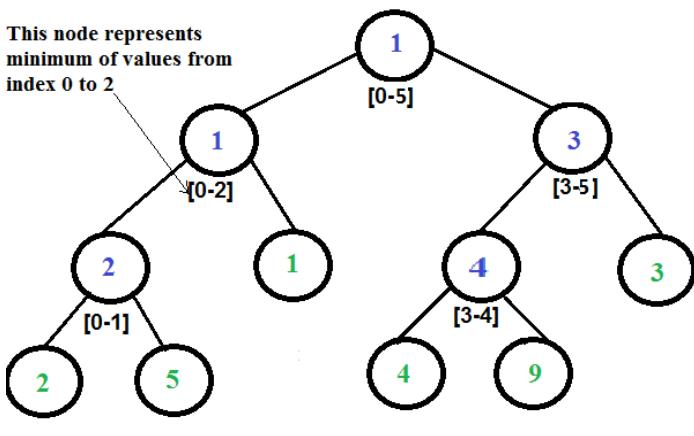
Another solution is to create a 2D array where an entry $[i, j]$ stores the minimum value in range $\text{arr}[i..j]$. Minimum of a given range can now be calculated in $O(1)$ time, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

[Segment tree](#) can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $\lfloor (i - 1)/2 \rfloor$.



Construction of Segment Tree from given array

We start with a segment $\text{arr}[0 \dots n-1]$, and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a [Full Binary Tree](#) because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for minimum value of given range

Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
    if range of node is within qs and qe
        return value in node
    else if range of node is completely outside qs and qe
        return INFINITE
    else
        return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs, qe) )
}
```

Implementation:

C

```

// C program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e -s)/2; }

/* A recursive function to get the minimum value in a given range
   of array indexes. The following are parameters for this function.

st    --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially
          0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment represented
               by current node, i.e., st[index]
qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return INT_MAX;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

// Return minimum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
                    constructSTUtil(arr, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

```

```

// Maximum size of segment tree
int max_size = 2*(int)pow(2, x) - 1;

int *st = new int[max_size];

// Fill the allocated memory st
constructSTUtil(arr, 0, n-1, st, 0);

// Return the constructed segment tree
return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    printf("Minimum of values in range [%d, %d] is = %d\n",
           qs, qe, RMQ(st, n, qs, qe));

    return 0;
}

```

Java

```

// Program for range minimum query using segment tree
class SegmentTreeRMQ
{
    int st[]; //array to store segment tree

    // A utility function to get minimum of two numbers
    int minVal(int x, int y) {
        return (x < y) ? x : y;
    }

    // A utility function to get the middle index from corner
    // indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the minimum value in a given
       range of array indexes. The following are parameters for
       this function.

       st --> Pointer to segment tree
       index --> Index of current node in the segment tree. Initially
                  0 is passed as root is always at index 0
       ss & se --> Starting and ending indexes of the segment
                  represented by current node, i.e., st[index]
       qs & qe --> Starting and ending indexes of query range */
    int RMQUtil(int ss, int se, int qs, int qe, int index)
    {
        // If segment of this node is a part of given range, then
        // return the min of the segment
        if (qs <= ss && qe >= se)
            return st[index];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return Integer.MAX_VALUE;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return minVal(RMQUtil(ss, mid, qs, qe, 2 * index + 1),
                      RMQUtil(mid + 1, se, qs, qe, 2 * index + 2));
    }

    // Return minimum of elements in range from index qs (quey
    // start) to qe (query end). It mainly uses RMQUtil()
    int RMQ(int n, int qs, int qe)
    {

```

```

// Check for erroneous input values
if (qs < 0 || qe > n - 1 || qs > qe) {
    System.out.println("Invalid Input");
    return -1;
}

return RMQUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current
    // node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, si * 2 + 1),
                    constructSTUtil(arr, mid + 1, se, si * 2 + 2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
void constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

    //Maximum size of segment tree
    int max_size = 2 * (int) Math.pow(2, x) - 1;
    st = new int[max_size]; // allocate memory

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = arr.length;
    SegmentTreeRMQ tree = new SegmentTreeRMQ();

    // Build segment tree from given array
    tree.constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    System.out.println("Minimum of values in range [" + qs + ", "
                       + qe + "] is = " + tree.RMQ(n, qs, qe));
}
}

// This code is contributed by Ankur Narain Verma

```

Minimum of values in range [1, 5] is = 2

Time Complexity:

Time Complexity for tree construction is O(n). There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is O(Logn). To query a range minimum, we process at most two nodes at every level and number of levels is O(Logn).

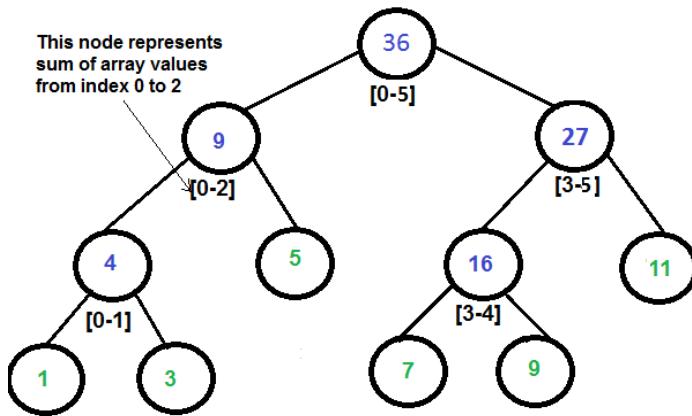
Please refer following links for more solutions to range minimum query problem.

[http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_\(RMQ\)](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))

http://wcipeg.com/wiki/Range_minimum_query

Lazy Propagation in Segment Tree

Segment tree is introduced in [previous post](#) with an example of range sum problem. We have used the same Sum of given Range problem to explain Lazy propagation



Segment Tree for input array {1, 3, 5, 7, 9, 11}

How does update work in Simple Segment Tree?

In the [previous post](#), update function was called to update only a single value in array. Please note that a single value update in array may cause multiple updates in Segment Tree as there may be many segment tree nodes that have a single array element in their ranges.

Below is simple logic used in previous post.

- 1) Start with root of segment tree.
- 2) If array index to be updated is not in current nodes range, then return
- 3) Else update current node and recur for children.

Below is code taken from previous post.

```
/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
   tree[] --> segment tree
   si      --> index of current node in segment tree.
              Initial value is passed as 0.
   ss and se --> Starting and ending indexes of array elements
                 covered under this node of segment tree.
                 Initial values passed as 0 and n-1.
   i      --> index of the element to be updated. This index
             is in input array.
   diff --> Value to be added to all nodes which have array
             index i in range */
void updateValueUtil(int tree[], int ss, int se, int i,
                     int diff, int si)
{
    // Base Case: If the input index lies outside the range
    // of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then
    // update the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}
```

What if there are updates on a range of array indexes?

For example add 10 to all values at indexes from 2 to 7 in array. The above update has to be called for every index from 2 to 7. We can avoid multiple calls by writing a function updateRange() that updates nodes accordingly.

```
/* Function to update segment tree for range update in input
   array.
   si -> index of current node in segment tree
   ss and se -> Starting and ending indexes of elements for
               which current nodes stores sum.
```

```

us and eu -> starting and ending indexes of update query
ue -> ending index of update query
diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                     int ue, int diff)
{
    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current node is a leaf node
    if (ss==se)
    {
        // Add the difference to current node
        tree[si] += diff;
        return;
    }

    // If not a leaf node, recur for children.
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // Use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

```

Lazy Propagation An optimization to make range updates faster

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

Please remember that a node in segment tree stores or represents result of a query for a range of indexes. And if this nodes range lies within the update operation range, then all descendants of the node must also be updated. For example consider the node with value 27 in above diagram, this node stores sum of values at indexes from 3 to 5. If our update query is for range 2 to 5, then we need to update this node and all descendants of this node. With Lazy propagation, we update only node with value 27 and postpone updates to its children by storing this update information in separate nodes called lazy nodes or values. We create an array `lazy[]` which represents lazy node. Size of `lazy[]` is same as array that represents segment tree, which is `tree[]` in below code.

The idea is to initialize all elements of `lazy[]` as 0. A value 0 in `lazy[i]` indicates that there are no pending updates on node i in segment tree. A non-zero value of `lazy[i]` means that this amount needs to be added to node i in segment tree before making any query to the node.

Below is modified update method.

```

// To update segment tree for change in array
// values at array indexes from us to ue.
updateRange(us, ue)
1) If current segment tree node has any pending
   update, then first add that pending update to
   current node.
2) If current node's range lies completely in
   update query range.
....a) Update current node
....b) Postpone updates to children by setting
       lazy value for children nodes.
3) If current node's range overlaps with update
   range, follow the same approach as above simple
   update.
....a) Recur for left and right children.
....b) Update current node using results of left
       and right calls.

```

Is there any change in Query Function also?

Since we have changed update to postpone its operations, there may be problems if a query is made to a node that is yet to be updated. So we need to update our query method also which is [getSumUtil in previous post](#). The `getSumUtil()` now first checks if there is a pending update and if there is, then updates the node. Once it makes sure that pending update is done, it works same as the previous `getSumUtil()`.

Below are programs to demonstrate working of Lazy Propagation.

C/C++

```

// Program to show segment tree to demonstrate lazy
// propagation
#include <stdio.h>
#include <math.h>

```

```

#define MAX 1000

// Ideally, we should not use global variables and large
// constant-sized arrays, we have done it here for simplicity.
int tree[MAX] = {0}; // To store segment tree
int lazy[MAX] = {0}; // To store pending updates

/* si -> index of current node in segment tree
   ss and se -> Starting and ending indexes of elements for
                  which current nodes stores sum.
   us and eu -> starting and ending indexes of update query
   ue -> ending index of update query
   diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                     int ue, int diff)
{
    // If lazy value is non-zero for current node of segment
    // tree, then there are some pending updates. So we need
    // to make sure that the pending updates are done before
    // making new updates. Because this value may be used by
    // parent after recursive calls (See last line of this
    // function)
    if (lazy[si] != 0)
    {
        // Make pending updates using value stored in lazy
        // nodes
        tree[si] += (se-ss+1)*lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children we don't
            // need their new values now.
            // Since we are not yet updating children of si,
            // we need to set lazy flags for the children
            lazy[si*2 + 1] += lazy[si];
            lazy[si*2 + 2] += lazy[si];
        }

        // Set the lazy value for current node as 0 as it
        // has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current segment is fully in range
    if (ss>=us && se<=ue)
    {
        // Add the difference to current node
        tree[si] += (se-ss+1)*diff;

        // same logic for checking leaf node or not
        if (ss != se)
        {
            // This is where we store values in lazy nodes,
            // rather than updating the segment tree itself
            // Since we don't need these updated values now
            // we postpone updates by storing values in lazy[]
            lazy[si*2 + 1] += diff;
            lazy[si*2 + 2] += diff;
        }
        return;
    }

    // If not completely in rang, but overlaps, recur for
    // children,
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // And use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

// Function to update a range of values in segment
// tree

```

```

/* us and eu -> starting and ending indexes of update query
ue -> ending index of update query
diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff)
{
    updateRangeUtil(0, 0, n-1, us, ue, diff);
}

/* A recursive function to get the sum of values in given
range of the array. The following are parameters for
this function.
si --> Index of current node in the segment tree.
    Initially 0 is passed as root is always at'
    index 0
ss & se --> Starting and ending indexes of the
            segment represented by current node,
            i.e., tree[si]
qs & qe --> Starting and ending indexes of query
            range */
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se-ss+1)*lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children os si,
            // we need to set lazy values for the children
            lazy[si*2+1] += lazy[si];
            lazy[si*2+2] += lazy[si];
        }

        // unset the lazy value for current node as it has
        // been updated
        lazy[si] = 0;
    }

    // Out of range
    if (ss>se || ss>qe || se<qs)
        return 0;

    // At this point we are sure that pending lazy updates
    // are done for current node. So we can return value
    // (same as it was for query in our previous post)

    // If this segment lies in range
    if (ss>=qs && se<=qe)
        return tree[si];

    // If a part of this segment overlaps with the given
    // range
    int mid = (ss + se)/2;
    return getSumUtil(ss, mid, qs, qe, 2*si+1) +
           getSumUtil(mid+1, se, qs, qe, 2*si+2);
}

// Return sum of elements in range from index qs (quey
// start) to qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n-1, qs, qe, 0);
}

```

```

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st.
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return ;

    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum
    // of values in this node
    int mid = (ss + se)/2;
    constructSTUtil(arr, ss, mid, si*2+1);
    constructSTUtil(arr, mid+1, se, si*2+2);

    tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, 0);
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n",
           getSum(n, 1, 3));

    // Add 10 to all nodes at indexes from 1 to 5.
    updateRange(n, 1, 5, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n",
           getSum( n, 1, 3));

    return 0;
}

```

Java

```

// Java program to demonstrate lazy propagation in segment tree
class LazySegmentTree
{
    final int MAX = 1000;          // Max tree size
    int tree[] = new int[MAX];    // To store segment tree
    int lazy[] = new int[MAX];    // To store pending updates

    /* si -> index of current node in segment tree
       ss and se -> Starting and ending indexes of elements for
                      which current nodes stores sum.
       us and eu -> starting and ending indexes of update query
       ue -> ending index of update query
       diff -> which we need to add in the range us to ue */
    void updateRangeUtil(int si, int ss, int se, int us,
                         int ue, int diff)
    {
        // If lazy value is non-zero for current node of segment
        // tree, then there are some pending updates. So we need

```

```

// to make sure that the pending updates are done before
// making new updates. Because this value may be used by
// parent after recursive calls (See last line of this
// function)
if (lazy[si] != 0)
{
    // Make pending updates using value stored in lazy
    // nodes
    tree[si] += (se - ss + 1) * lazy[si];

    // checking if it is not leaf node because if
    // it is leaf node then we cannot go further
    if (ss != se)
    {
        // We can postpone updating children we don't
        // need their new values now.
        // Since we are not yet updating children of si,
        // we need to set lazy flags for the children
        lazy[si * 2 + 1] += lazy[si];
        lazy[si * 2 + 2] += lazy[si];
    }

    // Set the lazy value for current node as 0 as it
    // has been updated
    lazy[si] = 0;
}

// out of range
if (ss > se || ss > ue || se < us)
    return;

// Current segment is fully in range
if (ss >= us && se <= ue)
{
    // Add the difference to current node
    tree[si] += (se - ss + 1) * diff;

    // same logic for checking leaf node or not
    if (ss != se)
    {
        // This is where we store values in lazy nodes,
        // rather than updating the segment tree itself
        // Since we don't need these updated values now
        // we postpone updates by storing values in lazy[]
        lazy[si * 2 + 1] += diff;
        lazy[si * 2 + 2] += diff;
    }
    return;
}

// If not completely in rang, but overlaps, recur for
// children,
int mid = (ss + se) / 2;
updateRangeUtil(si * 2 + 1, ss, mid, us, ue, diff);
updateRangeUtil(si * 2 + 2, mid + 1, se, us, ue, diff);

// And use the result of children calls to update this
// node
tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

// Function to update a range of values in segment
// tree
/* us and eu -> starting and ending indexes of update query
ue -> ending index of update query
diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff) {
    updateRangeUtil(0, 0, n - 1, us, ue, diff);
}

/* A recursive function to get the sum of values in given
range of the array. The following are parameters for
this function.
si --> Index of current node in the segment tree.
      Initially 0 is passed as root is always at'
      index 0
ss & se --> Starting and ending indexes of the
            segment represented by current node,
            i.e., tree[si]
qs & qe --> Starting and ending indexes of query
            range */

```

```

int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se - ss + 1) * lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children os si,
            // we need to set lazy values for the children
            lazy[si * 2 + 1] += lazy[si];
            lazy[si * 2 + 2] += lazy[si];
        }

        // unset the lazy value for current node as it has
        // been updated
        lazy[si] = 0;
    }

    // Out of range
    if (ss > se || ss > qe || se < qs)
        return 0;

    // At this point sure, pending lazy updates are done
    // for current node. So we can return value (same as
    // was for query in our previous post)

    // If this segment lies in range
    if (ss >= qs && se <= qe)
        return tree[si];

    // If a part of this segment overlaps with the given
    // range
    int mid = (ss + se) / 2;
    return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
        getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
}

// Return sum of elements in range from index qs (query
// start) to qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        System.out.println("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n - 1, qs, qe, 0);
}

/* A recursive function that constructs Segment Tree for
array[ss..se]. si is index of current node in segment
tree st. */
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return;

    /* If there is one element in array, store it in
    current node of segment tree and return */
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    /* If there are more than one elements, then recur
    for left and right subtrees and store the sum
    of values in this node */

```

```

int mid = (ss + se) / 2;
constructSTUtil(arr, ss, mid, si * 2 + 1);
constructSTUtil(arr, mid + 1, se, si * 2 + 2);

tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    LazySegmentTree tree = new LazySegmentTree();

    // Build segment tree from given array
    tree.constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Add 10 to all nodes at indexes from 1 to 5.
    tree.updateRange(n, 1, 5, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
                       tree.getSum(n, 1, 3));
}
}

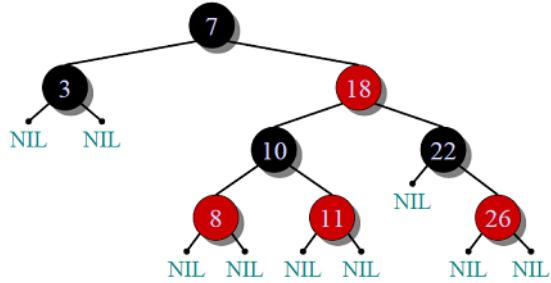
// This Code is contributed by Ankur Narain Verma

```

Sum of values in given range = 15
 Updated sum of values in given range = 45

Red-Black Tree | Set 1 (Introduction)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.



- 1) Every node has a color either red or black.
- 2) Root of tree is always black.
- 3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- 4) Every path from root to a NULL node has same number of black nodes.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

Comparison with AVL Tree

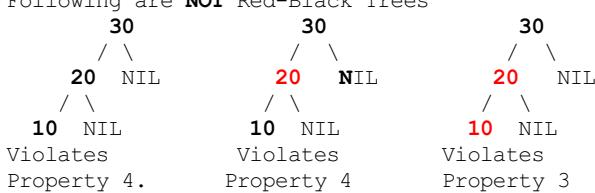
The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

How does a Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in red black tree. We can try any combination of colors and see all of them violate Red-Black tree property.

A chain of 3 nodes is nodes is not possible in Red-Black Trees.

Following are NOT Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Every Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

This can be proved using following facts:

- 1) For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k \leq \log_2(n+1)$
- 2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
- 3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lceil \frac{n}{2} \rceil$ where n is total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on Red-Black tree.

Exercise:

- 1) Is it possible to have all black nodes in a Red-Black tree?
- 2) Draw a Red-Black Tree that is not an [AVL tree](#) structure wise?

Insertion and Deletion

[Red Black Tree Insertion](#)

[Red-Black Tree Deletion](#)

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

http://en.wikipedia.org/wiki/Red%20-%20black_tree

[Video Lecture on Red-Black Tree by Tim Roughgarden](#)

[MIT Video Lecture on Red-Black Tree](#)

[MIT Lecture Notes on Red Black Tree](#)

Red-Black Tree | Set 2 (Insert)

In the [previous post](#), we discussed introduction to Red-Black Trees. In this post, insertion is discussed.

In [AVL tree insertion](#), we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

1) Recoloring

2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

1) Perform [standard BST insertion](#) and make the color of newly inserted nodes as RED.

2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

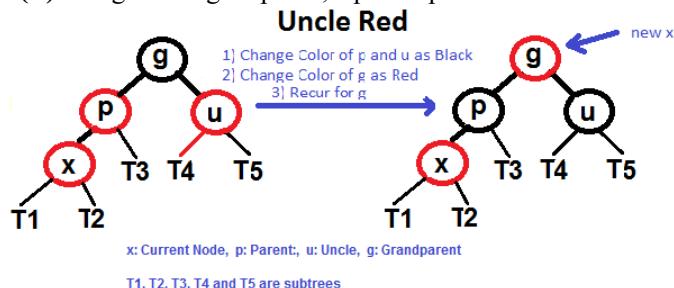
3) Do following if color of xs parent is not BLACK or x is not root.

.a) If xs uncle is RED (Grand parent must have been black from [property 4](#))

..(i) Change color of parent and uncle as BLACK.

..(ii) color of grand parent as RED.

..(iii) Change x = xs grandparent, repeat steps 2 and 3 for new x.



.b) If xs uncle is BLACK, then there can be four configurations for x, xs parent (p) and xs grandparent (g) (This is similar to [AVL Tree](#))

..i) Left Left Case (p is left child of g and x is left child of p)

..ii) Left Right Case (p is left child of g and x is right child of p)

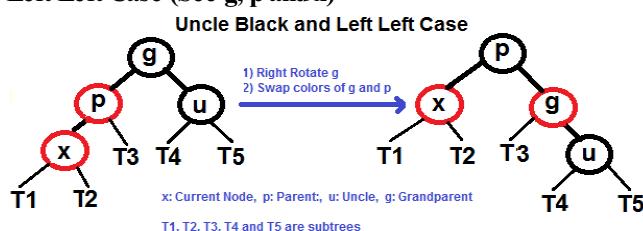
..iii) Right Right Case (Mirror of case a)

..iv) Right Left Case (Mirror of case c)

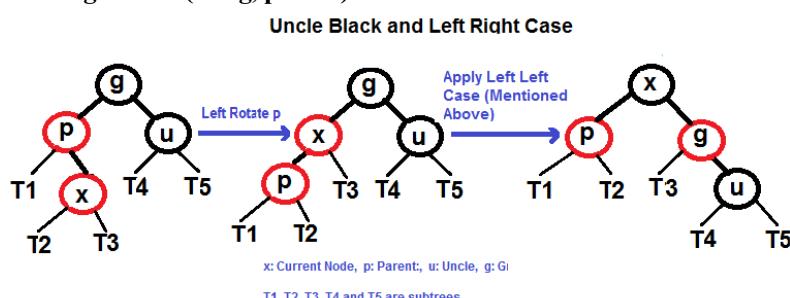
Following are operations to be performed in four subcases when uncle is BLACK.

All four cases when Uncle is BLACK

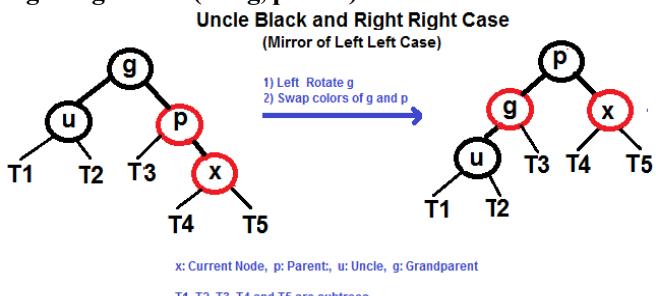
Left Left Case (See g, p and x)



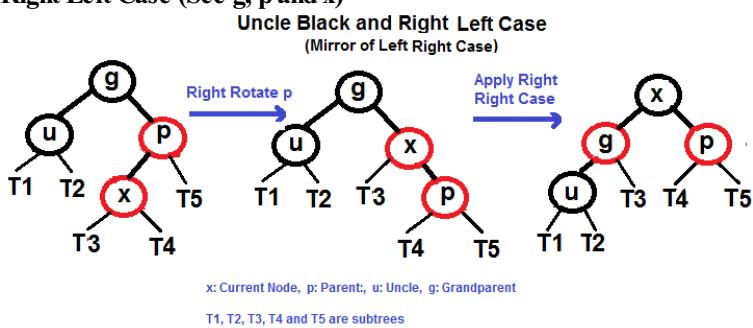
Left Right Case (See g, p and x)



Right Right Case (See g, p and x)

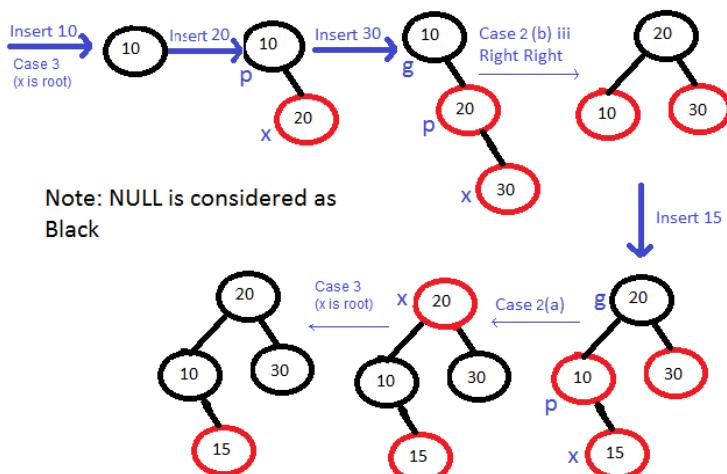


Right Left Case (See g, p and x)



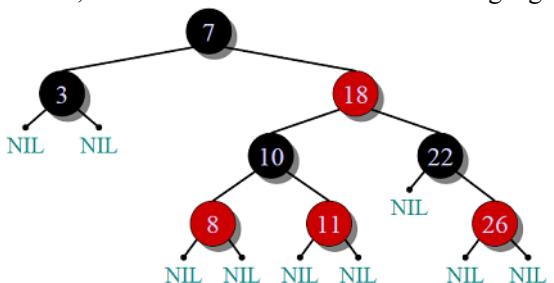
Examples of Insertion

Insert 10, 20, 30 and 15 in an empty tree



Exercise:

Insert 2, 7 and 13 in below tree. Insertion of 13 is going to be really interesting, try it to check if you have understood insertion well for exams.



Please refer [C Program for Red Black Tree Insertion](#) for complete implementation of above algorithm.

[Red-Black Tree | Set 3 \(Delete\)](#)

Red-Black Tree | Set 3 (Delete)

We have discussed following topics on Red-Black tree in previous posts. We strongly recommend to refer following post as prerequisite of this post.

[Red-Black Tree Introduction](#)

[Red Black Tree Insert](#)

Insertion Vs Deletion:

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, *we check color of sibling* to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

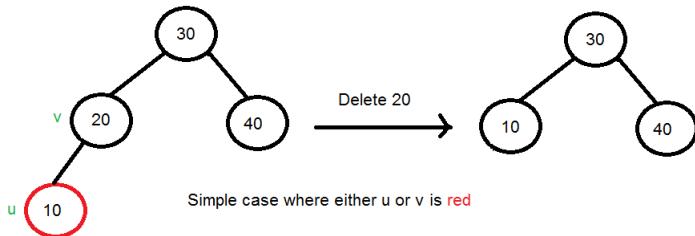
Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as **double black**. The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

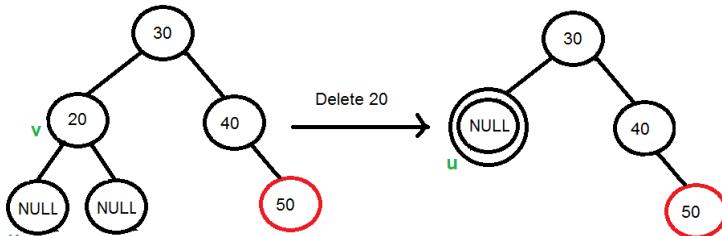
1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



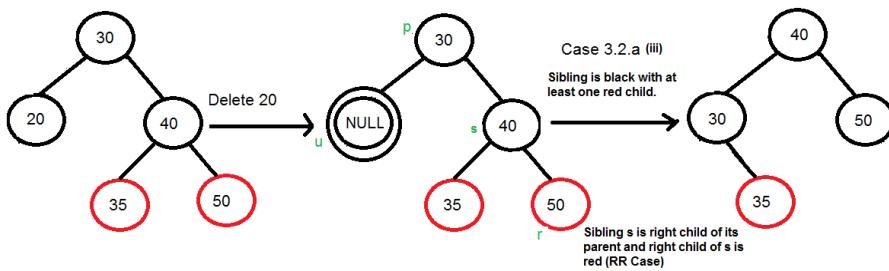
3.2) Do following while the current node u is double black or it is not root. Let sibling of node be s.

(a): If sibling s is black and at least one of siblings children is red, perform rotation(s). Let the red child of s be r. This case can be divided in four subcases depending upon positions of s and r.

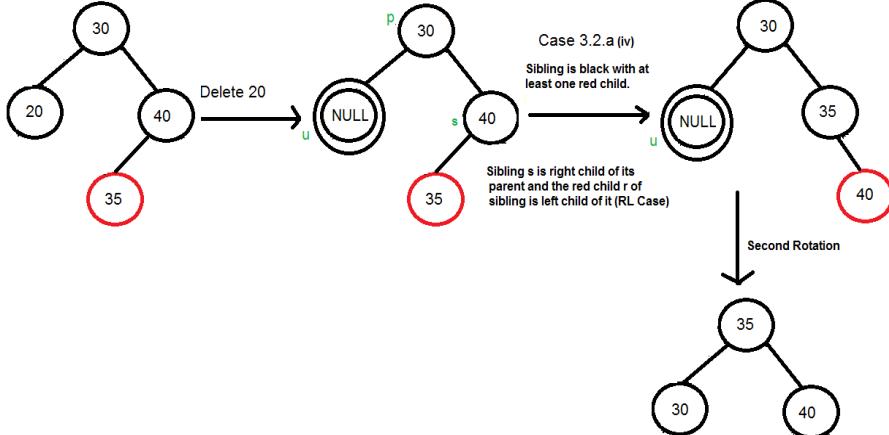
..(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

..(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

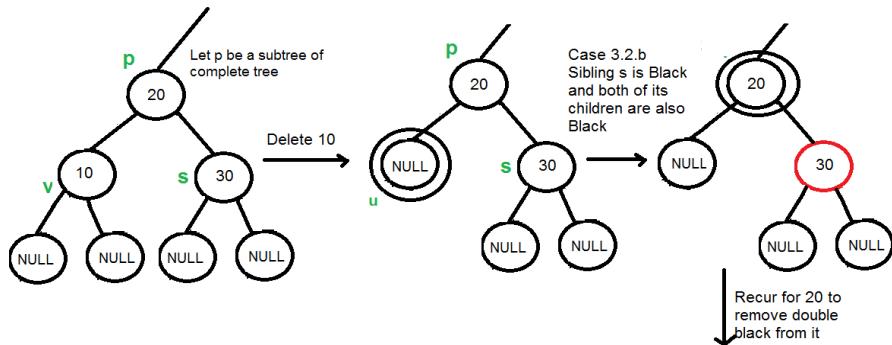
..(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



..(iv) Right Left Case (s is right child of its parent and r is left child of s)



..(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

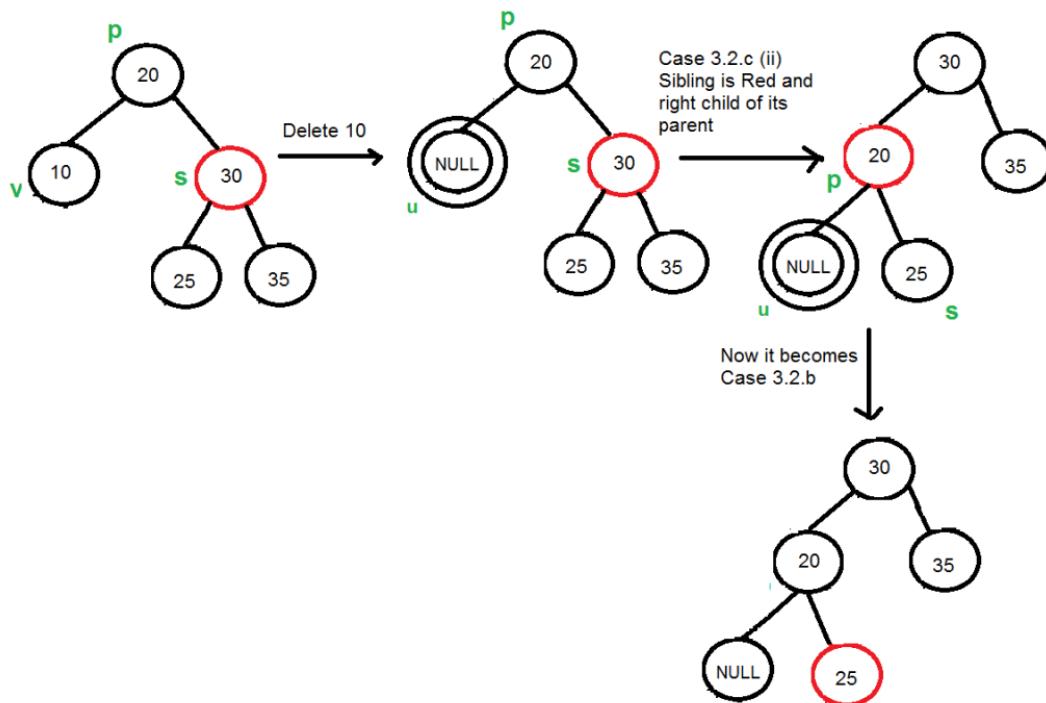


In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

..(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

..(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

..(iii) Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

References:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

C Program for Red Black Tree Insertion

We strongly recommend to go through following articles as a prerequisite for this.

[Red-Black Tree Introduction](#)

[Red Black Tree Insertion](#)

Following is the complete algorithm discussed in [Red Black Tree Insertion](#) article.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

1) Perform standard BST insertion and make the color of newly inserted nodes as RED.

2) Do following if color of xs parent is not BLACK or x is not root.

.a) If xs uncle is RED (Grand parent must have been black from property 4)

..(i) Change color of parent and uncle as BLACK.

..(ii) color of grand parent as RED.

..(iii) Change x = xs grandparent, repeat steps 2 and 3 for new x.

.b) If xs uncle is BLACK, then there can be four configurations for x, xs parent (p) and xs grandparent (g).

..i) Left Left Case (p is left child of g and x is left child of p)

..ii) Left Right Case (p is left child of g and x is right child of p)

..iii) Right Right Case (Mirror of case a)

..iv) Right Left Case (Mirror of case c)

3) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

In this article, C code for insertion is discussed. Following is complete C code for insertion. Following are few important points for coding the insertion process.

1) Since we need to access uncle node to decide case, it is a good idea to have parent pointer in every node.

2) We also need to write functions for left and right rotations. These functions are similar to left and right rotations for AVL Tree insertion. In AVL Tree, we maintain height, but here we need to maintain parent pointer and color.

The following code strictly follows the steps given in [CLRS book](#).

```
// C program for Red-Black Tree insertion
#include<stdio.h>
#include<stdlib.h>

//A Red-Black tree node structure
struct node
{
    int data;      // for data part
    char color;   // for color property

    //links for left, right children and parent
    struct node *left, *right, *parent;
};

// Left Rotation
void LeftRotate(struct node **root, struct node *x)
{
    //y stored pointer of right child of x
    struct node *y = x->right;

    //store y's left subtree's pointer as x's right child
    x->right = y->left;

    //update parent pointer of x's right
    if (x->right != NULL)
        x->right->parent = x;

    //update y's parent pointer
    y->parent = x->parent;

    // if x's parent is null make y as root of tree
    if (x->parent == NULL)
        (*root) = y;

    // store y at the place of x
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

    // make x as left child of y
    y->left = x;
}
```

```

//update parent pointer of x
x->parent = y;
}

// Right Rotation (Similar to LeftRotate)
void rightRotate(struct node **root, struct node *y)
{
    struct node *x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
    if (x->parent == NULL)
        (*root) = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else y->parent->right = x;
    x->right = y;
    y->parent = x;
}

// Utility function to fixup the Red-Black tree after standard BST insertion
void insertFixUp(struct node **root, struct node *z)
{
    // iterate until z is not the root and z's parent color is red
    while (z != *root && z->parent->color == 'R')
    {
        struct node *y;

        // Find uncle and store uncle in y
        if (z->parent == z->parent->parent->left)
            y = z->parent->parent->right;
        else
            y = z->parent->parent->left;

        // If uncle is RED, do following
        // (i) Change color of parent and uncle as BLACK
        // (ii) Change color of grandparent as RED
        // (iii) Move z to grandparent
        if (y->color == 'R')
        {
            y->color = 'B';
            z->parent->color = 'B';
            z->parent->parent->color = 'R';
            z = z->parent->parent;
        }

        // Uncle is BLACK, there are four cases (LL, LR, RL and RR)
        else
        {
            // Left-Left (LL) case, do following
            // (i) Swap color of parent and grandparent
            // (ii) Right Rotate Grandparent
            if (z->parent == z->parent->parent->left &&
                z == z->parent->left)
            {
                char ch = z->parent->color ;
                z->parent->color = z->parent->parent->color;
                z->parent->parent->color = ch;
                rightRotate(root, z->parent->parent);
            }

            // Left-Right (LR) case, do following
            // (i) Swap color of current node and grandparent
            // (ii) Left Rotate Parent
            // (iii) Right Rotate Grand Parent
            if (z->parent == z->parent->parent->left &&
                z == z->parent->right)
            {
                char ch = z->color ;
                z->color = z->parent->parent->color;
                z->parent->parent->color = ch;
                LeftRotate(root, z->parent);
                rightRotate(root, z->parent->parent);
            }

            // Right-Right (RR) case, do following
            // (i) Swap color of parent and grandparent
            // (ii) Left Rotate Grandparent
        }
    }
}

```

```

        if (z->parent == z->parent->parent->right &&
            z == z->parent->right)
        {
            char ch = z->parent->color ;
            z->parent->color = z->parent->parent->color;
            z->parent->parent->color = ch;
            LeftRotate(root,z->parent->parent);
        }

        // Right-Left (RL) case, do following
        // (i) Swap color of current node and grandparent
        // (ii) Right Rotate Parent
        // (iii) Left Rotate Grand Parent
        if (z->parent == z->parent->parent->right &&
            z == z->parent->left)
        {
            char ch = z->color ;
            z->color = z->parent->parent->color;
            z->parent->parent->color = ch;
            rightRotate(root,z->parent);
            LeftRotate(root,z->parent->parent);
        }
    }
}

(*root)->color = 'B'; //keep root always black
}

// Utility function to insert newly node in RedBlack tree
void insert(struct node **root, int data)
{
    // Allocate memory for new node
    struct node *z = (struct node*)malloc(sizeof(struct node));
    z->data = data;
    z->left = z->right = z->parent = NULL;

    //if root is null make z as root
    if (*root == NULL)
    {
        z->color = 'B';
        (*root) = z;
    }
    else
    {
        struct node *y = NULL;
        struct node *x = (*root);

        // Follow standard BST insert steps to first insert the node
        while (x != NULL)
        {
            y = x;
            if (z->data < x->data)
                x = x->left;
            else
                x = x->right;
        }
        z->parent = y;
        if (z->data > y->data)
            y->right = z;
        else
            y->left = z;
        z->color = 'R';

        // call insertFixUp to fix reb-black tree's property if it
        // is violated due to insertion.
        insertFixUp(root,z);
    }
}

// A utility function to traverse Red-Black tree in inorder fashion
void inorder(struct node *root)
{
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

/* Drier program to test above function*/
int main()
{

```

```
struct node *root = NULL;
insert(&root,5);
insert(&root,3);
insert(&root,7);
insert(&root,2);
insert(&root,4);
insert(&root,6);
insert(&root,8);
insert(&root,11);
printf("inorder Traversal Is : ");
inorder(root);

return 0;
}
```

Output:

Inorder Traversal Is : 2 3 4 5 6 7 8 11

[GATE Corner](#)[Quiz Corner](#)

K Dimensional Tree | Set 1 (Search and Insert)

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space.

A non-leaf node in K-D tree divides the space into two parts, called as half-spaces.

Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed.

For the sake of simplicity, let us understand a 2-D Tree with an example.

The root would have an x-aligned plane, the roots children would both have y-aligned planes, the roots grandchildren would all have x-aligned planes, and the roots great-grandchildren would all have y-aligned planes and so on.

Generalization:

Let us number the planes as 0, 1, 2, (K 1). From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as:

$$A = D \bmod K$$

How to determine if a point will lie in the left subtree or in right subtree?

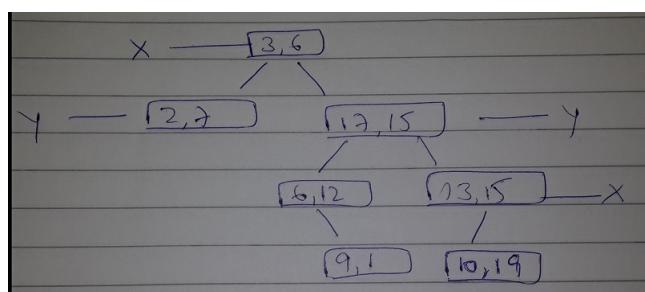
If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

Creation of a 2-D Tree:

Consider following points in a 2-D plane:

(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

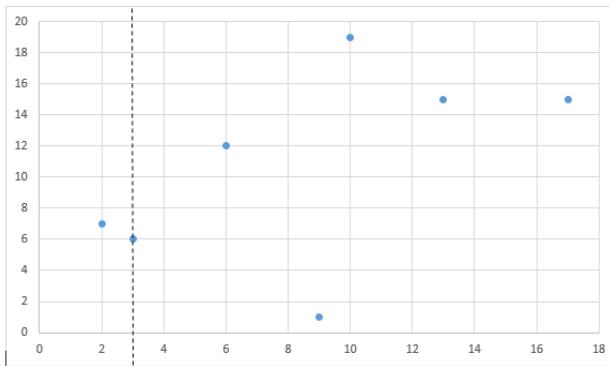
1. Insert (3, 6): Since tree is empty, make it the root node.
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the right subtree or in the left subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the left subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, 12 < 15, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. Insert (9, 1): Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7): Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the left of (13, 15).



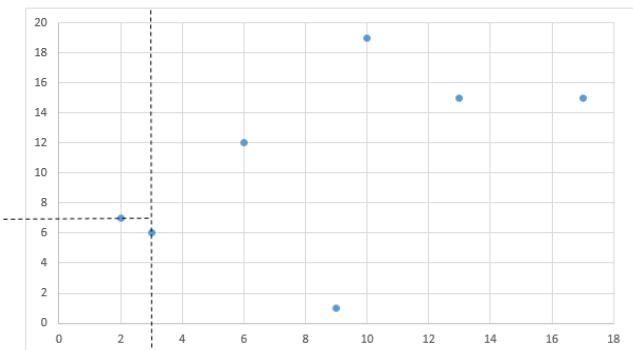
How is space partitioned?

All 7 points will be plotted in the X-Y plane as follows:

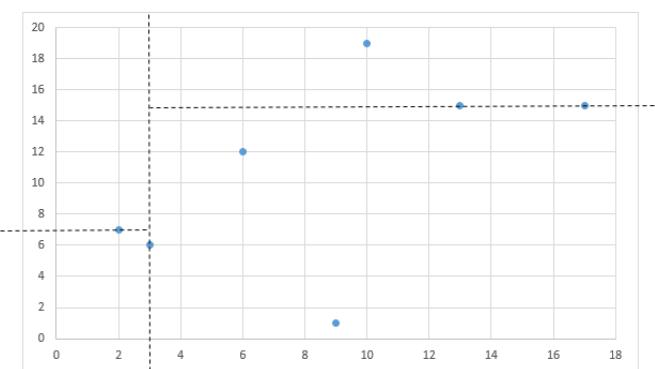
1. Point (3, 6) will divide the space into two parts: Draw line $X = 3$.



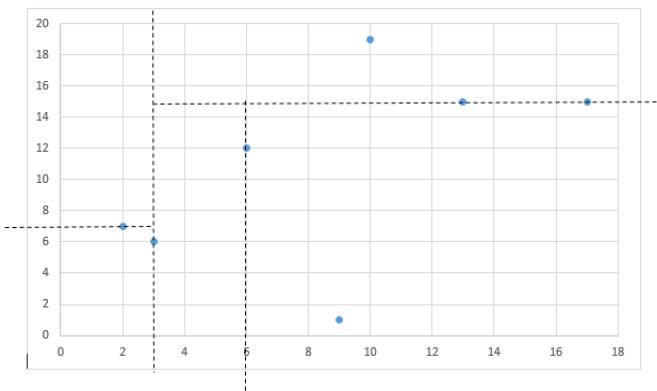
2. Point (2, 7) will divide the space to the left of line $X = 3$ into two parts horizontally.
Draw line $Y = 7$ to the left of line $X = 3$.



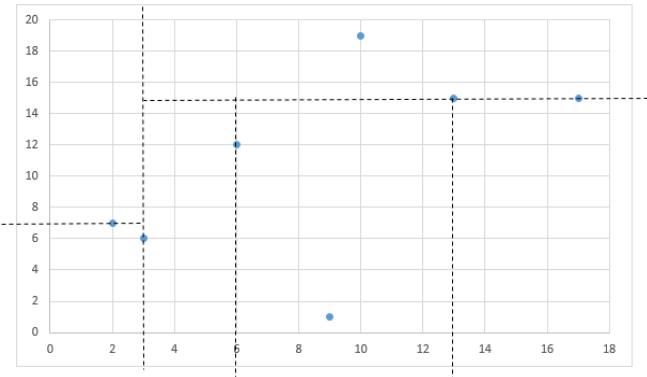
3. Point (17, 15) will divide the space to the right of line $X = 3$ into two parts horizontally.
Draw line $Y = 15$ to the right of line $X = 3$.



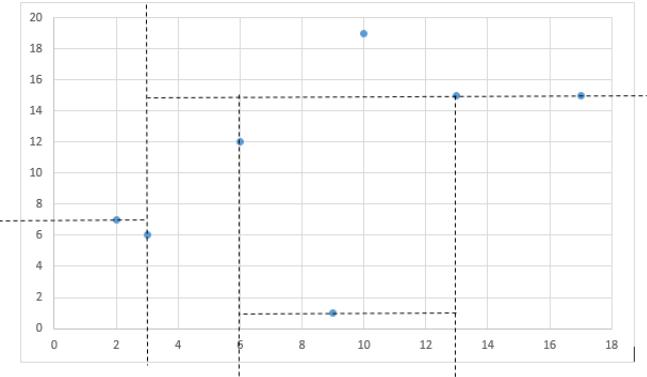
- Point (6, 12) will divide the space below line $Y = 15$ and to the right of line $X = 3$ into two parts.
Draw line $X = 6$ to the right of line $X = 3$ and below line $Y = 15$.



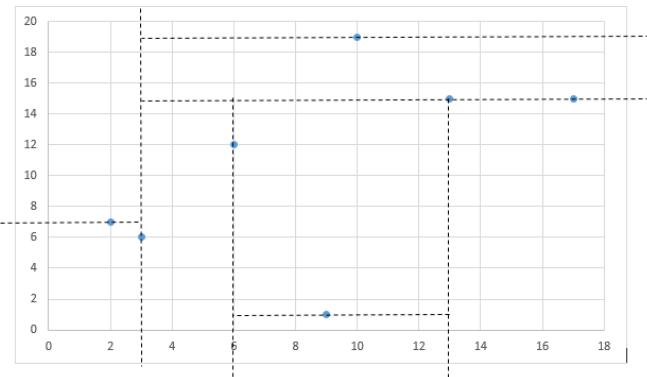
- Point (13, 15) will divide the space below line $Y = 15$ and to the right of line $X = 6$ into two parts.
Draw line $X = 13$ to the right of line $X = 6$ and below line $Y = 15$.



- Point (9, 1) will divide the space between lines $X = 3$, $X = 6$ and $Y = 15$ into two parts.
Draw line $Y = 1$ between lines $X = 3$ and $X = 6$.



- Point (10, 19) will divide the space to the right of line $X = 3$ and above line $Y = 15$ into two parts.
Draw line $Y = 19$ to the right of line $X = 3$ and above line $Y = 15$.



Following is C++ implementation of KD Tree basic operations like search, insert and delete.

```
// A C++ program to demonstrate operations of KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}
```

```

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])
            return false;

    return true;
}

// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
bool searchRec(Node* root, int point[], unsigned depth)
{
    // Base cases
    if (root == NULL)
        return false;
    if (arePointsSame(root->point, point))
        return true;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (point[cd] < root->point[cd])
        return searchRec(root->left, point, depth + 1);

    return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
    // Pass current depth as 0
    return searchRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[] = {{3, 6}, {17, 15}, {13, 15}, {6, 12},
                    {9, 1}, {2, 7}, {10, 19}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)

```

```
root = insert(root, points[i]);  
  
int point1[] = {10, 19};  
(search(root, point1)) ? cout << "Found\n": cout << "Not Found\n";  
  
int point2[] = {12, 19};  
(search(root, point2)) ? cout << "Found\n": cout << "Not Found\n";  
  
return 0;  
}
```

Output:

```
Found  
Not Found
```

Refer below articles for find minimum and delete operations.

- [K D Tree \(Find Minimum\)](#)
- [K D Tree \(Delete\)](#)

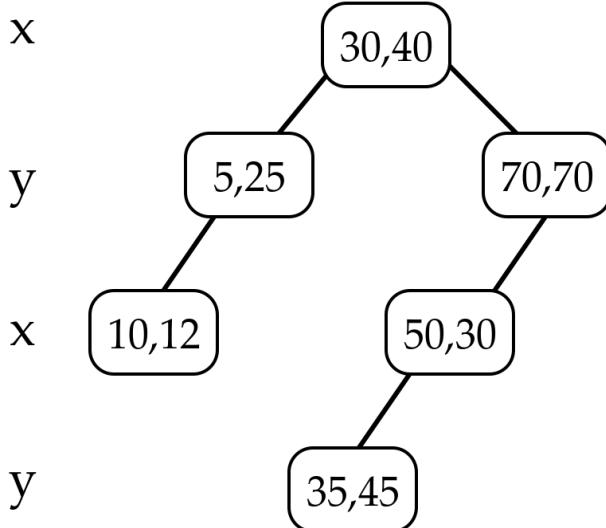
K Dimensional Tree | Set 2 (Find Minimum)

We strongly recommend to refer below post as a prerequisite of this.

[K Dimensional Tree | Set 1 \(Search and Insert\)](#)

In this post find minimum is discussed. The operation is to find minimum in the given dimension. This is especially needed in delete operation.

For example, consider below KD Tree, if given dimension is x, then output should be 5 and if given dimensions is t, then output should be 12. Below image is taken from [this](#) source.



In KD tree, points are divided dimension by dimension. For example, root divides keys by dimension 0, level next to root divides by dimension 1, next level by dimension 2 if k is more than 2 (else by dimension 0), and so on.

To find minimum we traverse nodes starting from root. **If dimension of current level is same as given dimension, then required minimum lies on left side if there is left child.** This is same as [Binary Search Tree Minimum](#).

Above is simple, what to do when current levels dimension is different. **When dimension of current level is different, minimum may be either in left subtree or right subtree or current node may also be minimum.** So we take minimum of three and return. This is different from Binary Search tree.

Below is C++ implementation of find minimum operation.

```
// A C++ program to demonstrate find minimum on KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;
```

```

// Compare the new point with root on current dimension 'cd'
// and decide the left or right subtree
if (point[cd] < (root->point[cd]))
    root->left = insertRec(root->left, point, depth + 1);
else
    root->right = insertRec(root->right, point, depth + 1);

return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
int min(int x, int y, int z)
{
    return min(x, min(y, z));
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.
int findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return INT_MAX;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root->point[d];
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return min(root->point[d],
               findMinRec(root->left, d, depth+1),
               findMinRec(root->right, d, depth+1));
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
int findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{30, 40}, {5, 25}, {70, 70},
                      {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    cout << "Minimum of 0'th dimension is " << findMin(root, 0) << endl;
    cout << "Minimum of 1'th dimension is " << findMin(root, 1) << endl;

    return 0;
}

```

Output:

Minimum of 0'th dimension is 5
 Minimum of 1'th dimension is 12

Source:

<https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf>

K Dimensional Tree | Set 3 (Delete)

We strongly recommend to refer below posts as a prerequisite of this.

[K Dimensional Tree | Set 1 \(Search and Insert\)](#)

[K Dimensional Tree | Set 2 \(Find Minimum\)](#)

In this post delete is discussed. The operation is to delete a given point from K D Tree.

Like [Binary Search Tree Delete](#), we recursively traverse down and search for the point to be deleted. Below are steps followed for every node visited.

1) If current node contains the point to be deleted

- a. If node to be deleted is a leaf node, simply delete it (Same as [BST Delete](#))
- b. If node to be deleted has right child as not NULL (Different from BST)
 - i. Find minimum of current nodes dimension in right subtree.
 - ii. Replace the node with above found minimum and recursively delete minimum in right subtree.
- c. Else If node to be deleted has left child as not NULL (Different from BST)
 - i. Find minimum of current nodes dimension in left subtree.
 - ii. Replace the node with above found minimum and recursively delete minimum in left subtree.
 - iii. Make new left subtree as right child of current node.

2) If current doesn't contain the point to be deleted

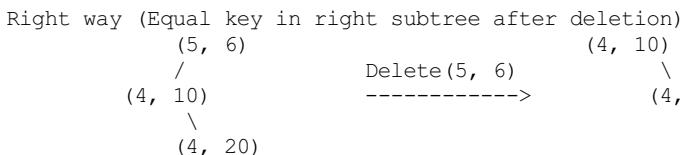
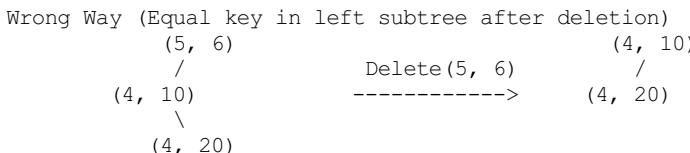
- a. If node to be deleted is smaller than current node on current dimension, recur for left subtree.
- b. Else recur for right subtree.

Why 1.b and 1.c are different from BST?

In BST delete, if a nodes left child is empty and right is not empty, we replace the node with right child. In K D Tree, doing this would violate the KD tree property as dimension of right child of node is different from nodes dimension. For example, if node divides point by x axis values, then its children divide by y axis, so we cant simply replace node with right child. Same is true for the case when right child is not empty and left child is empty.

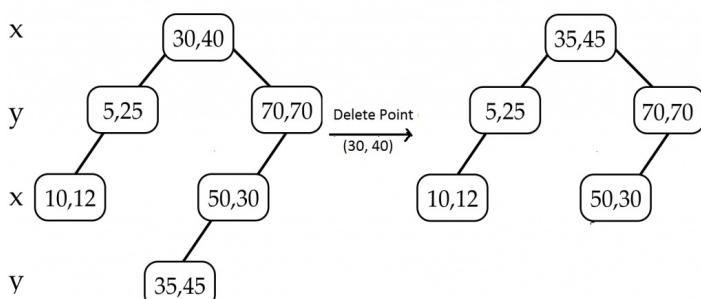
Why 1.c doesnt find max in left subtree and recur for max like 1.b?

Doing this violates the property that all equal values are in right subtree. For example, if we delete (10, 10) in below subtree and replace it with

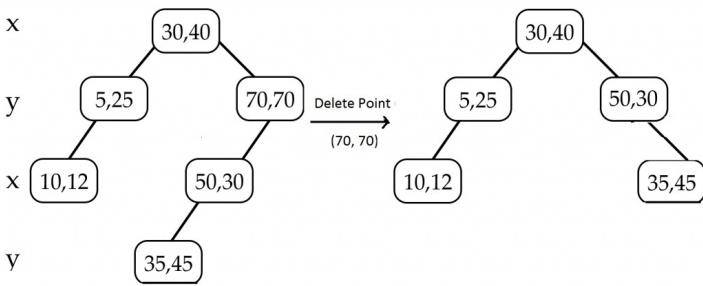


Example of Delete:

Delete (30, 40): Since right child is not NULL and dimension of node is x, we find the node with minimum x value in right child. The node is (35, 45), we replace (30, 40) with (35, 45) and delete (35, 45).



Delete (70, 70): Dimension of node is y. Since right child is NULL, we find the node with minimum y value in left child. The node is (50, 30), we replace (70, 70) with (50, 30) and recursively delete (50, 30) in left subtree. Finally we make the modified left subtree as right subtree of (50, 30).



Below is C++ implementation of K D Tree delete.

```

// A C++ program to demonstrate delete in K D tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
Node *minNode(Node *x, Node *y, Node *z, int d)
{
    Node *res = x;
    if (y != NULL && y->point[d] < res->point[d])
        res = y;
    if (z != NULL && z->point[d] < res->point[d])
        res = z;
    return res;
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.

```

```

Node *findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return NULL;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root;
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return minNode(root,
                   findMinRec(root->left, d, depth+1),
                   findMinRec(root->right, d, depth+1), d);
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
Node *findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointintate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])
            return false;

    return true;
}

// Copies point p2 to p1
void copyPoint(int p1[], int p2[])
{
    for (int i=0; i<k; i++)
        p1[i] = p2[i];
}

// Function to delete a given point 'point[]' from tree with root
// as 'root'. depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
Node *deleteNodeRec(Node *root, int point[], int depth)
{
    // Given point is not present
    if (root == NULL)
        return NULL;

    // Find dimension of current node
    int cd = depth % k;

    // If the point to be deleted is present at root
    if (arePointsSame(root->point, point))
    {
        // 2.b) If right child is not NULL
        if (root->right != NULL)
        {
            // Find minimum of root's dimension in right subtree
            Node *min = findMin(root->right, cd);

            // Copy the minimum to root
            copyPoint(root->point, min->point);

            // Recursively delete the minimum
            root->right = deleteNodeRec(root->right, min->point, depth+1);
        }
        else if (root->left != NULL) // same as above
        {
            Node *min = findMin(root->left, cd);
        }
    }
}

```

```

        copyPoint(root->point, min->point);
        root->right = deleteNodeRec(root->left, min->point, depth+1);
    }
    else // If node to be deleted is leaf node
    {
        delete root;
        return NULL;
    }
    return root;
}

// 2) If current node doesn't contain point, search downward
if (point[cd] < root->point[cd])
    root->left = deleteNodeRec(root->left, point, depth+1);
else
    root->right = deleteNodeRec(root->right, point, depth+1);
return root;
}

// Function to delete a given point from K D Tree with 'root'
Node* deleteNode(Node *root, int point[])
{
    // Pass depth as 0
    return deleteNodeRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{30, 40}, {5, 25}, {70, 70},
                       {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    // Delete (30, 40);
    root = deleteNode(root, points[0]);

    cout << "Root after deletion of (30, 40)\n";
    cout << root->point[0] << ", " << root->point[1] << endl;

    return 0;
}

```

Output:

```

Root after deletion of (30, 40)
35, 45

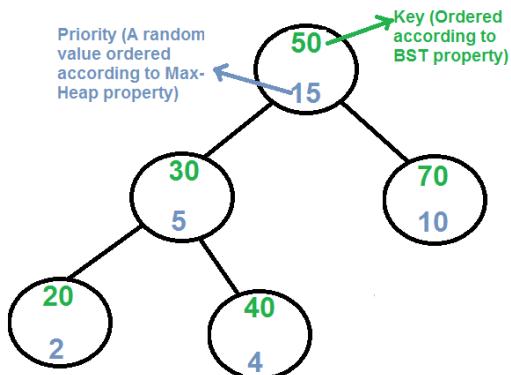
```

Source:

<https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf>

Treap (A Randomized Binary Search Tree)

Like [Red-Black](#) and [AVL](#) Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as $O(\log n)$. The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is $O(\log n)$.



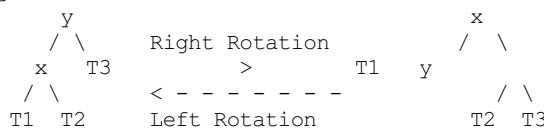
Every node of Treap maintains two values.

- 1) **Key** Follows standard BST ordering (left is smaller and right is greater)
- 2) **Priority** Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap:

Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)



Keys in both of the above trees follow the following order

$$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$$

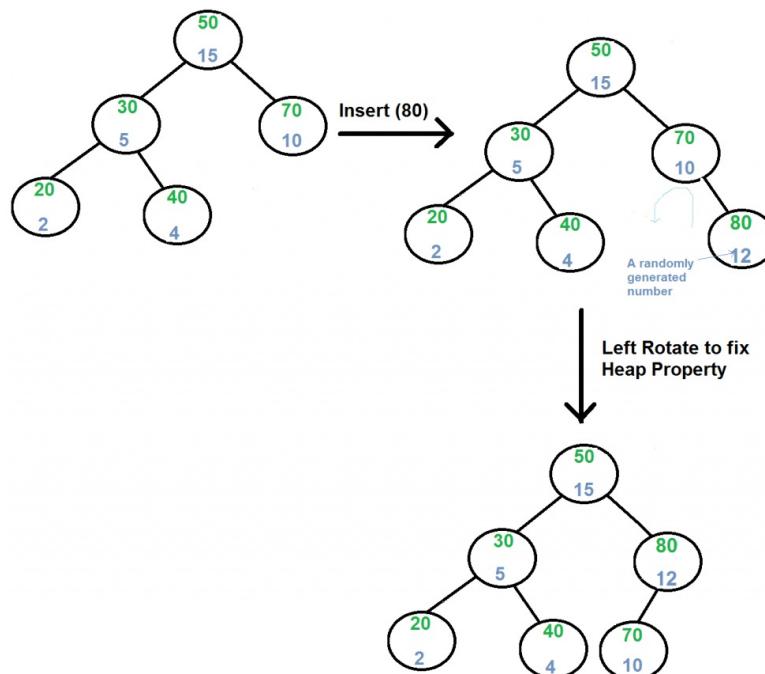
So BST property is not violated anywhere.

search(x)

Perform standard [BST Search](#) to find x.

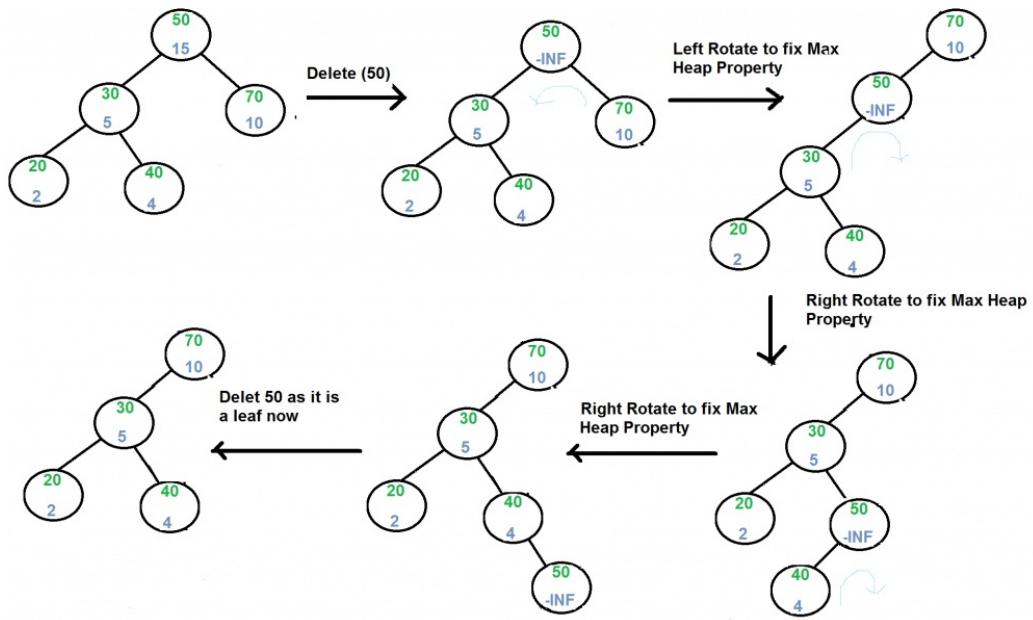
Insert(x):

- 1) Create new node with key equals to x and value equals to a random value.
- 2) Perform standard [BST insert](#).
- 3) Use rotations to make sure that inserted node's priority follows max heap property.



Delete(x):

- 1) If node to be deleted is a leaf, delete it.
- 2) Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.



Refer [Implementation of Treap Search, Insert and Delete](#) for more details.

References:

- <https://en.wikipedia.org/wiki/Treap>
- <https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture19/sld017.htm>

Ternary Search Tree

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

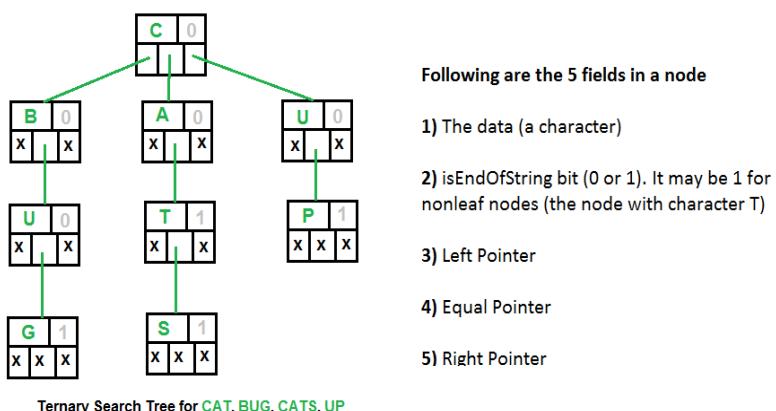
Representation of ternary search trees:

Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

Apart from above three pointers, each node has a field to indicate data(character in case of dictionary) and another field to mark end of a string. So, more or less it is similar to BST which stores data based on some order. However, data in a ternary search tree is distributed over the nodes. e.g. It needs 4 nodes to store the word Geek.

Below figure shows how exactly the words in a ternary search tree are stored?



One of the advantage of using ternary search trees over tries is that ternary search trees are a more space efficient (involve only three pointers per node as compared to 26 in standard tries). Further, ternary search trees can be used any time a hashtable would be used to store strings.

Tries are suitable when there is a proper distribution of words over the alphabets so that spaces are utilized most efficiently. Otherwise ternary search trees are better. Ternary search trees are efficient to use(in terms of space) when the strings to be stored share a common prefix.

Applications of ternary search trees:

1. Ternary search trees are efficient for queries like Given a word, find the next word in dictionary(near-neighbor lookups) or Find all telephone numbers starting with 9342 or typing few starting characters in a web browser displays all website names with this prefix(Auto complete feature).
2. Used in spell checks: Ternary search trees can be used as a dictionary to store all the words. Once the word is typed in an editor, the word can be parallelly searched in the ternary search tree to check for correct spelling.

Implementation:

Following is C implementation of ternary search tree. The operations implemented are, search, insert and traversal.

```
// C program to demonstrate Ternary Search Tree (TST) insert, traverse
// and search operations
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
struct Node
{
    char data;

    // True if this character is last character of one of the words
    unsigned isEndOfString: 1;

    struct Node *left, *eq, *right;
};

// A utility function to create a new ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp = (struct Node*) malloc(sizeof( struct Node));
    temp->data = data;
    temp->isEndOfString = 0;
    temp->left = temp->eq = temp->right = NULL;
    return temp;
}
```

```

}

// Function to insert a new word in a Ternary Search Tree
void insert(struct Node** root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller than root's character,
    // then insert this word in left subtree of root
    if ((*word) < (*root)->data)
        insert(&(*root)->left), word;

    // If current character of word is greater than root's character,
    // then insert this word in right subtree of root
    else if ((*word) > (*root)->data)
        insert(&(*root)->right), word;

    // If current character of word is same as root's character,
    else
    {
        if (*word+1)
            insert(&(*root)->eq), word+1;

        // the last character of the word
        else
            (*root)->isEndOfString = 1;
    }
}

// A recursive function to traverse Ternary Search Tree
void traverseTSTUtil(struct Node* root, char* buffer, int depth)
{
    if (root)
    {
        // First traverse the left subtree
        traverseTSTUtil(root->left, buffer, depth);

        // Store the character of this node
        buffer[depth] = root->data;
        if (root->isEndOfString)
        {
            buffer[depth+1] = '\0';
            printf( "%s\n", buffer);
        }

        // Traverse the subtree using equal pointer (middle subtree)
        traverseTSTUtil(root->eq, buffer, depth + 1);

        // Finally Traverse the right subtree
        traverseTSTUtil(root->right, buffer, depth);
    }
}

// The main function to traverse a Ternary Search Tree.
// It mainly uses traverseTSTUtil()
void traverseTST(struct Node* root)
{
    char buffer[MAX];
    traverseTSTUtil(root, buffer, 0);
}

// Function to search a given word in TST
int searchTST(struct Node *root, char *word)
{
    if (!root)
        return 0;

    if (*word < (root)->data)
        return searchTST(root->left, word);

    else if (*word > (root)->data)
        return searchTST(root->right, word);

    else
    {
        if (*word+1 == '\0')
            return root->isEndOfString;

        return searchTST(root->eq, word+1);
    }
}

```

```

    }

}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;

    insert(&root, "cat");
    insert(&root, "cats");
    insert(&root, "up");
    insert(&root, "bug");

    printf("Following is traversal of ternary search tree\n");
    traverseTST(root);

    printf("\nFollowing are search results for cats, bu and cat respectively\n");
    searchTST(root, "cats")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "bu")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "cat")? printf("Found\n"): printf("Not Found\n");

    return 0;
}

```

Output:

```

Following is traversal of ternary search tree
bug
cat
cats
up

```

```

Following are search results for cats, bu and cat respectively
Found
Not Found
Found

```

Time Complexity: The time complexity of the ternary search tree operations is similar to that of binary search tree. i.e. the insertion, deletion and search operations take time proportional to the height of the ternary search tree. The space is proportional to the length of the string to be stored.

Reference:

http://en.wikipedia.org/wiki/Ternary_search_tree

Interval Tree

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.

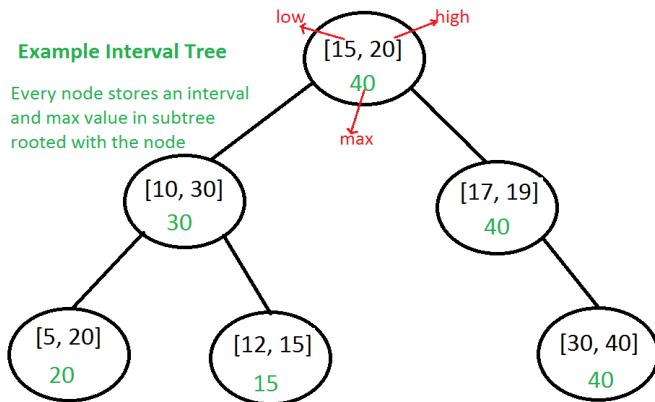
- 1) Add an interval
- 2) Remove an interval
- 3) Given an interval x , find if x overlaps with any of the existing intervals.

Interval Tree: The idea is to augment a self-balancing Binary Search Tree (BST) like [Red Black Tree](#), [AVL Tree](#), etc to maintain set of intervals so that all operations can be done in $O(\log n)$ time.

Every node of Interval Tree stores following information.

- a) **i:** An interval which is represented as a pair $[low, high]$
- b) **max:** Maximum $high$ value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.



The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with $root$.

```
Interval overlappingIntervalSearch(root, x)
1) If x overlaps with root's interval, return the root's interval.
2) If left child of root is not empty and the max in left child
   is greater than x's low value, recur for left child
3) Else recur for right child.
```

How does the above algorithm work?

Let the interval to be searched be x . We need to prove this in for following two cases.

Case 1: When we go to right subtree, one of the following must be true.

- a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than $x.low$. So the interval cannot be present in left subtree.

Case 2: When we go to left subtree, one of the following must be true.

- a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.

We went to left subtree because $x.low \leq max$ in left subtree

- . max in left subtree is a high of one of the intervals let us say $[a, max]$ in left subtree.
- . Since x doesn't overlap with any node in left subtree $x.low$ must be smaller than a .
- . All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than a .
- . From above two facts, we can say all intervals in right subtree have low value greater than $x.low$. So x cannot overlap with any interval in right subtree.

Implementation of Interval Tree:

Following is C++ implementation of Interval Tree. The implementation uses basic [insert operation of BST](#) to keep things simple. Ideally it should be [insertion of AVL Tree](#) or [insertion of Red-Black Tree](#). [Deletion from BST](#) is left as an exercise.

```
#include <iostream>
using namespace std;

// Structure to represent an interval
struct Interval
{
```

```

        int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i; // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree Node
// This is similar to BST Insert. Here the low value of interval
// is used to maintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval goes to
    // left subtree
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
bool doOverlap(Interval i1, Interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i in a given
// Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOverlap(*((root->i)), i))
        return root->i;

    // If left child of root is present and max of left child is
    // greater than or equal to given interval, then i may
    // overlap with an interval in left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

void inorder(ITNode *root)
{
    if (root == NULL) return;
}

```

```

inorder(root->left);

cout << "[" << root->i->low << ", " << root->i->high << "]"
     << " max = " << root->max << endl;

inorder(root->right);
}

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval ints[] = {{15, 20}, {10, 30}, {17, 19},
                       {5, 20}, {12, 15}, {30, 40}};
    };

    int n = sizeof(ints)/sizeof(ints[0]);
    ITNode *root = NULL;
    for (int i = 0; i < n; i++)
        root = insert(root, ints[i]);

    cout << "Inorder traversal of constructed Interval Tree is\n";
    inorder(root);

    Interval x = {6, 7};

    cout << "\nSearching for interval [" << x.low << ", " << x.high << "]";
    Interval *res = overlapSearch(root, x);
    if (res == NULL)
        cout << "\nNo Overlapping Interval";
    else
        cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";
    return 0;
}

```

Output:

```

Inorder traversal of constructed Interval Tree is
[5, 20] max = 20
[10, 30] max = 30
[12, 15] max = 15
[15, 20] max = 40
[17, 19] max = 40
[30, 40] max = 40

```

```

Searching for interval [6,7]
Overlaps with [5, 20]

```

Applications of Interval Tree:

Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source [Wiki](#)).

Interval Tree vs [Segment Tree](#)

Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Exercise:

- 1) Implement delete operation for interval tree.
- 2) Extend the intervalSearch() to print all overlapping intervals instead of just one.

http://en.wikipedia.org/wiki/Interval_tree

<http://www.cse.unr.edu/~mgunes/cs302/IntervalTrees.pptx>

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

<https://www.youtube.com/watch?v=dQF0zyaym8A>

Implement LRU Cache

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details (see the LRU page replacement slide [here](#)).

We use two data structures to implement an LRU Cache.

1. A Queue which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).

The most recently used pages will be near front end and least recently pages will be near rear end.

2. A Hash with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.

If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

Note: Initially no page is in the memory.

Below is C implementation:

```
// A C program to show implementation of LRU cache
#include <stdio.h>
#include <stdlib.h>

// A Queue Node (Queue is implemented using Doubly Linked List)
typedef struct QNode
{
    struct QNode *prev, *next;
    unsigned pageNumber; // the page number stored in this QNode
} QNode;

// A Queue (A FIFO collection of Queue Nodes)
typedef struct Queue
{
    unsigned count; // Number of filled frames
    unsigned numberOfFrames; // total number of frames
    QNode *front, *rear;
} Queue;

// A hash (Collection of pointers to Queue Nodes)
typedef struct Hash
{
    int capacity; // how many pages can be there
    QNode* *array; // an array of queue nodes
} Hash;

// A utility function to create a new Queue Node. The queue Node
// will store the given 'pageNumber'
QNode* newQNode( unsigned pageNumber )
{
    // Allocate memory and assign 'pageNumber'
    QNode* temp = (QNode *)malloc( sizeof( QNode ) );
    temp->pageNumber = pageNumber;

    // Initialize prev and next as NULL
    temp->prev = temp->next = NULL;

    return temp;
}

// A utility function to create an empty Queue.
// The queue can have at most 'numberOfFrames' nodes
Queue* createQueue( int numberOfFrames )
{
    Queue* queue = (Queue *)malloc( sizeof( Queue ) );

    // The queue is empty
    queue->count = 0;
    queue->front = queue->rear = NULL;
```

```

// Number of frames that can be stored in memory
queue->numberOfFrames = numberOfFrames;

return queue;
}

// A utility function to create an empty Hash of given capacity
Hash* createHash( int capacity )
{
    // Allocate memory for hash
    Hash* hash = (Hash *) malloc( sizeof( Hash ) );
    hash->capacity = capacity;

    // Create an array of pointers for referring queue nodes
    hash->array = (QNode **) malloc( hash->capacity * sizeof( QNode* ) );

    // Initialize all hash entries as empty
    int i;
    for( i = 0; i < hash->capacity; ++i )
        hash->array[i] = NULL;

    return hash;
}

// A function to check if there is slot available in memory
int AreAllFramesFull( Queue* queue )
{
    return queue->count == queue->numberOfFrames;
}

// A utility function to check if queue is empty
int isQueueEmpty( Queue* queue )
{
    return queue->rear == NULL;
}

// A utility function to delete a frame from queue
void deQueue( Queue* queue )
{
    if( isQueueEmpty( queue ) )
        return;

    // If this is the only node in list, then change front
    if (queue->front == queue->rear)
        queue->front = NULL;

    // Change rear and remove the previous rear
    QNode* temp = queue->rear;
    queue->rear = queue->rear->prev;

    if (queue->rear)
        queue->rear->next = NULL;

    free( temp );

    // decrement the number of full frames by 1
    queue->count--;
}

// A function to add a page with given 'pageNumber' to both queue
// and hash
void Enqueue( Queue* queue, Hash* hash, unsigned pageNumber )
{
    // If all frames are full, remove the page at the rear
    if ( AreAllFramesFull ( queue ) )
    {
        // remove page from hash
        hash->array[ queue->rear->pageNumber ] = NULL;
        deQueue( queue );
    }

    // Create a new node with given page number,
    // And add the new node to the front of queue
    QNode* temp = newQNode( pageNumber );
    temp->next = queue->front;

    // If queue is empty, change both front and rear pointers
    if ( isQueueEmpty( queue ) )
        queue->rear = queue->front = temp;
    else // Else change the front
    {

```

```

queue->front->prev = temp;
queue->front = temp;
}

// Add page entry to hash also
hash->array[ pageNumber ] = temp;

// increment number of full frames
queue->count++;

}

// This function is called when a page with given 'pageNumber' is referenced
// from cache (or memory). There are two cases:
// 1. Frame is not there in memory, we bring it in memory and add to the front
//    of queue
// 2. Frame is there in memory, we move the frame to front of queue
void ReferencePage( Queue* queue, Hash* hash, unsigned pageNumber )
{
    QNode* reqPage = hash->array[ pageNumber ];

    // the page is not in cache, bring it
    if ( reqPage == NULL )
        Enqueue( queue, hash, pageNumber );

    // page is there and not at front, change pointer
    else if (reqPage != queue->front)
    {
        // Unlink requested page from its current location
        // in queue.
        reqPage->prev->next = reqPage->next;
        if (reqPage->next)
            reqPage->next->prev = reqPage->prev;

        // If the requested page is rear, then change rear
        // as this node will be moved to front
        if (reqPage == queue->rear)
        {
            queue->rear = reqPage->prev;
            queue->rear->next = NULL;
        }

        // Put the requested page before current front
        reqPage->next = queue->front;
        reqPage->prev = NULL;

        // Change prev of current front
        reqPage->next->prev = reqPage;

        // Change front to the requested page
        queue->front = reqPage;
    }
}

// Driver program to test above functions
int main()
{
    // Let cache can hold 4 pages
    Queue* q = createQueue( 4 );

    // Let 10 different pages can be requested (pages to be
    // referenced are numbered from 0 to 9
    Hash* hash = createHash( 10 );

    // Let us refer pages 1, 2, 3, 1, 4, 5
    ReferencePage( q, hash, 1 );
    ReferencePage( q, hash, 2 );
    ReferencePage( q, hash, 3 );
    ReferencePage( q, hash, 1 );
    ReferencePage( q, hash, 4 );
    ReferencePage( q, hash, 5 );

    // Let us print cache frames after the above referenced pages
    printf ("%d ", q->front->pageNumber);
    printf ("%d ", q->front->next->pageNumber);
    printf ("%d ", q->front->next->next->pageNumber);
    printf ("%d ", q->front->next->next->next->pageNumber);

    return 0;
}

```

Output:

5 4 1 3

Sort numbers stored on different machines

Given N machines. Each machine contains some numbers in sorted form. But the amount of numbers, each machine has is not fixed. Output the numbers from all the machine in sorted non-decreasing form.

Example:

```
Machine M1 contains 3 numbers: {30, 40, 50}
Machine M2 contains 2 numbers: {35, 45}
Machine M3 contains 5 numbers: {10, 60, 70, 80, 100}

Output: {10, 30, 35, 40, 45, 50, 60, 70, 80, 100}
```

Representation of stream of numbers on each machine is considered as linked list. A Min Heap can be used to print all numbers in sorted order.

Following is the detailed process

1. Store the head pointers of the linked lists in a minHeap of size N where N is number of machines.
2. Extract the minimum item from the minHeap. Update the minHeap by replacing the head of the minHeap with the next number from the linked list or by replacing the head of the minHeap with the last number in the minHeap followed by decreasing the size of heap by 1.
3. Repeat the above step 2 until heap is not empty.

Below is C++ implementation of the above approach.

```
// A program to take numbers from different machines and print them in sorted order
#include <stdio.h>

// A Linked List node
struct ListNode
{
    int data;
    struct ListNode* next;
};

// A Min Heap Node
struct MinHeapNode
{
    ListNode* head;
};

// A Min Heao (Collection of Min Heap nodes)
struct MinHeap
{
    int count;
    int capacity;
    MinHeapNode* array;
};

// A function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;
    minHeap->capacity = capacity;
    minHeap->count = 0;
    minHeap->array = new MinHeapNode [minHeap->capacity];
    return minHeap;
}

/* A utility function to insert a new node at the begining
   of linked list */
void push (ListNode** head_ref, int new_data)
{
    /* allocate node */
    ListNode* new_node = new ListNode;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
```

```

void swap( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;
    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;

    if ( left < minHeap->count &&
        minHeap->array[left].head->data <
        minHeap->array[smallest].head->data
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[right].head->data <
        minHeap->array[smallest].head->data
    )
        smallest = right;

    if( smallest != idx )
    {
        swap( &minHeap->array[smallest], &minHeap->array[idx] );
        minHeapify( minHeap, smallest );
    }
}

// A utility function to check whether a Min Heap is empty or not
int isEmpty( MinHeap* minHeap )
{
    return (minHeap->count == 0);
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int i, n;
    n = minHeap->count - 1;
    for( i = (n - 1) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// This function inserts array elements to heap and then calls
// buildHeap for heap property among nodes
void populateMinHeap( MinHeap* minHeap, ListNode* *array, int n )
{
    for( int i = 0; i < n; ++i )
        minHeap->array[ minHeap->count++ ].head = array[i];

    buildMinHeap( minHeap );
}

// Return minimum element from all linked lists
ListNode* extractMin( MinHeap* minHeap )
{
    if( isEmpty( minHeap ) )
        return NULL;

    // The root of heap will have minimum value
    MinHeapNode temp = minHeap->array[0];

    // Replace root either with next node of the same list.
    if( temp.head->next )
        minHeap->array[0].head = temp.head->next;
    else // If list empty, then reduce heap size
    {
        minHeap->array[0] = minHeap->array[ minHeap->count - 1 ];
        --minHeap->count;
    }

    minHeapify( minHeap, 0 );
    return temp.head;
}

```

```

// The main function that takes an array of lists from N machines
// and generates the sorted output
void externalSort( ListNode *array[], int N )
{
    // Create a min heap of size equal to number of machines
    MinHeap* minHeap = createMinHeap( N );

    // populate first item from all machines
    populateMinHeap( minHeap, array, N );

    while ( !isEmpty( minHeap ) )
    {
        ListNode* temp = extractMin( minHeap );
        printf( "%d ", temp->data );
    }
}

// Driver program to test above functions
int main()
{
    int N = 3; // Number of machines

    // an array of pointers storing the head nodes of the linked lists
    ListNode *array[N];

    // Create a Linked List 30->40->50 for first machine
    array[0] = NULL;
    push (&array[0], 50);
    push (&array[0], 40);
    push (&array[0], 30);

    // Create a Linked List 35->45 for second machine
    array[1] = NULL;
    push (&array[1], 45);
    push (&array[1], 35);

    // Create Linked List 10->60->70->80 for third machine
    array[2] = NULL;
    push (&array[2], 100);
    push (&array[2], 80);
    push (&array[2], 70);
    push (&array[2], 60);
    push (&array[2], 10);

    // Sort all elements
    externalSort( array, N );

    return 0;
}

```

Output:

10 30 35 40 45 50 60 70 80 100

Find the k most frequent words from a file

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in [this](#) post).

Trie and Min Heap are linked with each other by storing an additional field in Trie indexMinHeap and a pointer trNode in Min Heap. The value of indexMinHeap is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, indexMinHeap contains, index of the word in Min Heap. The pointer trNode in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by indexMinHeap field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().

3. The min heap is full. Two sub-cases arise.

.3.1 The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.

.3.2 The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the word to be replaced in Trie with -1 as the word is no longer in min heap.

4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

```
// A program to find k most frequent words in a file
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_CHARS 26
#define MAX_WORD_SIZE 30

// A Trie node
struct TrieNode
{
    bool isEnd; // indicates end of word
    unsigned frequency; // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to 'z'.
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; // number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; // represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;
```

```

// Initialize values for new node
trieNode->isEnd = 0;
trieNode->frequency = 0;
trieNode->indexMinHeap = -1;
for( int i = 0; i < MAX_CHARS; ++i )
    trieNode->child[i] = NULL;

return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHeapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;

    if ( left < minHeap->count &&
        minHeap->array[ left ].frequency <
        minHeap->array[ smallest ].frequency
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[ right ].frequency <
        minHeap->array[ smallest ].frequency
    )
        smallest = right;

    if( smallest != idx )
    {
        // Update the corresponding index in Trie node.
        minHeap->array[ smallest ].root->indexMinHeap = idx;
        minHeap->array[ idx ].root->indexMinHeap = smallest;

        // Swap nodes in min heap
        swapMinHeapNodes ( &minHeap->array[ smallest ], &minHeap->array[ idx ] );

        minHeapify( minHeap, smallest );
    }
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained above
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )

```

```

{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ count ]. word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;

        ++( minHeap->count );
        buildMinHeap( minHeap );
    }

    // Case 3: Word is not present and heap is full. And frequency of word
    // is more than root. The root is the least frequent word in heap,
    // replace root with new word
    else if ( (*root)->frequency > minHeap->array[0]. frequency )
    {

        minHeap->array[ 0 ]. root->indexMinHeap = -1;
        minHeap->array[ 0 ]. root = *root;
        minHeap->array[ 0 ]. root->indexMinHeap = 0;
        minHeap->array[ 0 ]. frequency = (*root)->frequency;

        // delete previously allocated memory and
        delete [] minHeap->array[ 0 ]. word;
        minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ 0 ]. word, word );

        minHeapify ( minHeap, 0 );
    }
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                  const char* word, const char* dupWord )
{
    // Base Case
    if ( *root == NULL )
        *root = newTrieNode();

    // There are still more characters in word
    if ( *word != '\0' )
        insertUtil ( &((*root)->child[ tolower( *word ) - 97 ]),
                     minHeap, word + 1, dupWord );
    else // The complete word is processed
    {
        // word is already present, increase the frequency
        if ( (*root)->isEnd )
            ++( (*root)->frequency );
        else
        {
            (*root)->isEnd = 1;
            (*root)->frequency = 1;
        }

        // Insert in min heap also
        insertInMinHeap( minHeap, root, dupWord );
    }
}

// add a word to Trie & min heap. A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minHeap)
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap

```

```

// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                minHeap->array[i].frequency );
    }
}

// The main function that takes a file as input, add words to heap
// and Trie, finally shows result from heap
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];

    // Read words one by one from file. Insert the word in Trie and Min Heap
    while( fscanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);

    // The Min Heap will have the k most frequent words, so print Min Heap nodes
    displayMinHeap( minHeap );
}

// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ("file.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}

```

Output:

```

your : 3
well : 3
and : 4
to : 4
Geeks : 6

```

The above output is for a file with following content.

Welcome to the world of Geeks
This portal has been created to provide well written well thought and well explained
solutions for selected questions If you like Geeks for Geeks and would like to contribute
here is your chance You can write article and mail your article to contribute at
geeksforgeeks.org See your article appearing on the Geeks for Geeks main page and help
thousands of other Geeks

Given a sequence of words, print all anagrams together | Set 2

Given an array of words, print all anagrams together. For example, if the given array is {cat, dog, tac, god, act}, then output may be cat tac act dog god.

We have discussed two different methods in the [previous post](#). In this post, a more efficient solution is discussed.

Trie data structure can be used for a more efficient solution. Insert the sorted order of each word in the trie. Since all the anagrams will end at the same leaf node. We can start a linked list at the leaf nodes where each node represents the index of the original array of words. Finally, traverse the Trie. While traversing the Trie, traverse each linked list one line at a time. Following are the detailed steps.

- 1) Create an empty Trie
- 2) One by one take all words of input sequence. Do following for each word
 - a) Copy the word to a buffer.
 - b) Sort the buffer
 - c) Insert the sorted buffer and index of this word to Trie. Each leaf node of Trie is head of a Index list. The Index list stores index of words in original sequence. If sorted buffer is already present, we insert index of this word to the index list.
- 3) Traverse Trie. While traversing, if you reach a leaf node, traverse the index list. And print all words using the index obtained from Index list.

```
// An efficient program to print all anagrams together
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cctype.h>

#define NO_OF_CHARS 26

// Structure to represent list node for indexes of words in
// the given sequence. The list nodes are used to connect
// anagrams at leaf nodes of Trie
struct IndexNode
{
    int index;
    struct IndexNode* next;
};

// Structure to represent a Trie Node
struct TrieNode
{
    bool isEnd; // indicates end of word
    struct TrieNode* child[NO_OF_CHARS]; // 26 slots each for 'a' to 'z'
    struct IndexNode* head; // head of the index list
};

// A utility function to create a new Trie node
struct TrieNode* newTrieNode()
{
    struct TrieNode* temp = new TrieNode;
    temp->isEnd = 0;
    temp->head = NULL;
    for (int i = 0; i < NO_OF_CHARS; ++i)
        temp->child[i] = NULL;
    return temp;
}

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare(const void* a, const void* b)
{ return *(char*)a - *(char*)b; }

/* A utility function to create a new linked list node */
struct IndexNode* newIndexNode(int index)
{
    struct IndexNode* temp = new IndexNode;
    temp->index = index;
    temp->next = NULL;
    return temp;
}

// A utility function to insert a word to Trie
void insert(struct TrieNode** root, char* word, int index)
{
    // Base case
    if (*root == NULL)
        *root = newTrieNode();
    else
    {
```

```

if (*word != '\0')
    insert( &(*root)->child[tolower(*word) - 'a'], word+1, index );
else // If end of the word reached
{
    // Insert index of this word to end of index linked list
    if ((*root)->isEnd)
    {
        IndexNode* pCrawl = (*root)->head;
        while( pCrawl->next )
            pCrawl = pCrawl->next;
        pCrawl->next = newIndexNode(index);
    }
    else // If Index list is empty
    {
        (*root)->isEnd = 1;
        (*root)->head = newIndexNode(index);
    }
}
}

// This function traverses the built trie. When a leaf node is reached,
// all words connected at that leaf node are anagrams. So it traverses
// the list at leaf node and uses stored index to print original words
void printAnagramsUtil(struct TrieNode* root, char *wordArr[])
{
    if (root == NULL)
        return;

    // If a lead node is reached, print all anagrams using the indexes
    // stored in index linked list
    if (root->isEnd)
    {
        // traverse the list
        IndexNode* pCrawl = root->head;
        while (pCrawl != NULL)
        {
            printf( "%s \n", wordArr[ pCrawl->index ] );
            pCrawl = pCrawl->next;
        }
    }

    for (int i = 0; i < NO_OF_CHARS; ++i)
        printAnagramsUtil(root->child[i], wordArr);
}

// The main function that prints all anagrams together. wordArr[] is input
// sequence of words.
void printAnagramsTogether(char* wordArr[], int size)
{
    // Create an empty Trie
    struct TrieNode* root = NULL;

    // Iterate through all input words
    for (int i = 0; i < size; ++i)
    {
        // Create a buffer for this word and copy the word to buffer
        int len = strlen(wordArr[i]);
        char *buffer = new char[len+1];
        strcpy(buffer, wordArr[i]);

        // Sort the buffer
        qsort( (void*)buffer, strlen(buffer), sizeof(char), compare );

        // Insert the sorted buffer and its original index to Trie
        insert(&root, buffer, i);
    }

    // Traverse the built Trie and print all anagrams together
    printAnagramsUtil(root, wordArr);
}

// Driver program to test above functions
int main()
{
    char* wordArr[] = {"cat", "dog", "tac", "god", "act", "gdo"};
    int size = sizeof(wordArr) / sizeof(wordArr[0]);
    printAnagramsTogether(wordArr, size);
    return 0;
}

```

Output:

```
cat  
tac  
act  
dog  
god  
gdo
```

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

[Tournament tree](#) is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

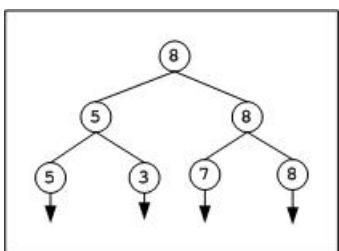
There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](#) (put $n = 2$ in equation given in the post).

It is obvious that to select the best player among N players, $(N - 1)$ players to be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons. For details read [this comment](#).

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

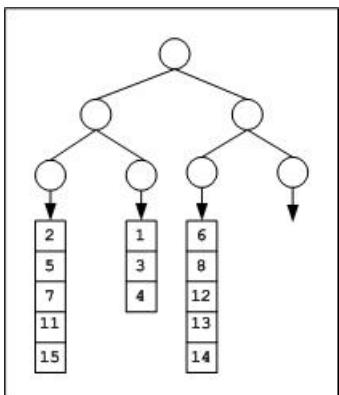
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2 M$)** to have atleast M external nodes.

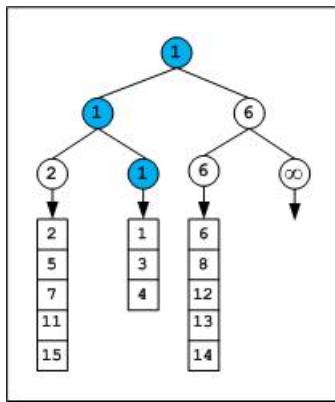
Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1  
{ 1, 3, 4 } ---- Array2  
{ 6, 8, 12, 13, 14 } ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 = 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



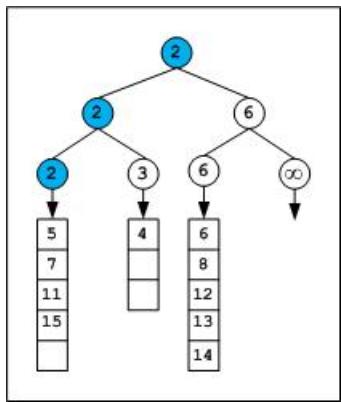
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being held in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2, \dots, L_m requires time complexity of $O(L_1 + L_2 + \dots + L_m * \log M)$ to merge all the arrays, and $O(m * \log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

Related Posts

[Link 1](#), [Link 2](#), [Link 3](#), [Link 4](#), [Link 5](#), [Link 6](#), [Link 7](#).

Decision Trees Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)

Let us solve the classic fake coin puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume $N = 8$.

Easy: Given a two pan fair balance and N identically looking coins, out of which only one coin is **lighter** (or **heavier**). To figure out the odd coin, how many minimum number of weighing are required in the worst case?

Difficult: Given a two pan fair balance and N identically looking coins out of which only one coin **may be** defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

Given 5 coins out of which one coin is **lighter**. In the worst case, how many minimum number of weighing are required to figure out the odd coin?

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best outcome of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin is heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k -th level will have 3^k leaves and hence we need $3^k \geq N$.

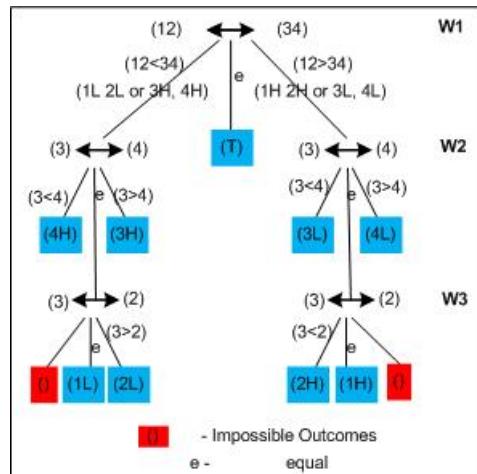
In other words, in k trials we can examine upto 3^k coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

We are given 4 coins, out of which only one coin **may be** defective. We dont know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.

From the above analysis we may think that $k = 2$ trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than $N = 4$ (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be $(12) < (34)$ i.e. we go on to left subtree or $(12) > (34)$ i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh (1, 2) or (3, 4). Let us consider (3, 4) as the analogy for (1, 2) is similar. The outcome of second trial can be three ways

- A) (3) < (4) yielding 4 as defective heavier coin, OR
- B) (3) > (4) yielding 3 as defective heavier coin OR
- C) (3) = (4), yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took (3, 2) where 3 is confirmed as genuine. We can get (3) > (2) in which 2 is lighter, or (3) = (2) in which 1 is lighter. Note that it impossible to get (3) < (2), it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast $(2N + 1)$ leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total $(2N + 1)$ leaves.

As explained earlier ternary tree at level k, can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > \log_3(2N + 1)$ weighing to find the defective one.

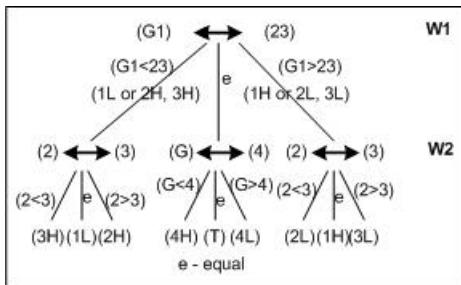
Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is going to yield valid output (in simple terms generate full 3-ary tree). Problem4 describes this approach of 12 coins.

Problem3: (Special case of two pan balance)

We are given 5 coins, a group of 4 coins out of which one coin is defective (we **don't know** whether it is heavier or lighter), and one coin is genuine. How many weighings are required in worst case to figure out the odd coin whether it is heavier or lighter?

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as [(G1, 23) and (4)]. Any other group cant generate full 3-ary tree, try yourself. The following diagram explains the procedure.



The middle case (G1) = (23) is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case (G1) < (23). This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance could be (2) < (3) yielding 3 as heavier or (2) > (3) yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case (G1) > (23). This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case could be (2) < (3) yielding 2 as lighter coin, or (2) > (3) yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with $(N + 1) = 5$ coins where $N = 4$, we end up with $(2N + 1) = 9$ leaves. **Infact we should have 11 outcomes since we stared with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure these two outcomes?**

If we observe the figure, after the first weighing the problem reduced to we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter). This can be solved in one weighing (read Problem1).

Analysis: Given $(N + 1)$ coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of $(2N + 1)$ leaves. Since the total possible outcomes are $(2(N + 1) + 1)$, number of weighing(trials) are given by the height of ternary tree, $k \geq \log_3[2(N + 1) + 1]$. Note the equality sign.

Rearranging k and N , we can weigh maximum of $N \leq (3^k - 1)/2$ coins in k trials.

Problem4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can't lead to best tree. From the above two examples, we can ensure that the decision tree can be used in optimal way if we can reveal at least one genuine coin. Remember to group coins such that the first weighing reveals at least one genuine coin.

Let us name the coins as 1, 2, 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. $(1234) = (5678)$, both groups are equal. Defective coin may be in (ABCD) group.
2. $(1234) < (5678)$, i.e. first group is less in weight than second group.
3. $(1234) > (5678)$, i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where $(1234) < (5678)$. It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing, [can you try?](#)). If we weigh these two groups again the outcome can be three ways, i) $(1235) < (4BCD)$ yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) $(1235) = (4BCD)$ yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) $(1235) > (4BCD)$ yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where $(1234) > (5678)$) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing in the worst case.

Few Interesting Puzzles:

1. Solve Problem4 with $N = 8$ and $N = 13$, How many minimum trials are required in each case?
2. Given a function $\text{int weigh}(\text{A}[], \text{B}[])$ where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if $A < B$ and 1 if $A > B$. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of others, smaller or greater than others. Write a program to find the odd element (if any) using `weigh()` minimum number of times.
3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are needed to figure out odd coin?

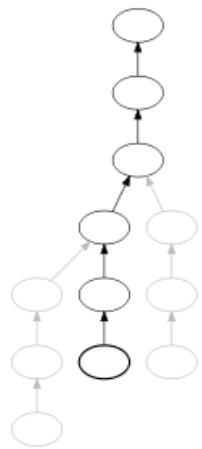
References:

Similar problem was provided in one of the exercises of the book *Introduction to Algorithms* by Levitin. Specifically read section 5.5 and section 11.2 including exercises.

Spaghetti Stack

Spaghetti stack

A spaghetti stack is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa)



Spaghetti stack structure is used in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use. Following are some applications of Spaghetti Stack.

Compilers for languages such as C create a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level parent symbol table and so on.

Spaghetti Stacks are also used to implement [Disjoint-set data structure](#).

Sources:

http://en.wikipedia.org/wiki/Spaghetti_stack

Data Structure for Dictionary and Spell Checker?

Which data structure can be used for efficiently building a word dictionary and [Spell Checker](#)?

The answer depends upon the functionalists required in Spell Checker and availability of memory. For example following are few possibilities.

[Hashing](#) is one simple option for this. We can put all words in a hash table. Refer [this](#) paper which compares hashing with self-balancing Binary Search Trees and Skip List, and shows that hashing performs better.

Hashing doesn't support operations like prefix search. Prefix search is something where a user types a prefix and your dictionary shows all words starting with that prefix. Hashing also doesn't support efficient printing of all words in dictionary in alphabetical order and nearest neighbor search.

If we want both operations, look up and prefix search, [Trie](#) is suited. With Trie, we can support all operations like insert, search, delete in $O(n)$ time where n is length of the word to be processed. Another advantage of Trie is, we can print all words in alphabetical order which is not possible with hashing.

The disadvantage of Trie is, it requires lots of space. If space is concern, then [Ternary Search Tree](#) can be preferred. In Ternary Search Tree, time complexity of search operation is $O(h)$ where h is height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

If we want to support suggestions, like google shows *did you mean*, then we need to find the closest word in dictionary. The closest word can be defined as the word that can be obtained with minimum number of character transformations (add, delete, replace). A Naive way is to take the given word and generate all words which are 1 distance (1 edit or 1 delete or 1 replace) away and one by one look them in dictionary. If nothing found, then look for all words which are 2 distant and so on. There are many complex algorithms for this. As per [the wiki page](#), The most successful algorithm to date is Andrew Golding and Dan Roth's Window-based spelling correction algorithm.

See [this](#) for a simple spell checker implementation.

Binary Indexed Tree or Fenwick tree

Let us consider the following problem to understand Binary Indexed Tree.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

1 Find the sum of first i elements.

2 Change value of a specified element of the array $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from 0 to $i-1$ and calculate sum of elements. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time. Another simple solution is to create another array and store sum from start to i at the i th index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

Can we perform both the operations in $O(\log n)$ time once given the array?

One Efficient Solution is to use [Segment Tree](#) that does both operations in $O(\log n)$ time.

Using *Binary Indexed Tree*, we can do both tasks in $O(\log n)$ time. The advantages of *Binary Indexed Tree* over Segment are, requires less space and very easy to implement..

Representation

Binary Indexed Tree is represented as an array. Let the array be $\text{BITree}[]$. Each node of Binary Indexed Tree stores sum of some elements of given array. Size of Binary Indexed Tree is equal to n where n is size of input array. In the below code, we have used size as $n+1$ for ease of implementation.

Construction

We construct the Binary Indexed Tree by first initializing all values in $\text{BITree}[]$ as 0. Then we call `update()` operation for all indexes to store actual sums, update is discussed below.

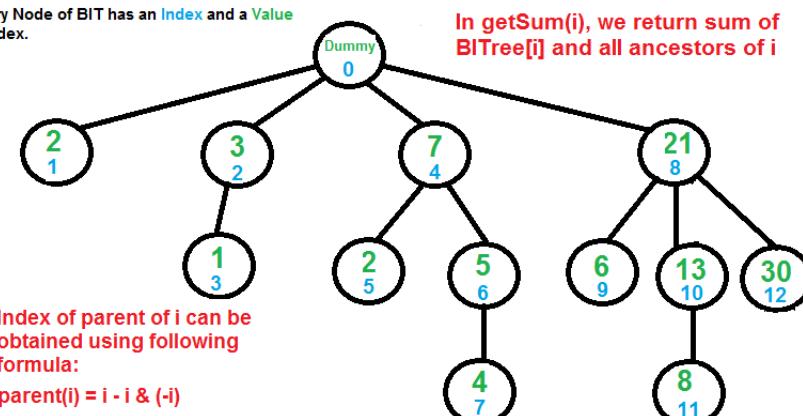
Operations

getSum(index) : Returns sum of $\text{arr}[0 \dots \text{index}]$

```
// Returns sum of arr[0..index] using BITree[0..n]. It assumes that
// BITree[] is constructed for given array arr[0..n-1]
1) Initialize sum as 0 and index as index+1.
2) Do following while index is greater than 0.
...a) Add BITree[index] to sum
...b) Go to parent of BITree[index]. Parent can be obtained by removing
   the last set bit from index, i.e., index = index - (index & (-index))
3) Return sum.
```

Every Node of BIT has an Index and a Value
at index.

In `getSum(i)`, we return sum of
BITree[i] and all ancestors of i



The above formula basically removes the last set bit from i. For example, if $i = 12$, then $\text{parent}(i)$ is 8

| | |
|----------------|---|
| Input Array: | $\text{arr}[0..n-1] = \{2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ |
| BI Tree Array: | $\text{BITree}[1..n] = \{2, 3, 1, 7, 2, 5, 4, 21, 6, 13, 8, 30\}$ |

View of Binary Indexed Tree to understand getSum() operation

The above diagram demonstrates working of `getSum()`. Following are some important observations.

Node at index 0 is a dummy node.

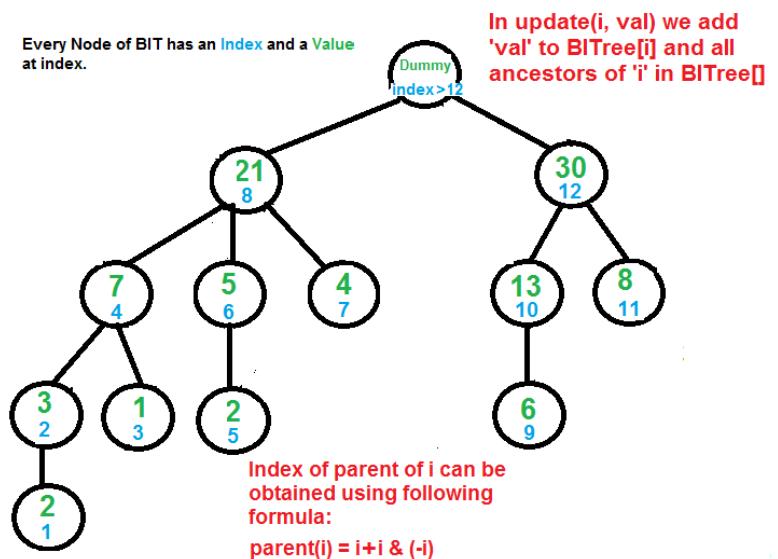
A node at index y is parent of a node at index x , iff y can be obtained by removing last set bit from binary representation of x .

A child x of a node y stores sum of elements from y (exclusive y) and of x (inclusive x).

```

update(index, val): Updates BIT for operation arr[index] += val
// Note that arr[] is not changed here. It changes
// only BI Tree for the already made change in arr[].
1) Initialize index as index+1.
2) Do following while index is smaller than or equal to n.
...a) Add value to BITree[index]
...b) Go to parent of BITree[index]. Parent can be obtained by removing
    the last set bit from index, i.e., index = index - (index & (-index))

```



Contents of arr[] and BITree[] are same as above diagram for getSum()

View of Binary Indexed Tree to understand update() operation

The update process needs to make sure that all BITree nodes that have arr[i] as part of the section they cover must be updated. We get all such nodes of BITree by repeatedly adding the decimal number corresponding to the last set bit.

How does Binary Indexed Tree work?

The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as $16 + 2 + 1$. Every node of BI Tree stores sum of *n* elements where *n* is a power of 2. For example, in the above first diagram for getSum(), sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number *n* is $O(\log n)$. Therefore, we traverse at-most $O(\log n)$ nodes in both getSum() and update() operations. Time complexity of construction is $O(n \log n)$ as it calls update() for all *n* elements.

Implementation:

Following is C++ implementation of Binary Indexed Tree.

```

// C++ code to demonstrate operations of Binary Index Tree
#include <iostream>
using namespace std;

/*
    n      --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary Indexed Tree.
    arr[0..n-1]  --> Input array for which prefix sum is evaluated. */

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int n, int index)
{
    int sum = 0; // Initialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node
        index -= index & (-index);
    }
}

```

```

    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree. The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int *BITree, int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of BI Tree
        BITree[index] += val;

        // Update index to that of parent
        index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //    cout << BITree[i] << " ";
}

return BITree;
}

// Driver program to test above functions
int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
        << getSum(BITree, n, 5);

    // Let us test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); //Update BIT for above change in arr[]

    cout << "\nSum of elements in arr[0..5] after update is "
        << getSum(BITree, n, 5);

    return 0;
}

```

Output:

```

Sum of elements in arr[0..5] is 12
Sum of elements in arr[0..5] after update is 18

```

Can we extend the Binary Indexed Tree for range Sum in Logn time?

This is simple to answer. The rangeSum(l, r) can be obtained as getSum(r) - getSum(l-1).

Applications:

Used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case. See [this](#) for more details.

References:

http://en.wikipedia.org/wiki/Fenwick_tree

<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees>

Search in a row wise and column wise sorted matrix

Given an $n \times n$ matrix, where every row and column is sorted in increasing order. Given a number x , how to decide whether this x is in the matrix. The designed algorithm should have linear time complexity.

Thanks to [devendraiit](#) for suggesting below approach.

- 1) Start with top right element
- 2) Loop: compare this element e with x
 - .i) if they are equal then return its position
 - ii) $e < x$ then move it to down (if out of bound of matrix then break return false) ..iii) $e > x$ then move it to left (if out of bound of matrix then break return false)
- 3) repeat the i), ii) and iii) till you find element or returned false

Implementation:

```
#include<stdio.h>

/* Searches the element x in mat[][] . If the element is found,
   then prints its position and returns true, otherwise prints
   "not found" and returns false */
int search(int mat[4][4], int n, int x)
{
    int i = 0, j = n-1; //set indexes for top right element
    while ( i < n && j >= 0 )
    {
        if ( mat[i][j] == x )
        {
            printf("\n Found at %d, %d", i, j);
            return 1;
        }
        if ( mat[i][j] > x )
            j--;
        else // if mat[i][j] < x
            i++;
    }

    printf("\n Element not found");
    return 0; // if ( i==n || j== -1 )
}

// driver program to test above function
int main()
{
    int mat[4][4] = { {10, 20, 30, 40},
                      {15, 25, 35, 45},
                      {27, 29, 37, 48},
                      {32, 33, 39, 50},
                  };
    search(mat, 4, 29);
    getchar();
    return 0;
}
```

Time Complexity: $O(n)$

The above approach will also work for $m \times n$ matrix (not only for $n \times n$). Complexity would be $O(m+n)$.

Print a given matrix in spiral form

Given a 2D array, print it in spiral form. See the following examples.

Input:

```
1   2   3   4
5   6   7   8
9  10  11  12
13 14  15  16
```

Output:

```
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

Input:

```
1   2   3   4   5   6
7   8   9   10  11  12
13 14  15  16  17  18
```

Output:

```
1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11
```

Solution:

```
/* This code is adopted from the solution given
 @ http://effprog.blogspot.com/2011/01/spiral-printing-of-two-dimensional.html */

#include <stdio.h>
#define R 3
#define C 6

void spiralPrint(int m, int n, int a[R][C])
{
    int i, k = 0, l = 0;

    /* k - starting row index
       m - ending row index
       l - starting column index
       n - ending column index
       i - iterator
    */
    while (k < m && l < n)
    {
        /* Print the first row from the remaining rows */
        for (i = l; i < n; ++i)
        {
            printf("%d ", a[k][i]);
        }
        k++;

        /* Print the last column from the remaining columns */
        for (i = k; i < m; ++i)
        {
            printf("%d ", a[i][n-1]);
        }
        n--;

        /* Print the last row from the remaining rows */
        if (k < m)
        {
            for (i = n-1; i >= l; --i)
            {
                printf("%d ", a[m-1][i]);
            }
            m--;
        }

        /* Print the first column from the remaining columns */
        if (l < n)
        {
            for (i = m-1; i >= k; --i)
            {
                printf("%d ", a[i][l]);
            }
            l++;
        }
    }
}

/* Driver program to test above functions */
int main()
```

```
{  
    int a[R][C] = { {1, 2, 3, 4, 5, 6},  
                    {7, 8, 9, 10, 11, 12},  
                    {13, 14, 15, 16, 17, 18}  
    };  
  
    spiralPrint(R, C, a);  
    return 0;  
}  
  
/* OUTPUT:  
   1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11  
*/
```

Time Complexity: Time complexity of the above solution is O(mn).

A Boolean Matrix Question

Given a boolean matrix $\text{mat}[M][N]$ of size $M \times N$, modify it such that if a matrix cell $\text{mat}[i][j]$ is 1 (or true) then make all the cells of i th row and j th column as 1.

Example 1

The matrix

```
1 0  
0 0
```

should be changed to following

```
1 1  
1 0
```

Example 2

The matrix

```
0 0 0  
0 0 1
```

should be changed to following

```
0 0 1  
1 1 1
```

Example 3

The matrix

```
1 0 0 1  
0 0 1 0  
0 0 0 0
```

should be changed to following

```
1 1 1 1  
1 1 1 1  
1 0 1 1
```

Method 1 (Use two temporary arrays)

- 1) Create two temporary arrays $\text{row}[M]$ and $\text{col}[N]$. Initialize all values of $\text{row}[]$ and $\text{col}[]$ as 0.
- 2) Traverse the input matrix $\text{mat}[M][N]$. If you see an entry $\text{mat}[i][j]$ as true, then mark $\text{row}[i]$ and $\text{col}[j]$ as true.
- 3) Traverse the input matrix $\text{mat}[M][N]$ again. For each entry $\text{mat}[i][j]$, check the values of $\text{row}[i]$ and $\text{col}[j]$. If any of the two values ($\text{row}[i]$ or $\text{col}[j]$) is true, then mark $\text{mat}[i][j]$ as true.

Thanks to [Dixit Sethi](#) for suggesting this method.

```
#include <stdio.h>  
#define R 3  
#define C 4  
  
void modifyMatrix(bool mat[R][C])  
{  
    bool row[R];  
    bool col[C];  
  
    int i, j;  
  
    /* Initialize all values of row[] as 0 */  
    for (i = 0; i < R; i++)  
    {  
        row[i] = 0;  
    }  
  
    /* Initialize all values of col[] as 0 */  
    for (i = 0; i < C; i++)  
    {  
        col[i] = 0;  
    }  
  
    /* Store the rows and columns to be marked as 1 in row[] and col[]  
       arrays respectively */  
    for (i = 0; i < R; i++)  
    {  
        for (j = 0; j < C; j++)  
        {  
            if (mat[i][j] == 1)  
            {  
                row[i] = 1;  
                col[j] = 1;  
            }  
        }  
    }  
}
```

```

        }

    }

/* Modify the input matrix mat[] using the above constructed row[] and
   col[] arrays */
for (i = 0; i < R; i++)
{
    for (j = 0; j < C; j++)
    {
        if (row[i] == 1 || col[j] == 1)
        {
            mat[i][j] = 1;
        }
    }
}

/* A utility function to print a 2D matrix */
void printMatrix(bool mat[R][C])
{
    int i, j;
    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

/* Driver program to test above functions */
int main()
{
    bool mat[R][C] = { {1, 0, 0, 1},
                      {0, 0, 1, 0},
                      {0, 0, 0, 0},
    };

    printf("Input Matrix \n");
    printMatrix(mat);

    modifyMatrix(mat);

    printf("Matrix after modification \n");
    printMatrix(mat);

    return 0;
}

```

Output:

```

Input Matrix
1 0 0 1
0 0 1 0
0 0 0 0
Matrix after modification
1 1 1 1
1 1 1 1
1 0 1 1

```

Time Complexity: O(M*N)

Auxiliary Space: O(M + N)

Method 2 (A Space Optimized Version of Method 1)

This method is a space optimized version of above method 1. This method uses the first row and first column of the input matrix in place of the auxiliary arrays row[] and col[] of method 1. So what we do is: first take care of first row and column and store the info about these two in two flag variables rowFlag and colFlag. Once we have this info, we can use first row and first column as auxiliary arrays and apply method 1 for submatrix (matrix excluding first row and first column) of size (M-1)*(N-1).

- 1) Scan the first row and set a variable rowFlag to indicate whether we need to set all 1s in first row or not.
- 2) Scan the first column and set a variable colFlag to indicate whether we need to set all 1s in first column or not.
- 3) Use first row and first column as the auxiliary arrays row[] and col[] respectively, consider the matrix as submatrix starting from second row and second column and apply method 1.
- 4) Finally, using rowFlag and colFlag, update first row and first column if needed.

Time Complexity: O(M*N)

Auxiliary Space: O(1)

Thanks to [Sidh](#) for suggesting this method.

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

Input:

```
{0, 1, 0, 0, 1}
{1, 0, 1, 1, 0}
{0, 1, 0, 0, 1}
{1, 1, 1, 0, 0}
```

Output:

```
0 1 0 0 1
1 0 1 1 0
1 1 1 0 0
```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, dont print it. If the current row doesnt match with any row, print it.

Time complexity: O(ROW^2 x COL)

Auxiliary Space: O(1)

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: O(ROW x COL + ROW x log(ROW))

Auxiliary Space: O(ROW)

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, dont print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
//Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert ( &( (*root)->child[ M[row][col] ] ), M, row, col+1 );

    else // If all entries of this row are processed
```

```

    {
        // unique row found, return 1
        if ( !( *root)->isEndOfCol ) )
            return (*root)->isEndOfCol = 1;

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow( int (*M) [COL], int row )
{
    int i;
    for( i = 0; i < COL; ++i )
        printf( "%d ", M[row][i] );
    printf("\n");
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows( int (*M) [COL] )
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for ( i = 0; i < ROW; ++i )
        // insert row to TRIE
        if ( insert(&root, M, i, 0) )
            // unique row found, print it
            printRow( M, i );
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{0, 1, 0, 0, 1},
                      {1, 0, 1, 1, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 1, 0, 0}
    };

    findUniqueRows( M );

    return 0;
}

```

Time complexity: O(ROW x COL)

Auxiliary Space: O(ROW x COL)

This method has better time complexity. Also, relative order of rows is maintained while printing.

Maximum size square sub-matrix with all 1s

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

The maximum square sub-matrix with all set bits is

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

Algorithm:

Let the given binary matrix be $M[R][C]$. The idea of the algorithm is to construct an auxiliary size matrix $S[][]$ in which each entry $S[i][j]$ represents size of the square sub-matrix with all 1s including $M[i][j]$ where $M[i][j]$ is the rightmost and bottommost entry in sub-matrix.

- 1) Construct a sum matrix $S[R][C]$ for the given $M[R][C]$.
 - a) Copy first row and first columns as it is from $M[][]$ to $S[][]$
 - b) For other entries, use following expressions to construct $S[][]$
If $M[i][j]$ is 1 then
 $S[i][j] = \min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1$
Else /*If $M[i][j]$ is 0*/
 $S[i][j] = 0$
- 2) Find the maximum entry in $S[R][C]$
- 3) Using the value and coordinates of maximum entry in $S[i]$, print sub-matrix of $M[][]$

For the given $M[R][C]$ in above example, constructed $S[R][C]$ would be:

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 2 | 2 | 0 |
| 1 | 2 | 2 | 3 | 1 |
| 0 | 0 | 0 | 0 | 0 |

The value of maximum entry in above matrix is 3 and coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

```
#include<stdio.h>
#define bool int
#define R 6
#define C 5

void printMaxSubSquare(bool M[R][C])
{
    int i,j;
    int S[R][C];
    int max_of_s, max_i, max_j;

    /* Set first column of S[][]*/
    for(i = 0; i < R; i++)
        S[i][0] = M[i][0];

    /* Set first row of S[][]*/
    for(j = 0; j < C; j++)
        S[0][j] = M[0][j];

    /* Construct other entries of S[][]*/
    for(i = 1; i < R; i++)
    {
        for(j = 1; j < C; j++)
        {
            if(M[i][j] == 1)
                S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1;
            else
                S[i][j] = 0;
        }
    }
}
```

```

/* Find the maximum entry, and indexes of maximum entry
   in S[][] */
max_of_s = S[0][0]; max_i = 0; max_j = 0;
for(i = 0; i < R; i++)
{
    for(j = 0; j < C; j++)
    {
        if(max_of_s < S[i][j])
        {
            max_of_s = S[i][j];
            max_i = i;
            max_j = j;
        }
    }
}

printf("\n Maximum size sub-matrix is: \n");
for(i = max_i; i > max_i - max_of_s; i--)
{
    for(j = max_j; j > max_j - max_of_s; j--)
    {
        printf("%d ", M[i][j]);
    }
    printf("\n");
}
}

/* UTILITY FUNCTIONS */
/* Function to get minimum of three values */
int min(int a, int b, int c)
{
    int m = a;
    if (m > b)
        m = b;
    if (m > c)
        m = c;
    return m;
}

/* Driver function to test above functions */
int main()
{
    bool M[R][C] = {{0, 1, 1, 0, 1},
                    {1, 1, 0, 1, 0},
                    {0, 1, 1, 1, 0},
                    {1, 1, 1, 1, 0},
                    {1, 1, 1, 1, 1},
                    {0, 0, 0, 0, 0}};

    printMaxSubSquare(M);
    getchar();
}

```

Time Complexity: O(m*n) where m is number of rows and n is number of columns in the given matrix.

Auxiliary Space: O(m*n) where m is number of rows and n is number of columns in the given matrix.

Algorithmic Paradigm: Dynamic Programming

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

Input:

```
{0, 1, 0, 0, 1}
{1, 0, 1, 1, 0}
{0, 1, 0, 0, 1}
{1, 1, 1, 0, 0}
```

Output:

```
0 1 0 0 1
1 0 1 1 0
1 1 1 0 0
```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, dont print it. If the current row doesnt match with any row, print it.

Time complexity: O(ROW^2 x COL)

Auxiliary Space: O(1)

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: O(ROW x COL + ROW x log(ROW))

Auxiliary Space: O(ROW)

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, dont print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
//Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert ( &( (*root)->child[ M[row][col] ] ), M, row, col+1 );

    else // If all entries of this row are processed
```

```

    {
        // unique row found, return 1
        if ( !( *root)->isEndOfCol ) )
            return (*root)->isEndOfCol = 1;

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow( int (*M) [COL], int row )
{
    int i;
    for( i = 0; i < COL; ++i )
        printf( "%d ", M[row][i] );
    printf("\n");
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows( int (*M) [COL] )
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for ( i = 0; i < ROW; ++i )
        // insert row to TRIE
        if ( insert(&root, M, i, 0) )
            // unique row found, print it
            printRow( M, i );
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{0, 1, 0, 0, 1},
                      {1, 0, 1, 1, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 1, 0, 0}
    };

    findUniqueRows( M );

    return 0;
}

```

Time complexity: O(ROW x COL)

Auxiliary Space: O(ROW x COL)

This method has better time complexity. Also, relative order of rows is maintained while printing.

Inplace M x N size matrix transpose | Updated

About four months of gap (missing GFG), a new post. Given an M x N matrix, transpose the matrix without auxiliary memory. It is easy to transpose matrix using an auxiliary array. If the matrix is symmetric in size, we can transpose the matrix inplace by mirroring the 2D array across its diagonal (try yourself). How to transpose an arbitrary size matrix inplace? See the following matrix,

```
a b c      a d g j  
d e f  ==>  b e h k  
g h i      c f i l  
j k l
```

As per 2D numbering in C/C++, corresponding location mapping looks like,

| Org element | New |
|-------------|------|
| 0 | a 0 |
| 1 | b 4 |
| 2 | c 8 |
| 3 | d 1 |
| 4 | e 5 |
| 5 | f 9 |
| 6 | g 2 |
| 7 | h 6 |
| 8 | i 10 |
| 9 | j 3 |
| 10 | k 7 |
| 11 | l 11 |

Note that the first and last elements stay in their original location. We can easily see the transformation forms few permutation cycles. $1 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 3 \rightarrow 1$. Total 5 elements form the cycle $2 \rightarrow 8 \rightarrow 10 \rightarrow 7 \rightarrow 6 \rightarrow 2$. Another 5 elements form the cycle 0 Self cycle 11 Self cycle. From the above example, we can easily devise an algorithm to move the elements along these cycles. *How can we generate permutation cycles?* Number of elements in both the matrices are constant, given by $N = R * C$, where R is row count and C is column count. An element at location ol (old location in $R \times C$ matrix), moved to nl (new location in $C \times R$ matrix). We need to establish relation between ol , nl , R and C . Assume $ol = A[or]$ [oc]. In C/C++ we can calculate the element address as,

```
ol = or * C + oc (ignore base reference for simplicity)
```

It is to be moved to new location nl in the transposed matrix, say $nl = A[nr][nc]$, or in C/C++ terms

```
nl = nr * R + nc (R - column count, C is row count as the matrix is transposed)
```

Observe, $nr = oc$ and $nc = or$, so replacing these for nl ,

```
nl = oc * R + or -----> [eq 1]
```

after solving for relation between ol and nl , we get

$$\begin{aligned} ol &= or \times C + oc \\ ol \times R &= or \times C \times R + oc \times R \\ &= or \times N + oc \times R \quad (\text{from the fact } R * C = N) \\ &= or \times N + (nl - or) \quad \text{--- from [eq 1]} \\ &= or \times (N-1) + nl \end{aligned}$$

OR,

```
nl = ol * R - or * (N-1)
```

Note that the values of nl and ol never go beyond $N-1$, so considering modulo division on both the sides by $(N-1)$, we get the following based on properties of congruence,

$$\begin{aligned} nl \bmod (N-1) &= (ol \times R - or \times (N-1)) \bmod (N-1) \\ &= (ol \times R) \bmod (N-1) - or \times (N-1) \bmod (N-1) \\ &= ol \times R \bmod (N-1), \text{ since second term evaluates to zero} \\ nl &= (ol \times R) \bmod (N-1), \text{ since } nl \text{ is always less than } N-1 \end{aligned}$$

A curious reader might have observed the significance of above relation. Every location is scaled by a factor of R (row size). It is obvious from the matrix that every location is displaced by scaled factor of R. The actual multiplier depends on congruence class of $(N-1)$, i.e. the multiplier can be both -ve and +ve value of the congruent class. Hence every location transformation is simple modulo division. These modulo divisions form cyclic permutations. We need some book keeping information to keep track of already moved elements. Here is code for inplace matrix transformation,

```
// Program for in-place matrix transpose
#include <stdio.h>
#include <iostream>
#include <bitset>
#define HASH_SIZE 128
```

```

using namespace std;

// A utility function to print a 2D array of size nr x nc and base address A
void Print2DArray(int *A, int nr, int nc)
{
    for(int r = 0; r < nr; r++)
    {
        for(int c = 0; c < nc; c++)
            printf("%4d", *(A + r*nc + c));

        printf("\n");
    }

    printf("\n\n");
}

// Non-square matrix transpose of matrix of size r x c and base address A
void MatrixInplaceTranspose(int *A, int r, int c)
{
    int size = r*c - 1;
    int t; // holds element to be replaced, eventually becomes next element to move
    int next; // location of 't' to be moved
    int cycleBegin; // holds start of cycle
    int i; // iterator
    bitset<HASH_SIZE> b; // hash to mark moved elements

    b.reset();
    b[0] = b[size] = 1;
    i = 1; // Note that A[0] and A[size-1] won't move
    while (i < size)
    {
        cycleBegin = i;
        t = A[i];
        do
        {
            // Input matrix [r x c]
            // Output matrix 1
            // i_new = (i*r)%(N-1)
            next = (i*r)%size;
            swap(A[next], t);
            b[i] = 1;
            i = next;
        }
        while (i != cycleBegin);

        // Get Next Move (what about querying random location?)
        for (i = 1; i < size && b[i]; i++)
            ;
        cout << endl;
    }
}

// Driver program to test above function
int main(void)
{
    int r = 5, c = 6;
    int size = r*c;
    int *A = new int[size];

    for(int i = 0; i < size; i++)
        A[i] = i+1;

    Print2DArray(A, r, c);
    MatrixInplaceTranspose(A, r, c);
    Print2DArray(A, c, r);

    delete[] A;
    return 0;
}

```

Output:

```

1   2   3   4   5   6
7   8   9   10  11  12
13  14  15  16  17  18
19  20  21  22  23  24
25  26  27  28  29  30

```

```

1   7   13  19  25

```

```

2   8   14   20   26
3   9   15   21   27
4  10   16   22   28
5  11   17   23   29
6  12   18   24   30

```

Extension: 17 March 2013 Some [readers](#) identified similarity between the matrix transpose and [string transformation](#). Without much theory I am presenting the problem and solution. In given array of elements like [a1b2c3d4e5f6g7h8i9j1k2l3m4]. Convert it to [abcdefghijklmnl234567891234]. The program should run inplace. What we need is an inplace transpose. Given below is code.

```

#include <stdio.h>
#include <iostream>
#include <bitset>
#define HASH_SIZE 128

using namespace std;

typedef char data_t;

void Print2DArray(char A[], int nr, int nc) {
    int size = nr*nc;
    for(int i = 0; i < size; i++)
        printf("%4c", *(A + i));

    printf("\n");
}

void MatrixTransposeInplaceArrangement(data_t A[], int r, int c) {
    int size = r*c - 1;
    data_t t; // holds element to be replaced, eventually becomes next element to move
    int next; // location of 't' to be moved
    int cycleBegin; // holds start of cycle
    int i; // iterator
    bitset<HASH_SIZE> b; // hash to mark moved elements

    b.reset();
    b[0] = b[size] = 1;
    i = 1; // Note that A[0] and A[size-1] won't move
    while( i < size ) {
        cycleBegin = i;
        t = A[i];
        do {
            // Input matrix [r x c]
            // Output matrix 1
            // i_new = (i*r)%size
            next = (i*r)%size;
            swap(A[next], t);
            b[i] = 1;
            i = next;
        } while( i != cycleBegin );

        // Get Next Move (what about querying random location?)
        for(i = 1; i < size && b[i]; i++)
            ;
        cout << endl;
    }
}

void Fill(data_t buf[], int size) {
    // Fill abcd ...
    for(int i = 0; i < size; i++)
        buf[i] = 'a'+i;

    // Fill 0123 ...
    buf += size;
    for(int i = 0; i < size; i++)
        buf[i] = '0'+i;
}

void TestCase_01(void) {
    int r = 2, c = 10;
    int size = r*c;
    data_t *A = new data_t[size];

    Fill(A, c);

    Print2DArray(A, r, c), cout << endl;
    MatrixTransposeInplaceArrangement(A, r, c);
    Print2DArray(A, c, r), cout << endl;

    delete[] A;
}

```

```
}
```

```
int main() {
```

```
    TestCase_01();
```

```
    return 0;
```

```
}
```

The post is incomplete without mentioning two links.

1. Aashish covered good theory behind cycle leader algorithm. See his post on [string transformation](#).

2. As usual, [Sambasiva](#) demonstrated his exceptional skills in recursion to the [problem](#). Ensure to understand his solution.

Print Matrix Diagonally

Given a 2D matrix, print all elements of the given matrix in diagonal order. For example, consider the following 5 X 4 input matrix.

| | | | |
|----|----|----|----|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 |

Diagonal printing of the above matrix is

| | | | |
|----|----|----|---|
| 1 | | | |
| 5 | 2 | | |
| 9 | 6 | 3 | |
| 13 | 10 | 7 | 4 |
| 17 | 14 | 11 | 8 |
| 18 | 15 | 12 | |
| 19 | 16 | | |
| 20 | | | |

Following is C++ code for diagonal printing.

The diagonal printing of a given matrix $\text{matrix}[\text{ROW}][\text{COL}]$ always has $\text{ROW} + \text{COL} - 1$ lines in output

```
#include <stdio.h>
#include <stdlib.h>

#define ROW 5
#define COL 4

// A utility function to find min of two integers
int min(int a, int b)
{ return (a < b)? a: b; }

// A utility function to find min of three integers
int min(int a, int b, int c)
{ return min(min(a, b), c);}

// A utility function to find max of two integers
int max(int a, int b)
{ return (a > b)? a: b; }

// The main function that prints given matrix in diagonal order
void diagonalOrder(int matrix[][][COL])
{
    // There will be  $\text{ROW}+\text{COL}-1$  lines in the output
    for (int line=1; line<=(ROW + COL -1); line++)
    {
        /* Get column index of the first element in this line of output.
           The index is 0 for first ROW lines and line - ROW for remaining
           lines */
        int start_col =  max(0, line-ROW);

        /* Get count of elements in this line. The count of elements is
           equal to minimum of line number,  $\text{COL}-\text{start\_col}$  and  $\text{ROW}$  */
        int count = min(line, (COL-start_col), ROW);

        /* Print elements of this line */
        for (int j=0; j<count; j++)
            printf("%5d ", matrix[min(ROW, line)-j-1][start_col+j]);

        /* Print elements of next diagonal on next line */
        printf("\n");
    }
}

// Utility function to print a matrix
void printMatrix(int matrix[ROW] [COL])
{
    for (int i=0; i< ROW; i++)
    {
        for (int j=0; j<COL; j++)
            printf("%5d ", matrix[i][j]);
        printf("\n");
    }
}

// Driver program to test above functions
```

```

int main()
{
    int M[ROW][COL] = {{1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12},
                        {13, 14, 15, 16},
                        {17, 18, 19, 20},
                        };
    printf ("Given matrix is \n");
    printMatrix(M);

    printf ("\nDiagonal printing of matrix is \n");
    diagonalOrder(M);
    return 0;
}

```

Output:

```

Given matrix is
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
17  18  19  20

```

```

Diagonal printing of matrix is
 1
 5   2
 9   6   3
13  10   7   4
17  14   11  8
18  15   12
19  16
20

```

Below is an **Alternate Method** to solve the above problem.

```

Matrix =>   1   2   3   4
              5   6   7   8
              9  10  11  12
             13  14  15  16
             17  18  19  20

```

Observe the sequence

```

 1 /  2 /  3 /  4
 / 5 /  6 /  7 /  8
 /  9 / 10 / 11 / 12
 / 13 / 14 / 15 / 16
 / 17 / 18 / 19 / 20

```

```

#include<bits/stdc++.h>
#define R 5
#define C 4
using namespace std;

bool isValid(int i, int j)
{
    if (i < 0 || i >= R || j >= C || j < 0) return false;
    return true;
}

void diagonalOrder(int arr[][C])
{
    /* through this for loop we choose each element of first column
       as starting point and print diagonal starting at it.
       arr[0][0], arr[1][0]....arr[R-1][0] are all starting points */
    for (int k = 0; k < R; k++)
    {
        cout << arr[k][0] << " ";
        int i = k-1; // set row index for next point in diagonal
        int j = 1; // set column index for next point in diagonal

        /* Print Diagonally upward */
        while (isValid(i,j))
        {
            cout << arr[i][j] << " ";
            i--;
            j++; // move in upright direction
        }
    }
}

```

```

        }
        cout << endl;
    }

/* through this for loop we choose each element of last row
   as starting point (except the [0][c-1] it has already been
   processed in previous for loop) and print diagonal starting at it.
   arr[R-1][0], arr[R-1][1]....arr[R-1][c-1] are all starting points */

//Note : we start from k = 1 to C-1;
for (int k = 1; k < C; k++)
{
    cout << arr[R-1][k] << " ";
    int i = R-2; // set row index for next point in diagonal
    int j = k+1; // set column index for next point in diagonal

    /* Print Diagonally upward */
    while (isValid(i,j))
    {
        cout << arr[i][j] << " ";
        i--;
        j++; // move in upright direction
    }
    cout << endl;
}

// Driver program to test above
int main()
{
    int arr[][][C] = {{1, 2, 3, 4},
                      {5, 6, 7, 8},
                      {9, 10, 11, 12},
                      {13, 14, 15, 16},
                      {17, 18, 19, 20},
    };
    diagonalOrder(arr);
    return 0;
}

```

Output:

```

1
5 2
9 6 3
13 10 7 4
17 14 11 8
18 15 12
19 16
20

```

Thanks to Gaurav Ahirwar for suggesting this method.

Dynamic Programming | Set 27 (Maximum sum rectangle in a 2D matrix)

Given a 2D array, find the maximum sum subarray in it. For example, in the following 2D array, the maximum sum subarray is highlighted with blue rectangle and sum of this subarray is 29.

| | | | | |
|----|----|----|----|-----|
| 1 | 2 | -1 | -4 | -20 |
| -8 | -3 | 4 | 2 | 1 |
| 3 | 8 | 10 | 1 | 3 |
| -4 | -1 | 1 | 7 | -6 |

This problem is mainly an extension of [Largest Sum Contiguous Subarray for 1D array](#).

The **naive solution** for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be $O(n^4)$.

Kadane's algorithm for 1D array can be used to reduce the time complexity to $O(n^3)$. The idea is to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sum of elements in every row from left to right and store these sums in an array say $\text{temp}[]$. So $\text{temp}[i]$ indicates sum of elements from left to right in row i . If we apply Kadane's 1D algorithm on $\text{temp}[]$, and get the maximum sum subarray of temp , this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far.

```
// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
#define ROW 4
#define COL 5

// Implementation of Kadane's algorithm for 1D array. The function returns the
// maximum sum and stores starting and ending indexes of the maximum sum subarray
// at addresses pointed by start and finish pointers respectively.
int kadane(int* arr, int* start, int* finish, int n)
{
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i)
    {
        sum += arr[i];
        if (sum < 0)
        {
            sum = 0;
            local_start = i+1;
        }
        else if (sum > maxSum)
        {
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

    // There is at-least one non-negative number
    if (*finish != -1)
        return maxSum;

    // Special Case: When all numbers in arr[] are negative
    maxSum = arr[0];
    *start = *finish = 0;
```

```

// Find the maximum element in array
for (i = 1; i < n; i++)
{
    if (arr[i] > maxSum)
    {
        maxSum = arr[i];
        *start = *finish = i;
    }
}
return maxSum;
}

// The main function that finds maximum sum rectangle in M[][]
void findMaxSum(int M[][])
{
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left)
    {
        // Initialize all elements of temp as 0
        memset(temp, 0, sizeof(temp));

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right)
        {
            // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            // Find the maximum sum subarray in temp[]. The kadane() function
            // also sets values of start and finish. So 'sum' is sum of
            // rectangle between (start, left) and (finish, right) which is the
            // maximum sum with boundary columns strictly as left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then update
            // maxSum and other output values
            if (sum > maxSum)
            {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

    // Print final values
    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
    printf("Max sum is: %d\n", maxSum);
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{1, 2, -1, -4, -20},
                      {-8, -3, 4, 2, 1},
                      {3, 8, 10, 1, 3},
                      {-4, -1, 1, 7, -6}};
}

findMaxSum(M);

return 0;
}

```

Output:

```
(Top, Left) (1, 1)
(Bottom, Right) (3, 3)
Max sum is: 29
```

Time Complexity: O(n^3)

Divide and Conquer | Set 5 (Strassens Matrix Multiplication)

Given two square matrices A and B of size $n \times n$ each, find their multiplication matrix.

Naive Method

Following is a simple way to multiply two matrices.

```
void multiply(int A[][][N], int B[][][N], int C[][][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is $O(N^3)$.

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

- 1) Divide matrices A and B in 4 sub-matrices of size $N/2 \times N/2$ as shown in the below diagram
- 2) Calculate following values recursively. $ae + bg$, $af + bh$, $ce + dg$ and $cf + dh$.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size $N \times N$
a, b, c and d are submatrices of A, of size $N/2 \times N/2$
e, f, g and h are submatrices of B, of size $N/2 \times N/2$

In the above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of above method is $O(N^3)$
which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassens method** is to reduce the number of recursive calls to 7. Strassens method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $N/2 \times N/2$ as shown in the above diagram, but in Strassens method, the four sub-matrices of result are calculated using following formulae.

$$\begin{array}{ll}
 p1 = a(f - h) & p2 = (a + b)h \\
 p3 = (c + d)e & p4 = d(g - e) \\
 p5 = (a + d)(e + h) & p6 = (b - d)(g + h) \\
 p7 = (a - c)(e + f) &
 \end{array}$$

The $A \times B$ can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array} \right] \times \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array} \right] = \left[\begin{array}{c|c} p5 + p4 - p2 + p6 & p1 + p2 \\ \hline p3 + p4 & p1 + p5 - p3 - p7 \end{array} \right]$$

A B C

A , B and C are square matrices of size $N \times N$

a , b , c and d are submatrices of A , of size $N/2 \times N/2$

e , f , g and h are submatrices of B , of size $N/2 \times N/2$

$p1$, $p2$, $p3$, $p4$, $p5$, $p6$ and $p7$ are submatrices of size $N/2 \times N/2$

Time Complexity of Strassens Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of above method is

$O(N^{\log 7})$ which is approximately $O(N^{2.8074})$

Generally Strassens Method is not preferred for practical applications for following reasons.

- 1) The constants used in Strassens method are high and for a typical application Naive method works better.
- 2) For Sparse matrices, there are better methods especially designed for them.
- 3) The submatrices in recursion take extra space.
- 4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassens algorithm than in Naive Method (Source: [CLRS Book](#))

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

<https://www.youtube.com/watch?v=LOLebQ8nKHA>

<https://www.youtube.com/watch?v=QXY4RskLQcI>

Create a matrix with alternating rectangles of O and X

Write a code which inputs two numbers m and n and creates a matrix of size m x n (m rows and n columns) in which every elements is either X or 0. The Xs and 0s must be filled alternatively, the matrix should have outermost rectangle of Xs, then a rectangle of 0s, then a rectangle of Xs, and so on.

Examples:

Input: m = 3, n = 3
Output: Following matrix

X X X
X 0 X
X X X

Input: m = 4, n = 5
Output: Following matrix
X X X X X
X 0 0 0 X
X 0 0 0 X
X X X X X

Input: m = 5, n = 5
Output: Following matrix
X X X X X
X 0 0 0 X
X 0 X 0 X
X 0 0 0 X
X X X X X

Input: m = 6, n = 7
Output: Following matrix
X X X X X X X
X 0 0 0 0 0 X
X 0 X X X 0 X
X 0 X X X 0 X
X 0 0 0 0 0 X
X X X X X X X

This question was asked in campus recruitment of Shreepartners Gurgaon. I followed the following approach.

- 1) Use the [code for Printing Matrix in Spiral form](#)
- 2) Instead of printing the array, inserted the element X or 0 alternatively in the array.

Following is C implementation of the above approach.

```
#include <stdio.h>

// Function to print alternating rectangles of 0 and X
void fill0X(int m, int n)
{
    /* k - starting row index
     * m - ending row index
     * l - starting column index
     * n - ending column index
     * i - iterator */
    int i, k = 0, l = 0;

    // Store given number of rows and columns for later use
    int r = m, c = n;

    // A 2D array to store the output to be printed
    char a[m][n];
    char x = 'X'; // Initialize the character to be stored in a[][]

    // Fill characters in a[][] in spiral form. Every iteration fills
    // one rectangle of either Xs or Os
    while (k < m && l < n)
    {
        /* Fill the first row from the remaining rows */
        for (i = l; i < n; ++i)
            a[k][i] = x;
        k++;

        /* Fill the last column from the remaining columns */
        for (i = k; i < m; ++i)
            a[i][n-1] = x;
        n--;
    }
}
```

```

/* Fill the last row from the remaining rows */
if (k < m)
{
    for (i = n-1; i >= l; --i)
        a[m-1][i] = x;
    m--;
}

/* Print the first column from the remaining columns */
if (l < n)
{
    for (i = m-1; i >= k; --i)
        a[i][l] = x;
    l++;
}

// Flip character for next iteration
x = (x == '0')? 'X': '0';

// Print the filled matrix
for (i = 0; i < r; i++)
{
    for (int j = 0; j < c; j++)
        printf("%c ", a[i][j]);
    printf("\n");
}
}

/* Driver program to test above functions */
int main()
{
    puts("Output for m = 5, n = 6");
    fill0X(5, 6);

    puts("\nOutput for m = 4, n = 4");
    fill0X(4, 4);

    puts("\nOutput for m = 3, n = 4");
    fill0X(3, 4);

    return 0;
}

```

Output:

Output for m = 5, n = 6

```

X X X X X
X 0 0 0 0 X
X 0 X X 0 X
X 0 0 0 0 X
X X X X X X

```

Output for m = 4, n = 4

```

X X X X
X 0 0 X
X 0 0 X
X X X X

```

Output for m = 3, n = 4

```

X X X X
X 0 0 X
X X X X

```

Time Complexity: O(mn)

Auxiliary Space: O(mn)

Please suggest if someone has a better solution which is more efficient in terms of space and time.

Find the row with maximum number of 1s

Given a boolean 2D array, where each row is sorted. Find the row with the maximum number of 1s.

Example

```
Input matrix
0 1 1 1
0 0 1 1
1 1 1 1 // this row has maximum 1s
0 0 0 0
```

Output: 2

A **simple method** is to do a row wise traversal of the matrix, count the number of 1s in each row and compare the count with max. Finally, return the index of row with maximum 1s. The time complexity of this method is $O(m*n)$ where m is number of rows and n is number of columns in matrix.

We can do better. Since each row is sorted, we can **use Binary Search** to count of 1s in each row. We find the index of first instance of 1 in each row. The count of 1s will be equal to total number of columns minus the index of first 1.

See the following code for implementation of the above approach.

```
#include <stdio.h>
#define R 4
#define C 4

/* A function to find the index of first index of 1 in a boolean array arr[] */
int first(bool arr[], int low, int high)
{
    if(high >= low)
    {
        // get the middle index
        int mid = low + (high - low)/2;

        // check if the element at middle index is first 1
        if ((mid == 0 || arr[mid-1] == 0) && arr[mid] == 1)
            return mid;

        // if the element is 0, recur for right side
        else if (arr[mid] == 0)
            return first(arr, (mid + 1), high);

        else // If element is not first 1, recur for left side
            return first(arr, low, (mid -1));
    }
    return -1;
}

// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    int max_row_index = 0, max = -1; // Initialize max values

    // Traverse for each row and count number of 1s by finding the index
    // of first 1
    int i, index;
    for (i = 0; i < R; i++)
    {
        index = first (mat[i], 0, C-1);
        if (index != -1 && C-index > max)
        {
            max = C - index;
            max_row_index = i;
        }
    }

    return max_row_index;
}

/* Driver program to test above functions */
int main()
{
    bool mat[R][C] = { {0, 0, 0, 1},
                      {0, 1, 1, 1},
                      {1, 1, 1, 1},
                      {0, 0, 0, 0}
                    };

    printf("Index of row with maximum 1s is %d \n", rowWithMax1s(mat));
```

```
    return 0;
}
```

Output:

```
Index of row with maximum 1s is 2
```

Time Complexity: $O(m\log n)$ where m is number of rows and n is number of columns in matrix.

The above solution **can be optimized further**. Instead of doing binary search in every row, we first check whether the row has more 1s than max so far. If the row has more 1s, then only count 1s in the row. Also, to count 1s in a row, we don't do binary search in complete row, we do search in before the index of last max.

Following is an optimized version of the above solution.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    int i, index;

    // Initialize max using values from first row.
    int max_row_index = 0;
    int max = C - first(mat[0], 0, C-1);

    // Traverse for each row and count number of 1s by finding the index
    // of first 1
    for (i = 1; i < R; i++)
    {
        // Count 1s in this row only if this row has more 1s than
        // max so far
        if (mat[i][C-max-1] == 1)
        {
            // Note the optimization here also
            index = first (mat[i], 0, C-max);

            if (index != -1 && C-index > max)
            {
                max = C - index;
                max_row_index = i;
            }
        }
    }
    return max_row_index;
}
```

The worst case time complexity of the above optimized version is also $O(m\log n)$, the will solution work better on average. Thanks to [Naveen Kumar Singh](#) for suggesting the above solution.

Sources: [this](#) and [this](#)

The worst case of the above solution occurs for a matrix like following

```
0 0 0 0 1
0 0 0 ..0 1 1
0 0 1 1 1
.0 1 1 1 1
```

Following method works in $O(m+n)$ time complexity in worst case.

Step1: Get the index of first (or leftmost) 1 in the first row.

Step2: Do following for every row after the first row

IF the element on left of previous leftmost 1 is 0, ignore this row.

ELSE Move left until a 0 is found. Update the leftmost index to this index and max_row_index to be the current row.

The time complexity is $O(m+n)$ because we can possibly go as far left as we came ahead in the first step.

Following is C++ implementation of this method.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    // Initialize first row as row with max 1s
    int max_row_index = 0;

    // The function first() returns index of first 1 in row 0.
```

```
// Use this index to initialize the index of leftmost 1 seen so far
int j = first(mat[0], 0, C-1) - 1;
if (j == -1) // if 1 is not present in first row
    j = C - 1;

for (int i = 1; i < R; i++)
{
    // Move left until a 0 is found
    while (j >= 0 && mat[i][j] == 1)
    {
        j = j-1; // Update the index of leftmost 1 seen so far
        max_row_index = i; // Update max_row_index
    }
}
return max_row_index;
}
```

Thanks to Tylor, Ankan and Palash for their inputs.

Print all elements in sorted order from row and column wise sorted matrix

Given an $n \times n$ matrix, where every row and column is sorted in non-decreasing order. Print all elements of matrix in sorted order.

Example:

```
Input: mat[][] = { {10, 20, 30, 40},  
                  {15, 25, 35, 45},  
                  {27, 29, 37, 48},  
                  {32, 33, 39, 50},  
                };
```

Output:

```
Elements of matrix in sorted order  
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

We can use [Young Tableau](#) to solve the above problem. The idea is to consider given 2D array as Young Tableau and call extract minimum $O(N)$

```
// A C++ program to Print all elements in sorted order from row and  
// column wise sorted matrix  
#include<iostream>  
#include<climits>  
using namespace std;  
  
#define INF INT_MAX  
#define N 4  
  
// A utility function to youngify a Young Tableau. This is different  
// from standard youngify. It assumes that the value at mat[0][0] is  
// infinite.  
void youngify(int mat[][N], int i, int j)  
{  
    // Find the values at down and right sides of mat[i][j]  
    int downVal = (i+1 < N)? mat[i+1][j]: INF;  
    int rightVal = (j+1 < N)? mat[i][j+1]: INF;  
  
    // If mat[i][j] is the down right corner element, return  
    if (downVal==INF && rightVal==INF)  
        return;  
  
    // Move the smaller of two values (downVal and rightVal) to  
    // mat[i][j] and recur for smaller value  
    if (downVal < rightVal)  
    {  
        mat[i][j] = downVal;  
        mat[i+1][j] = INF;  
        youngify(mat, i+1, j);  
    }  
    else  
    {  
        mat[i][j] = rightVal;  
        mat[i][j+1] = INF;  
        youngify(mat, i, j+1);  
    }  
}  
  
// A utility function to extract minimum element from Young tableau  
int extractMin(int mat[][N])  
{  
    int ret = mat[0][0];  
    mat[0][0] = INF;  
    youngify(mat, 0, 0);  
    return ret;  
}  
  
// This function uses extractMin() to print elements in sorted order  
void printSorted(int mat[][N])  
{  
    cout << "Elements of matrix in sorted order \n";  
    for (int i=0; i<N*N; i++)  
        cout << extractMin(mat) << " ";  
}  
  
// driver program to test above function  
int main()  
{  
    int mat[N][N] = { {10, 20, 30, 40},  
                      {15, 25, 35, 45},  
                    };
```

```

        {27, 29, 37, 48},
        {32, 33, 39, 50},
    };
printSorted(mat);
return 0;
}

```

Output:

```

Elements of matrix in sorted order
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50

```

Time complexity of extract minimum is $O(N)$ and it is called $O(N^2)$ times. Therefore the overall time complexity is $O(N^3)$.

A better solution is to use the [approach used for merging k sorted arrays](#). The idea is to use a Min Heap of size N which stores elements of first column. To do extract minimum. In extract minimum, replace the minimum element with the next element of the row from which the element is extracted. Time complexity of this solution is $O(N^2 \log N)$.

```

// C++ program to merge k sorted arrays of size n each.
#include<iostream>
#include<climits>
using namespace std;

#define N 4

// A min heap node
struct MinHeapNode
{
    int element; // The element to be stored
    int i; // index of the row from which the element is taken
    int j; // index of the next element to be picked from row
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{
    MinHeapNode *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x) { harr[0] = x; MinHeapify(0); }
};

// This function prints elements of a given matrix in non-decreasing
// order. It assumes that mat[][] is sorted row wise sorted.
void printSorted(int mat[][N])
{
    // Create a min heap with k heap nodes. Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[N];
    for (int i = 0; i < N; i++)
    {
        harr[i].element = mat[i][0]; // Store the first element
        harr[i].i = i; // index of row
        harr[i].j = 1; // Index of next element to be stored from row
    }
    MinHeap hp(harr, N); // Create the min heap

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < N*N; count++)

```

```

{
    // Get the minimum element and store it in output
    MinHeapNode root = hp.getMin();

    cout << root.element << " ";

    // Find the next elelement that will replace current
    // root of heap. The next element belongs to same
    // array as the current root.
    if (root.j < N)
    {
        root.element = mat[root.i][root.j];
        root.j += 1;
    }
    // If root was the last element of its array
    else root.element = INT_MAX; //INT_MAX is for infinite

    // Replace root with next element of array
    hp.replaceMin(root);
}
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS
// FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element < harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x; *x = *y; *y = temp;
}

// driver program to test above function
int main()
{
    int mat[N][N] = { {10, 20, 30, 40},
                      {15, 25, 35, 45},
                      {27, 29, 37, 48},
                      {32, 33, 39, 50},
                    };
    printSorted(mat);
    return 0;
}

```

Output:

10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50

Exercise:

Above solutions work for a square matrix. Extend the above solutions to work for an M*N rectangular matrix.

Given an n x n square matrix, find sum of all sub-squares of size k x k

Given an n x n square matrix, find sum of all sub-squares of size k x k where k is smaller than or equal to n.

Examples

Input:

```
n = 5, k = 3
arr[][] = { {1, 1, 1, 1, 1},
            {2, 2, 2, 2, 2},
            {3, 3, 3, 3, 3},
            {4, 4, 4, 4, 4},
            {5, 5, 5, 5, 5},
        };
```

Output:

```
18 18 18
27 27 27
36 36 36
```

Input:

```
n = 3, k = 2
arr[][] = { {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9},
        };
```

Output:

```
12 16
24 28
```

A **Simple Solution** is to one by one pick starting point (leftmost-topmost corner) of all possible sub-squares. Once the starting point is picked, calculate sum of sub-square starting with the picked starting point.

Following is C++ implementation of this idea.

```
// A simple C++ program to find sum of all subsquares of size k x k
#include <iostream>
using namespace std;

// Size of given matrix
#define n 5

// A simple function to find sum of all sub-squares of size k x k
// in a given square matrix of size n x n
void printSumSimple(int mat[][n], int k)
{
    // k must be smaller than or equal to n
    if (k > n) return;

    // row number of first cell in current sub-square of size k x k
    for (int i=0; i<n-k+1; i++)
    {
        // column of first cell in current sub-square of size k x k
        for (int j=0; j<n-k+1; j++)
        {
            // Calculate and print sum of current sub-square
            int sum = 0;
            for (int p=i; p<k+i; p++)
                for (int q=j; q<k+j; q++)
                    sum += mat[p][q];
            cout << sum << " ";
        }

        // Line separator for sub-squares starting with next row
        cout << endl;
    }
}

// Driver program to test above function
int main()
{
    int mat[n][n] = {{1, 1, 1, 1, 1},
                     {2, 2, 2, 2, 2},
                     {3, 3, 3, 3, 3},
                     {4, 4, 4, 4, 4},
                     {5, 5, 5, 5, 5},
    };
    int k = 3;
    printSumSimple(mat, k);
```

```

        return 0;
}
}

```

Output:

```

18 18 18
27 27 27
36 36 36

```

Time complexity of above solution is $O(k^2n^2)$. We can solve this problem in $O(n^2)$ time using a **Tricky Solution**. The idea is to preprocess the given square matrix. In the preprocessing step, calculate sum of all vertical strips of size $k \times 1$ in a temporary square matrix $\text{stripSum}[][]$. Once we have sum of all vertical strips, we can calculate sum of first sub-square in a row as sum of first k strips in that row, and for remaining sub-squares, we can calculate sum in $O(1)$ time by removing the leftmost strip of previous subsquare and adding the rightmost strip of new square.

Following is C++ implementation of this idea.

```

// An efficient C++ program to find sum of all subsquares of size k x k
#include <iostream>
using namespace std;

// Size of given matrix
#define n 5

// A O(n^2) function to find sum of all sub-squares of size k x k
// in a given square matrix of size n x n
void printSumTricky(int mat[][] , int k)
{
    // k must be smaller than or equal to n
    if (k > n) return;

    // 1: PREPROCESSING
    // To store sums of all strips of size k x 1
    int stripSum[n][n];

    // Go column by column
    for (int j=0; j<n; j++)
    {
        // Calculate sum of first k x 1 rectangle in this column
        int sum = 0;
        for (int i=0; i<k; i++)
            sum += mat[i][j];
        stripSum[0][j] = sum;

        // Calculate sum of remaining rectangles
        for (int i=1; i<n-k+1; i++)
        {
            sum += (mat[i+k-1][j] - mat[i-1][j]);
            stripSum[i][j] = sum;
        }
    }

    // 2: CALCULATE SUM of Sub-Squares using stripSum[][]
    for (int i=0; i<n-k+1; i++)
    {
        // Calculate and print sum of first subsquare in this row
        int sum = 0;
        for (int j = 0; j<k; j++)
            sum += stripSum[i][j];
        cout << sum << " ";

        // Calculate sum of remaining squares in current row by
        // removing the leftmost strip of previous sub-square and
        // adding a new strip
        for (int j=1; j<n-k+1; j++)
        {
            sum += (stripSum[i][j+k-1] - stripSum[i][j-1]);
            cout << sum << " ";
        }

        cout << endl;
    }
}

// Driver program to test above function
int main()
{
    int mat[n][n] = {{1, 1, 1, 1, 1},
                     {2, 2, 2, 2, 2},
                     {3, 3, 3, 3, 3},
                     {4, 4, 4, 4, 4},

```

```
    {5, 5, 5, 5, 5},  
};  
int k = 3;  
printSumTricky(mat, k);  
return 0;  
}
```

Output:

```
18 18 18  
27 27 27  
36 36 36
```

Count number of islands where every island is row-wise and column-wise separated

Given a rectangular matrix which has only two possible values X and O. The values X always appear in form of rectangular islands and these islands are always row-wise and column-wise separated by at least one line of Os. Note that islands can only be diagonally adjacent. Count the number of islands in the given matrix.

Examples:

```
mat[M][N] = {{'O', 'O', 'O'},  
              {'X', 'X', 'O'},  
              {'X', 'X', 'O'},  
              {'O', 'O', 'X'},  
              {'O', 'O', 'X'},  
              {'X', 'X', 'O'}  
};
```

Output: Number of islands is 3

```
mat[M][N] = {{'X', 'O', 'O', 'O', 'O', 'O'},  
              {'X', 'O', 'X', 'X', 'X', 'X'},  
              {'O', 'O', 'O', 'O', 'O', 'O'},  
              {'X', 'X', 'X', 'O', 'X', 'X'},  
              {'X', 'X', 'X', 'O', 'X', 'X'},  
              {'O', 'O', 'O', 'O', 'X', 'X'}  
};
```

Output: Number of islands is 4

The idea is to count all top-leftmost corners of given matrix. We can check if a X is top left or not by checking following conditions.

- 1) A X is top of rectangle if the cell just above it is a O
- 2) A X is leftmost of rectangle if the cell just left of it is a O

Note that we must check for both conditions as there may be more than one top cells and more than one leftmost cells in a rectangular island.
Below is C++ implementation of above idea.

```
// A C++ program to count the number of rectangular  
// islands where every island is separated by a line  
  
#include<iostream>  
using namespace std;  
  
// Size of given matrix is M X N  
#define M 6  
#define N 3  
  
// This function takes a matrix of 'X' and 'O'  
// and returns the number of rectangular islands  
// of 'X' where no two islands are row-wise or  
// column-wise adjacent, the islands may be diagonally  
// adjacent  
int countIslands(int mat[][N])  
{  
    int count = 0; // Initialize result  
  
    // Traverse the input matrix  
    for (int i=0; i<M; i++)  
    {  
        for (int j=0; j<N; j++)  
        {  
            // If current cell is 'X', then check  
            // whether this is top-leftmost of a  
            // rectangle. If yes, then increment count  
            if (mat[i][j] == 'X')  
            {  
                if ((i == 0 || mat[i-1][j] == 'O') &&  
                    (j == 0 || mat[i][j-1] == 'O'))  
                    count++;  
            }  
        }  
    }  
  
    return count;  
}  
  
// Driver program to test above function  
int main()  
{  
    int mat[M][N] = {{'O', 'O', 'O'},  
                     {'X', 'X', 'O'},  
                     {'X', 'X', 'O'},  
                     {'O', 'O', 'X'},  
                     {'O', 'O', 'X'},  
                     {'X', 'X', 'O'}};
```

```
        {'O', 'O', 'X'},
        {'X', 'X', 'O'}
    };
cout << "Number of rectangular islands is "
    << countIslands(mat);
return 0;
}
```

Output:

```
Number of rectangular islands is 3
```

Time complexity of this solution is O(MN).

This article is contributed by **Udit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Find a common element in all rows of a given row-wise sorted matrix

Given a matrix where every row is sorted in increasing order. Write a function that finds and returns a common element in all rows. If there is no common element, then returns -1.

Example:

```
Input: mat[4][5] = { {1, 2, 3, 4, 5},
                     {2, 4, 5, 8, 10},
                     {3, 5, 7, 9, 11},
                     {1, 3, 5, 7, 9},
                   };
Output: 5
```

A **O(m*n*n)** simple solution is to take every element of first row and search it in all other rows, till we find a common element. Time complexity of this solution is $O(m*n*n)$ where m is number of rows and n is number of columns in given matrix. This can be improved to $O(m*n*\log n)$ if we use [Binary Search](#) instead of linear search.

We can solve this problem in **O(mn)** time using the approach similar to merge of [Merge Sort](#). The idea is to start from the last column of every row. If elements at all last columns are same, then we found the common element. Otherwise we find the minimum of all last columns. Once we find a minimum element, we know that all other elements in last columns cannot be a common element, so we reduce last column index for all rows except for the row which has minimum value. We keep repeating these steps till either all elements at current last column dont become same, or a last column index reaches 0.

Below is C implementation of above idea.

```
// A C program to find a common element in all rows of a
// row wise sorted array
#include<stdio.h>

// Specify number of rows and columns
#define M 4
#define N 5

// Returns common element in all rows of mat[M][N]. If there is no
// common element, then -1 is returned
int findCommon(int mat[M][N])
{
    // An array to store indexes of current last column
    int column[M];
    int min_row; // To store index of row whose current
                 // last element is minimum

    // Initialize current last element of all rows
    int i;
    for (i=0; i<M; i++)
        column[i] = N-1;

    min_row = 0; // Initialize min_row as first row

    // Keep finding min_row in current last column, till either
    // all elements of last column become same or we hit first column.
    while (column[min_row] >= 0)
    {
        // Find minimum in current last column
        for (i=0; i<M; i++)
        {
            if (mat[i][column[i]] < mat[min_row][column[min_row]] )
                min_row = i;
        }

        // eq_count is count of elements equal to minimum in current last
        // column.
        int eq_count = 0;

        // Travers current last column elements again to update it
        for (i=0; i<M; i++)
        {
            // Decrease last column index of a row whose value is more
            // than minimum.
            if (mat[i][column[i]] > mat[min_row][column[min_row]])
            {
                if (column[i] == 0)
                    return -1;

                column[i] -= 1; // Reduce last column index by 1
            }
        }
    }
}
```

```

        else
            eq_count++;
    }

    // If equal count becomes M, return the value
    if (eq_count == M)
        return mat[min_row][column[min_row]];
}
return -1;
}

// driver program to test above function
int main()
{
    int mat[M][N] = { {1, 2, 3, 4, 5},
                      {2, 4, 5, 8, 10},
                      {3, 5, 7, 9, 11},
                      {1, 3, 5, 7, 9},
                  };
    int result = findCommon(mat);
    if (result == -1)
        printf("No common element");
    else
        printf("Common element is %d", result);
    return 0;
}

```

Output:

Common element is 5

Explanation for working of above code

Let us understand working of above code for following example.

Initially entries in last column array are N-1, i.e., {4, 4, 4, 4}

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

The value of min_row is 0, so values of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 3, 3, 3}.

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

The value of min_row remains 0 and and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 2, 2}.

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

The value of min_row remains 0 and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 1, 2}.

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

Now all values in current last columns of all rows is same, so 5 is returned.

A Hashing Based Solution

We can also use hashing. This solution works even if the rows are not sorted. It can be used to print all common elements.

Step1: Create a Hash Table with all key as distinct elements of row1. Value for all these will be 0.

Step2:

For i = 1 to M-1
For j = 0 to N-1
If (mat[i][j] is already present in Hash Table)
If (And this is not a repetition in current row.
This can be checked by comparing HashTable value with

```
row number)
Update the value of this key in HashTable with current
row number
```

Step3: Iterate over HashTable and print all those keys for
which value = M

Time complexity of the above hashing based solution is $O(MN)$ under the assumption that search and insert in HashTable take $O(1)$ time. Thanks to Nishant for suggesting this solution in a comment below.

Exercise: Given n sorted arrays of size m each, find all common elements in all arrays in $O(mn)$ time.

Given a matrix of O and X, replace O with X if surrounded by X

Given a matrix where every element is either O or X, replace O with X if surrounded by X. A O (or a set of O) is considered to be by surrounded by X if there are X at locations just below, just above, just left and just right of it.

Examples:

```
Input: mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
                     {'X', 'O', 'X', 'X', 'O', 'X'},  
                     {'X', 'X', 'X', 'O', 'O', 'X'},  
                     {'O', 'X', 'X', 'X', 'X', 'X'},  
                     {'X', 'X', 'X', 'O', 'X', 'O'},  
                     {'O', 'O', 'X', 'O', 'O', 'O'},  
                     };
```

```
Output: mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
                      {'X', 'O', 'X', 'X', 'X', 'X'},  
                      {'X', 'X', 'X', 'X', 'X', 'X'},  
                      {'O', 'X', 'X', 'X', 'X', 'X'},  
                      {'X', 'X', 'X', 'O', 'X', 'O'},  
                      {'O', 'O', 'X', 'O', 'O', 'O'},  
                      };
```

```
Input: mat[M][N] = {{'X', 'X', 'X', 'X'},  
                     {'X', 'O', 'X', 'X'},  
                     {'X', 'O', 'O', 'X'},  
                     {'X', 'O', 'X', 'X'},  
                     {'X', 'X', 'O', 'O'},  
                     };
```

```
Input: mat[M][N] = {{'X', 'X', 'X', 'X'},  
                     {'X', 'X', 'X', 'X'},  
                     {'X', 'X', 'X', 'X'},  
                     {'X', 'X', 'X', 'X'},  
                     {'X', 'X', 'O', 'O'},  
                     };
```

This is mainly an application of [Flood-Fill algorithm](#). The main difference here is that a O is not replaced by X if it lies in region that ends on a boundary. Following are simple steps to do this special flood fill.

1) Traverse the given matrix and replace all O with a special character -.

2) Traverse four edges of given matrix and call `floodFill(-, O)` for every - on edges. The remaining - are the characters that indicate Os (in the original matrix) to be replaced by X.

3) Traverse the matrix and replace all -s with Xs.

Let us see steps of above algorithm with an example. Let following be the input matrix.

```
mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
               {'X', 'O', 'X', 'X', 'O', 'X'},  
               {'X', 'X', 'X', 'O', 'O', 'X'},  
               {'O', 'X', 'X', 'X', 'X', 'X'},  
               {'X', 'X', 'X', 'O', 'X', 'O'},  
               {'O', 'O', 'X', 'O', 'O', 'O'},  
               };
```

Step 1: Replace all O with -.

```
mat[M][N] = {{'X', '-', 'X', 'X', 'X', 'X'},  
               {'X', '-', 'X', 'X', ' ', 'X'},  
               {'X', 'X', 'X', ' ', ' ', 'X'},  
               {' ', 'X', 'X', 'X', 'X', 'X'},  
               {'X', 'X', 'X', ' ', 'X', ' '},  
               {' ', ' ', 'X', ' ', ' ', ' '},  
               };
```

Step 2: Call `floodFill(-, O)` for all edge elements with value equals to -

```
mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
               {'X', 'O', 'X', 'X', ' ', 'X'},  
               {'X', 'X', 'X', ' ', ' ', 'X'},  
               {'O', 'X', 'X', 'X', 'X', 'X'},  
               {'X', 'X', 'X', 'O', 'X', 'O'},  
               {'O', 'O', 'X', 'O', 'O', 'O'},  
               };
```

Step 3: Replace all - with X.

```

mat [M] [N] = { {'X', 'O', 'X', 'X', 'X', 'X'},
                {'X', 'O', 'X', 'X', 'X', 'X'},
                {'X', 'X', 'X', 'X', 'X', 'X'},
                {'O', 'X', 'X', 'X', 'X', 'X'},
                {'X', 'X', 'X', 'O', 'X', 'O'},
                {'O', 'O', 'X', 'O', 'O', 'O'} };

```

The following is C++ implementation of above algorithm.

```

// A C++ program to replace all 'O's with 'X''s if surrounded by 'X'
#include<iostream>
using namespace std;

// Size of given matrix is M X N
#define M 6
#define N 6

// A recursive function to replace previous value 'prevV' at '(x, y)'
// and all surrounding values of (x, y) with new value 'newV'.
void floodFillUtil(char mat[][][N], int x, int y, char prevV, char newV)
{
    // Base cases
    if (x < 0 || x >= M || y < 0 || y >= N)
        return;
    if (mat[x][y] != prevV)
        return;

    // Replace the color at (x, y)
    mat[x][y] = newV;

    // Recur for north, east, south and west
    floodFillUtil(mat, x+1, y, prevV, newV);
    floodFillUtil(mat, x-1, y, prevV, newV);
    floodFillUtil(mat, x, y+1, prevV, newV);
    floodFillUtil(mat, x, y-1, prevV, newV);
}

// Returns size of maximum size subsquare matrix
// surrounded by 'X'
int replaceSurrounded(char mat[][][N])
{
    // Step 1: Replace all 'O' with '-'
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            if (mat[i][j] == 'O')
                mat[i][j] = '-';

    // Call floodFill for all '-' lying on edges
    for (int i=0; i<M; i++) // Left side
        if (mat[i][0] == '-')
            floodFillUtil(mat, i, 0, '-', 'O');
    for (int i=0; i<M; i++) // Right side
        if (mat[i][N-1] == '-')
            floodFillUtil(mat, i, N-1, '-', 'O');
    for (int i=0; i<N; i++) // Top side
        if (mat[0][i] == '-')
            floodFillUtil(mat, 0, i, '-', 'O');
    for (int i=0; i<N; i++) // Bottom side
        if (mat[M-1][i] == '-')
            floodFillUtil(mat, M-1, i, '-', 'O');

    // Step 3: Replace all '-' with 'X'
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            if (mat[i][j] == '-')
                mat[i][j] = 'X';
}

// Driver program to test above function
int main()
{
    char mat[][][N] = {{'X', 'O', 'X', 'O', 'X', 'X'}, {
        {'X', 'O', 'X', 'X', 'O', 'X'}, {
        {'X', 'X', 'X', 'O', 'X', 'X'}, {
        {'O', 'X', 'X', 'X', 'X', 'X'}, {
        {'X', 'X', 'X', 'O', 'X', 'O'}}};
}

```

```

        {'O', 'O', 'X', 'O', 'O', 'O'},
    };
replaceSurrounded(mat);

for (int i=0; i<M; i++)
{
    for (int j=0; j<N; j++)
        cout << mat[i][j] << " ";
    cout << endl;
}
return 0;
}

```

Output:

```

X O X O X X
X O X X X X
X X X X X X
O X X X X X
X X X O X O
O O X O O O

```

Time Complexity of the above solution is $O(MN)$. Note that every element of matrix is processed at most three times.

Find the longest path in a matrix with given constraints

Given a $n \times n$ matrix where numbers all numbers are distinct and are distributed from range 1 to n^2 , find the maximum length path (starting from any cell) such that all cells along the path are increasing order with a difference of 1.

We can move in 4 directions from a given cell (i, j) , i.e., we can move to $(i+1, j)$ or $(i, j+1)$ or $(i-1, j)$ or $(i, j-1)$ with the condition that the adjacent cells have a difference of 1.

Example:

```
Input: mat[][] = {{1, 2, 9},  
                  {5, 3, 8},  
                  {4, 6, 7}}
```

Output: 4

The longest path is 6-7-8-9.

Below is Dynamic Programming based C implementation that uses a lookup table $dp[][],$ to check if a problem is already solved or not.

```
#include<bits/stdc++.h>  
#define n 3  
using namespace std;  
  
// Returns length of the longest path beginning with mat[i][j].  
// This function mainly uses lookup table dp[n][n]  
int findLongestFromACell(int i, int j, int mat[n][n], int dp[n][n])  
{  
    // Base case  
    if (i<0 || i>=n || j<0 || j>=n)  
        return 0;  
  
    // If this subproblem is already solved  
    if (dp[i][j] != -1)  
        return dp[i][j];  
  
    // Since all numbers are unique and in range from 1 to n*n,  
    // there is atmost one possible direction from any cell  
    if ((mat[i][j] +1) == mat[i][j+1])  
        return dp[i][j] = 1 + findLongestFromACell(i, j+1, mat, dp);  
  
    if (mat[i][j] +1 == mat[i][j-1])  
        return dp[i][j] = 1 + findLongestFromACell(i, j-1, mat, dp);  
  
    if (mat[i][j] +1 == mat[i-1][j])  
        return dp[i][j] = 1 + findLongestFromACell(i-1, j, mat, dp);  
  
    if (mat[i][j] +1 == mat[i+1][j])  
        return dp[i][j] = 1 + findLongestFromACell(i+1, j, mat, dp);  
  
    // If none of the adjacent fours is one greater  
    return dp[i][j] = 1;  
}  
  
// Returns length of the longest path beginning with any cell  
int finLongestOverAll(int mat[n][n])  
{  
    int result = 1; // Initialize result  
  
    // Create a lookup table and fill all entries in it as -1  
    int dp[n][n];  
    memset(dp, -1, sizeof dp);  
  
    // Compute longest path beginning from all cells  
    for (int i=0; i<n; i++)  
    {  
        for (int j=0; j<n; j++)  
        {  
            if (dp[i][j] == -1)  
                findLongestFromACell(i, j, mat, dp);  
  
            // Update result if needed  
            result = max(result, dp[i][j]);  
        }  
    }  
  
    return result;  
}  
  
// Driver program  
int main()  
{
```

```
int mat[n][n] = {{1, 2, 9},  
                  {5, 3, 8},  
                  {4, 6, 7}};  
cout << "Length of the longest path is "  
      << finLongestOverAll(mat);  
return 0;  
}
```

Output:

Length of the longest path is 4

Time complexity of the above solution is $O(n^2)$. It may seem more at first look. If we take a closer look, we can notice that all values of $dp[i][j]$ are computed only once.

Given a Boolean Matrix, find k such that all elements in kth row are 0 and kth column are 1.

Given a square boolean matrix mat[n][n], find k such that all elements in kth row are 0 and all elements in kth column are 1. The value of mat[k][k] can be anything (either 0 or 1). If no such k exists, return -1.

Examples:

```
Input: bool mat[n][n] = { {1, 0, 0, 0},  
                         {1, 1, 1, 0},  
                         {1, 1, 0, 0},  
                         {1, 1, 1, 0},  
                         };
```

Output: 0

All elements in 0'th row are 0 and all elements in 0'th column are 1. mat[0][0] is 1 (can be any value)

```
Input: bool mat[n][n] = {{0, 1, 1, 0, 1},  
                         {0, 0, 0, 0, 0},  
                         {1, 1, 1, 0, 0},  
                         {1, 1, 1, 1, 0},  
                         {1, 1, 1, 1, 1}};
```

Output: 1

All elements in 1'st row are 0 and all elements in 1'st column are 1. mat[1][1] is 0 (can be any value)

```
Input: bool mat[n][n] = {{0, 1, 1, 0, 1},  
                         {0, 0, 0, 0, 0},  
                         {1, 1, 1, 0, 0},  
                         {1, 0, 1, 1, 0},  
                         {1, 1, 1, 1, 1}};
```

Output: -1

There is no k such that k'th row elements are 0 and k'th column elements are 1.

Expected time complexity is O(n)

A **Simple Solution** is check all rows one by one. If we find a row i such that all elements of this row are 0 except mat[i][i] which may be either 0 or 1, then we check all values in column i. If all values are 1 in the column, then we return i. Time complexity of this solution is O(n²).

An **Efficient Solution** can solve this problem in O(n) time. The solution is based on below facts.

1) There can be at most one k that can be qualified to be an answer (Why? Note that if kth row has all 0s probably except mat[k][k], then no column can have all 1?).

2) If we traverse the given matrix from a corner (preferably from top right and bottom left), we can quickly discard complete row or complete column based on below rules.

- .a) If mat[i][j] is 0 and i != j, then column j cannot be the solution.
- .b) If mat[i][j] is 1 and i != j, then row i cannot be the solution.

Below is complete algorithm based on above observations.

1) Start from top right corner, i.e., i = 0, j = n-1.
Initialize result as -1.

2) Do following until we find the result or reach outside the matrix.

.....a) If mat[i][j] is 0, then check all elements on left of j in current row.
.....If all elements on left of j are also 0, then set result as i. Note
.....that i may not be result, but if there is a result, then it must be i
.....(Why? we reach mat[i][j] after discarding all rows above it and all
.....columns on right of it)

.....If all left side elements of i'th row are not 0, then this row cannot
.....be a solution, increment i.

.....b) If mat[i][j] is 1, then check all elements below i in current column.
.....If all elements below i are 1, then set result as j. Note that j may
.....not be result, but if there is a result, then it must be j

.....If all elements of j'th column are not 1, then this column cannot be a
.....solution decrement j.

3) If result is -1, return it.

4) Else check validity of result by checking all row and column

```
elements of result
```

Below is C++ implementation based on above idea.

C++

```
// C++ program to find i such that all entries in i'th row are 0
// and all entries in i't column are 1
#include <iostream>
using namespace std;
#define n 5

int find(bool arr[n][n])
{
    // Start from top-most rightmost corner
    // (We could start from other corners also)
    int i=0, j=n-1;

    // Initialize result
    int res = -1;

    // Find the index (This loop runs at most 2n times, we either
    // increment row number or decrement column number)
    while (i<n && j>=0)
    {
        // If current element is 0, then this row may be a solution
        if (arr[i][j] == 0)
        {
            // Check for all elements in this row
            while (j >= 0 && (arr[i][j] == 0 || i == j))
                j--;

            // If all values are 0, then store this row as result
            if (j == -1)
            {
                res = i;
                break;
            }
        }

        // We reach here if we found a 1 in current row, so this
        // row cannot be a solution, increment row number
        else i++;
    }

    else // If current element is 1
    {
        // Check for all elements in this column
        while (i<n && (arr[i][j] == 1 || i == j))
            i++;

        // If all elements are 1
        if (i == n)
        {
            res = j;
            break;
        }

        // We reach here if we found a 0 in current column, so this
        // column cannot be a solution, increment column number
        else j--;
    }
}

// If we could not find result in above loop, then result doesn't exist
if (res == -1)
    return res;

// Check if above computed res is valid
for (int i=0; i<n; i++)
    if (res != i && arr[i][res] != 1)
        return -1;
for (int j=0; j<n; j++)
    if (res != j && arr[res][j] != 0)
        return -1;

return res;
}

/* Driver program to test above functions */
int main()
```

```

{
    bool mat[n][n] = {{0, 0, 1, 1, 0},
                      {0, 0, 0, 1, 0},
                      {1, 1, 1, 1, 0},
                      {0, 0, 0, 0, 0},
                      {1, 1, 1, 1, 1}};
    cout << find(mat);
    return 0;
}

```

Python

```

''' Python program to find k such that all elements in k'th row
are 0 and k'th column are 1'''

def find(arr):

    # store length of the array
    n = len(arr)

    # start from top right-most corner
    i = 0
    j = n - 1

    # initialise result
    res = -1

    # find the index (This loop runs at most 2n times, we
    # either increment row number or decrement column number)
    while i < n and j >= 0:

        # if the current element is 0, then this row may be a solution
        if arr[i][j] == 0:

            # check for all the elements in this row
            while j >= 0 and (arr[i][j] == 0 or i == j):
                j -= 1

            # if all values are 0, update result as row number
            if j == -1:
                res = i
                break

            # if found a 1 in current row, the row can't be a
            # solution, increment row number
            else: i += 1

        # if the current element is 1
        else:

            #check for all the elements in this column
            while i < n and (arr[i][j] == 1 or i == j):
                i +=1

            # if all elements are 1, update result as col number
            if i == n:
                res = j
                break

            # if found a 0 in current column, the column can't be a
            # solution, decrement column number
            else: j -= 1

    # if we couldn't find result in above loop, result doesn't exist
    if res == -1:
        return res

    # check if the above computed res value is valid
    for i in range(0, n):
        if res != i and arr[i][res] != 1:
            return -1
    for j in range(0, j):
        if res != j and arr[res][j] != 0:
            return -1;

    return res;

# test find(arr) function
arr = [ [0,0,1,1,0],
        [1,1,1,1,1],
        [0,0,0,0,0],
        [0,0,1,1,0],
        [1,1,1,1,1]];

```

```
[0,0,0,1,0],  
[1,1,1,1,0],  
[0,0,0,0,0],  
[1,1,1,1,1] ]
```

```
print find(arr)
```

3

Time complexity of this solution is $O(n)$. Note that we traverse at most $2n$ elements in the main while loop.

Find the largest rectangle of 1s with swapping of columns allowed

Given a matrix with 0 and 1s, find the largest rectangle of all 1s in the matrix. The rectangle can be formed by swapping any pair of columns of given matrix.

Example:

```
Input: bool mat[][] = { {0, 1, 0, 1, 0},
                      {0, 1, 0, 1, 1},
                      {1, 1, 0, 1, 0}
                    };
Output: 6
The largest rectangle's area is 6. The rectangle
can be formed by swapping column 2 with 3
The matrix after swapping will be
```

```
0 0 1 1 0
0 0 1 1 1
1 0 1 1 0
```

```
Input: bool mat[R][C] = { {0, 1, 0, 1, 0},
                        {0, 1, 1, 1, 1},
                        {1, 1, 1, 0, 1},
                        {1, 1, 1, 1, 1}
                      };
Output: 9
```

Below are detailed steps for first example mentioned above.

Step 1: First of all, calculate no. of consecutive 1s in every column. An auxiliary array hist[][] is used to store the counts of consecutive 1s. So for the above first example, contents of hist[R][C] would be

```
0 1 0 1 0
0 2 0 2 1
1 3 0 3 0
```

Time complexity of this step is O(R*C)

Step 2: Sort the rows in non-increasing fashion. After sorting step the matrix hist[][] would be

```
1 1 0 0 0
2 2 1 0 0
3 3 1 0 0
```

This step can be done in O(R * (R + C)). Since we know that the values are in range from 0 to R, we can use counting sort for every row.

Step 3: Traverse each row of hist[][] and check for the max area. Since every row is sorted by count of 1s, current area can be calculated by multiplying column number with value in hist[i][j]. This step also takes O(R * C) time.

Below is C++ implementation based of above idea.

```
// C++ program to find the largest rectangle of 1's with swapping
// of columns allowed.
#include<bits/stdc++.h>
#define R 3
#define C 5
using namespace std;

// Returns area of the largest rectangle of 1's
int maxArea(bool mat[R][C])
{
    // An auxiliary array to store count of consecutive 1's
    // in every column.
    int hist[R+1][C+1];

    // Step 1: Fill the auxiliary array hist[][]
    for (int i=0; i<C; i++)
    {
        // First row in hist[][] is copy of first row in mat[][]
        hist[0][i] = mat[0][i];

        // Fill remaining rows of hist[][]
        for (int j=1; j<R; j++)
            hist[j][i] = (mat[j][i]==0)? 0: hist[j-1][i]+1;
    }

    // Step 2: Sort the rows in non-increasing order
    for (int i=0; i<R; i++)
    {
        for (int j=i+1; j<R; j++)
        {
            if (hist[i][j] < hist[j][j])
            {
                int temp = hist[i][j];
                hist[i][j] = hist[j][j];
                hist[j][j] = temp;
            }
        }
    }

    // Step 3: Calculate maximum area
    int maxArea = 0;
    for (int i=0; i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            if (hist[i][j] > 0)
            {
                int area = hist[i][j] * (j - i + 1);
                if (area > maxArea)
                    maxArea = area;
            }
        }
    }

    return maxArea;
}
```

```

// Step 2: Sort rows of hist[][] in non-increasing order
for (int i=0; i<R; i++)
{
    int count[R+1] = {0};

    // counting occurrence
    for (int j=0; j<C; j++)
        count[hist[i][j]]++;

    // Traverse the count array from right side
    int col_no = 0;
    for (int j=R; j>=0; j--)
    {
        if (count[j] > 0)
        {
            for (int k=0; k<count[j]; k++)
            {
                hist[i][col_no] = j;
                col_no++;
            }
        }
    }
}

// Step 3: Traverse the sorted hist[][] to find maximum area
int curr_area, max_area = 0;
for (int i=0; i<R; i++)
{
    for (int j=0; j<C; j++)
    {
        // Since values are in decreasing order,
        // The area ending with cell (i, j) can
        // be obtained by multiplying column number
        // with value of hist[i][j]
        curr_area = (j+1)*hist[i][j];
        if (curr_area > max_area)
            max_area = curr_area;
    }
}
return max_area;
}

// Driver program
int main()
{
    bool mat[R][C] = { {0, 1, 0, 1, 0},
                      {0, 1, 0, 1, 1},
                      {1, 1, 0, 1, 0}
                    };
    cout << "Area of the largest rectangle is " << maxArea(mat);
    return 0;
}

```

Output:

Area of the largest rectangle is 6

Time complexity of above solution is $O(R * (R + C))$ where R is number of rows and C is number of columns in input matrix.

Extra space: $O(R * C)$

Validity of a given Tic-Tac-Toe board configuration

A Tic-Tac-Toe board is given after some moves are played. Find out if the given board is valid, i.e., is it possible to reach this board position after some moves or not.

Note that every arbitrary filled grid of 9 spaces isn't valid e.g. a grid filled with 3 X and 6 O isn't valid situation because each player needs to take alternate turns.

| | | |
|---|---|---|
| X | X | O |
| O | O | X |
| X | O | X |

Valid Board

| | | |
|---|---|---|
| O | X | X |
| O | X | X |
| O | O | X |

Invalid Board

(Both X and O cannot win)

Input is given as a 1D array of size 9.

Input: board[] = {'X', 'X', 'O',
'O', 'O', 'X',
'X', 'O', 'X'};

Output: Valid

Input: board[] = {'O', 'X', 'X',
'O', 'X', 'X',
'O', 'O', 'X'};

Output: Invalid

(Both X and O cannot win)

Input: board[] = {'O', 'X', ' ',
' ', ' ', ' ',
' ', ' ', ' '};

Output: Valid

(Valid board with only two moves played)

Basically, to find the validity of an input grid, we can think of the conditions when an input grid is invalid. Let no. of Xs be countX and no. of Os be countO. Since we know that the game starts with X, a given grid of Tic-Tac-Toe game would be definitely invalid if following two conditions meet

a) countX != countO AND

b) countX != countO + 1

Since X is always the first move, second condition is also required.

Now does it mean that all the remaining board positions are valid one? The answer is NO. Think of the cases when input grid is such that both X and O are making straight lines. This is also not valid position because the game ends when one player wins. So we need to check the following condition as well

c) If input grid shows that both the players are in winning situation, its an invalid position.

d) If input grid shows that the player with O has put a straight-line (i.e. is in win condition) and countX != countO, its an invalid position. The reason is that O plays his move only after X plays his move. Since X has started the game, O would win when both X and O has played equal no. of moves.

e) If input grid shows that X is in winning condition than xCount must be one greater than oCount.

Armed with above conditions i.e. a), b), c) and d), we can now easily formulate an algorithm/program to check the validity of a given Tic-Tac-Toe board position.

- 1) countX == countO or countX == countO + 1
- 2) If O is in win condition then check
 - a) If X also wins, not valid
 - b) If xbox != obox , not valid
- 3) If X is in win condition then check if xCount is one more than oCount or not

Another way to find the validity of a given board is using inverse method i.e. rule out all the possibilities when a given board is invalid.

```

// C++ program to check whether a given tic tac toe
// board is valid or not
#include <iostream>
using namespace std;

// This matrix is used to find indexes to check all
// possible winning triplets in board[0..8]
int win[8][3] = {{0, 1, 2}, // Check first row.
                 {3, 4, 5}, // Check second Row
                 {6, 7, 8}, // Check third Row
                 {0, 3, 6}, // Check first column
                 {1, 4, 7}, // Check second Column
                 {2, 5, 8}, // Check third Column
                 {0, 4, 8}, // Check first Diagonal
                 {2, 4, 6}}; // Check second Diagonal

// Returns true if character 'c' wins. c can be either
// 'X' or 'O'
bool isCWin(char *board, char c)
{
    // Check all possible winning combinations
    for (int i=0; i<8; i++)
        if (board[win[i][0]] == c &&
            board[win[i][1]] == c &&
            board[win[i][2]] == c )
            return true;
    return false;
}

// Returns true if given board is valid, else returns false
bool isValid(char board[9])
{
    // Count number of 'X' and 'O' in the given board
    int xCount=0, oCount=0;
    for (int i=0; i<9; i++)
    {
        if (board[i]=='X') xCount++;
        if (board[i]=='O') oCount++;
    }

    // Board can be valid only if either xCount and oCount
    // is same or xCount is one more than oCount
    if (xCount==oCount || xCount==oCount+1)
    {
        // Check if 'O' is winner
        if (isCWin(board, 'O'))
        {
            // Check if 'X' is also winner, then
            // return false
            if (isCWin(board, 'X'))
                return false;

            // Else return true xCount and oCount are same
            return (xCount == oCount);
        }

        // If 'X' wins, then count of X must be greater
        if (isCWin(board, 'X') && xCount != oCount + 1)
            return false;
    }

    // If 'O' is not winner, then return true
    return true;
}
return false;
}

// Driver program
int main()
{
    char board[] = {'X', 'X', 'O',
                   'O', 'O', 'X',
                   'X', 'O', 'X'};
    (isValid(board)) ? cout << "Given board is valid":
                           cout << "Given board is not valid";
    return 0;
}

```

Output:

Given board is valid

Thanks to [Utkarsh](#) for suggesting this solution [here](#).

Minimum Initial Points to Reach Destination

Given a grid with each cell consisting of positive, negative or no points i.e, zero points. We can move across a cell only if we have positive points (> 0). Whenever we pass through a cell, points in that cell are added to our overall points. We need to find minimum initial points to reach cell $(m-1, n-1)$ from $(0, 0)$.

Constraints :

Find length of the longest consecutive path from a given starting character

Given a matrix of characters. Find length of the longest path from a given character, such that all characters in the path are consecutive to each other, i.e., every character in path is next to previous in alphabetical order. It is allowed to move in all 8 directions from a cell.

| | | |
|---|---|---|
| a | c | d |
| h | b | e |
| i | g | f |

Starting Point 'e'

Example

```
Input: mat[][] = { {a, c, d},
                  {h, b, e},
                  {i, g, f}}
Starting Point = 'e'

Output: 5
If starting point is 'e', then longest path with consecutive
characters is "e f g h i".

Input: mat[R][C] = { {b, e, f},
                  {h, d, a},
                  {i, c, a}};
Starting Point = 'b'

Output: 1
'c' is not present in all adjacent cells of 'b'
```

The idea is to first search given starting character in the given matrix. Do Depth First Search (DFS) from all occurrences to find all consecutive paths. While doing DFS, we may encounter many subproblems again and again. So we use dynamic programming to store results of subproblems.

Below is C++ implementation of above idea.

```
// C++ program to find the longest consecutive path
#include<bits/stdc++.h>
#define R 3
#define C 3
using namespace std;

// tool matrices to recur for adjacent cells.
int x[] = {0, 1, 1, -1, 1, 0, -1, -1};
int y[] = {1, 0, 1, 1, -1, -1, 0, -1};

// dp[i][j] Stores length of longest consecutive path
// starting at arr[i][j].
int dp[R][C];

// check whether mat[i][j] is a valid cell or not.
bool isvalid(int i, int j)
{
    if (i < 0 || j < 0 || i >= R || j >= C)
        return false;
    return true;
}

// Check whether current character is adjacent to previous
// character (character processed in parent call) or not.
bool isadjacent(char prev, char curr)
{
    return ((curr - prev) == 1);
}
```

```

// i, j are the indices of the current cell and prev is the
// character processed in the parent call.. also mat[i][j]
// is our current character.
int getLenUtil(char mat[R][C], int i, int j, char prev)
{
    // If this cell is not valid or current character is not
    // adjacent to previous one (e.g. d is not adjacent to b )
    // or if this cell is already included in the path than return 0.
    if (!isValid(i, j) || !isAdjacent(prev, mat[i][j]))
        return 0;

    // If this subproblem is already solved , return the answer
    if (dp[i][j] != -1)
        return dp[i][j];

    int ans = 0; // Initialize answer

    // recur for paths with differnt adjacent cells and store
    // the length of longest path.
    for (int k=0; k<8; k++)
        ans = max(ans, 1 + getLenUtil(mat, i + x[k],
                                       j + y[k], mat[i][j]));

    // save the answer and return
    return dp[i][j] = ans;
}

// Returns length of the longest path with all characters consecutive
// to each other. This function first initializes dp array that
// is used to store results of subproblems, then it calls
// recursive DFS based function getLenUtil() to find max length path
int getLen(char mat[R][C], char s)
{
    memset(dp, -1, sizeof dp);
    int ans = 0;

    for (int i=0; i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            // check for each possible starting point
            if (mat[i][j] == s) {

                // recur for all eight adjacent cells
                for (int k=0; k<8; k++)
                    ans = max(ans, 1 + getLenUtil(mat,
                                                   i + x[k], j + y[k], s));
            }
        }
    }
    return ans;
}

// Driver program
int main() {

    char mat[R][C] = { {'a','c','d'},
                      { 'h','b','a'},
                      { 'i','g','f'}};

    cout << getLen(mat, 'a') << endl;
    cout << getLen(mat, 'e') << endl;
    cout << getLen(mat, 'b') << endl;
    cout << getLen(mat, 'f') << endl;
    return 0;
}

```

Output:

```

4
0
3
4

```

Thanks to [Gaurav Ahirwar](#) for above solution.

Collect maximum points in a grid using two traversals

Given a matrix where every cell represents points. How to collect maximum points using two traversals under following conditions?

Let the dimensions of given grid be R x C.

1) The first traversal starts from top left corner, i.e., (0, 0) and should reach left bottom corner, i.e., (R-1, 0). The second traversal starts from top right corner, i.e., (0, C-1) and should reach bottom right corner, i.e., (R-1, C-1)/

2) From a point (i, j), we can move to (i+1, j+1) or (i+1, j-1) or (i+1, j)

3) A traversal gets all points of a particular cell through which it passes. If one traversal has already collected points of a cell, then the other traversal gets no points if goes through that cell again.

Input :

```
int arr[R][C] = {{3, 6, 8, 2},  
                  {5, 2, 4, 3},  
                  {1, 1, 20, 10},  
                  {1, 1, 20, 10},  
                  {1, 1, 20, 10},  
};
```

Output: 73

Explanation :

| | | | |
|----------|----------|-----------|-----------|
| 3 | 6 | 8 | 2 |
| 5 | 2 | 4 | 3 |
| 1 | 1 | 20 | 10 |
| 1 | 1 | 20 | 10 |
| 1 | 1 | 20 | 10 |

First traversal collects total points of value $3 + 2 + 20 + 1 + 1 = 27$

Second traversal collects total points of value $2 + 4 + 10 + 20 + 10 = 46$.
Total Points collected = $27 + 46 = 73$.

Source: <http://qa.geeksforgeeks.org/1485/running-through-the-grid-to-get-maximum-nutritional-value>

Both traversals always move forward along x

Base Cases:

```
// If destinations reached  
if (x == R-1 && y1 == 0 && y2 == C-1)  
maxPoints(arr, x, y1, y2) = arr[x][y1] + arr[x][y2];  
  
// If any of the two locations is invalid (going out of grid)  
if input is not valid  
maxPoints(arr, x, y1, y2) = -INF (minus infinite)  
  
// If both traversals are at same cell, then we count the value of cell  
// only once.  
If y1 and y2 are same  
    result = arr[x][y1]  
Else  
    result = arr[x][y1] + arr[x][y2]  
  
result += max { // Max of 9 cases  
    maxPoints(arr, x+1, y1+1, y2),  
    maxPoints(arr, x+1, y1+1, y2+1),  
    maxPoints(arr, x+1, y1+1, y2-1),  
    maxPoints(arr, x+1, y1-1, y2),  
    maxPoints(arr, x+1, y1-1, y2+1),  
    maxPoints(arr, x+1, y1-1, y2-1),  
    maxPoints(arr, x+1, y1, y2),  
    maxPoints(arr, x+1, y1, y2+1),  
    maxPoints(arr, x+1, y1, y2-1)  
}
```

The above recursive solution has many subproblems that are solved again and again. Therefore, we can use Dynamic Programming to solve the above problem more efficiently. Below is [memoization](#) (Memoization is alternative to table based iterative solution in Dynamic Programming) based implementation. In below implementation, we use a memoization table mem to keep track of already solved problems.

```
// A Memoization based program to find maximum collection  
// using two traversals of a grid
```

```

#include<bits/stdc++.h>
using namespace std;
#define R 5
#define C 4

// checks whether a given input is valid or not
bool isValid(int x, int y1, int y2)
{
    return (x >= 0 && x < R && y1 >=0 &&
            y1 < C && y2 >=0 && y2 < C);
}

// Driver function to collect max value
int getMaxUtil(int arr[R][C], int mem[R][C][C], int x, int y1, int y2)
{
    /*----- BASE CASES -----*/
    // if P1 or P2 is at an invalid cell
    if (!isValid(x, y1, y2)) return INT_MIN;

    // if both traversals reach their destinations
    if (x == R-1 && y1 == 0 && y2 == C-1)
        return arr[x][y1] + arr[x][y2];

    // If both traversals are at last row but not at their destination
    if (x == R-1) return INT_MIN;

    // If subproblem is already solved
    if (mem[x][y1][y2] != -1) return mem[x][y1][y2];

    // Initialize answer for this subproblem
    int ans = INT_MIN;

    // this variable is used to store gain of current cell(s)
    int temp = (y1 == y2)? arr[x][y1]: arr[x][y1] + arr[x][y2];

    /* Recur for all possible cases, then store and return the
       one with max value */
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2-1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2+1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2));

    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2-1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2+1));

    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2-1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2+1));

    return (mem[x][y1][y2] = ans);
}

// This is mainly a wrapper over recursive function getMaxUtil().
// This function creates a table for memoization and calls
// getMaxUtil()
int geMaxCollection(int arr[R][C])
{
    // Create a memoization table and initialize all entries as -1
    int mem[R][C][C];
    memset(mem, -1, sizeof(mem));

    // Calculation maximum value using memoization based function
    // getMaxUtil()
    return getMaxUtil(arr, mem, 0, 0, C-1);
}

// Driver program to test above functions
int main()
{
    int arr[R][C] = {{3, 6, 8, 2},
                     {5, 2, 4, 3},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10}};
    cout << "Maximum collection is " << geMaxCollection(arr);
    return 0;
}

```

Output:

Maximum collection is 73

Thanks to Gaurav Ahirwar for suggesting above problem and solution [here](#).

Rotate Matrix Elements

Given a matrix, clockwise rotate elements in it.

Examples:

Input
1 2 3
4 5 6
7 8 9

Output:
4 1 2
7 5 3
8 9 6

For 4*4 matrix

Input:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Output:
5 1 2 3
9 10 6 4
13 11 7 8
14 15 16 12

The idea is to use loops similar to the [program for printing a matrix in spiral form](#). One by one rotate all rings of elements, starting from the outermost. To rotate a ring, we need to do following.

- 1) Move elements of top row.
- 2) Move elements of last column.
- 3) Move elements of bottom row.
- 4) Move elements of first column.

Repeat above steps for inner ring while there is an inner ring.

Below is the implementation of above idea. Thanks to Gaurav Ahirwar for suggesting below solution [here](#).

C/C++

```
// C++ program to rotate a matrix

#include <bits/stdc++.h>
#define R 4
#define C 4
using namespace std;

// A function to rotate a matrix mat[][] of size R x C.
// Initially, m = R and n = C
void rotatematrix(int m, int n, int mat[R][C])
{
    int row = 0, col = 0;
    int prev, curr;

    /*
        row - Starting row index
        m - ending row index
        col - starting column index
        n - ending column index
        i - iterator
    */
    while (row < m && col < n)
    {

        if (row + 1 == m || col + 1 == n)
            break;

        // Store the first element of next row, this
        // element will replace first element of current
        // row
        prev = mat[row + 1][col];

        /* Move elements of first row from the remaining rows */
        for (int i = col; i < n; i++)
        {
            curr = mat[row][i];
```

```

        mat[row][i] = prev;
        prev = curr;
    }
    row++;

    /* Move elements of last column from the remaining columns */
    for (int i = row; i < m; i++)
    {
        curr = mat[i][n-1];
        mat[i][n-1] = prev;
        prev = curr;
    }
    n--;

    /* Move elements of last row from the remaining rows */
    if (row < m)
    {
        for (int i = n-1; i >= col; i--)
        {
            curr = mat[m-1][i];
            mat[m-1][i] = prev;
            prev = curr;
        }
    }
    m--;

    /* Move elements of first column from the remaining rows */
    if (col < n)
    {
        for (int i = m-1; i >= row; i--)
        {
            curr = mat[i][col];
            mat[i][col] = prev;
            prev = curr;
        }
    }
    col++;
}

// Print rotated matrix
for (int i=0; i<R; i++)
{
    for (int j=0; j<C; j++)
        cout << mat[i][j] << " ";
    cout << endl;
}
}

/* Driver program to test above functions */
int main()
{
    // Test Case 1
    int a[R][C] = { {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12},
                    {13, 14, 15, 16} };

    // Test Case 2
    /* int a[R][C] = {{1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9}
                     }; */
    /* rotatematrix(R, C, a);
    return 0;
}

```

Python

```

# Python program to rotate a matrix

# Function to rotate a matrix
def rotateMatrix(mat):

    if not len(mat):
        return

    """
        top : starting row index
        bottom : ending row index
        left : starting column index
    """

```

```

    right : ending column index
"""

top = 0
bottom = len(mat)-1

left = 0
right = len(mat[0])-1

while left < right and top < bottom:

    # Store the first element of next row,
    # this element will replace first element of
    # current row
    prev = mat[top+1][left]

    # Move elements of top row one step right
    for i in range(left, right+1):
        curr = mat[top][i]
        mat[top][i] = prev
        prev = curr

    top += 1

    # Move elements of rightmost column one step downwards
    for i in range(top, bottom+1):
        curr = mat[i][right]
        mat[i][right] = prev
        prev = curr

    right -= 1

    # Move elements of bottom row one step left
    for i in range(right, left-1, -1):
        curr = mat[bottom][i]
        mat[bottom][i] = prev
        prev = curr

    bottom -= 1

    # Move elements of leftmost column one step upwards
    for i in range(bottom, top-1, -1):
        curr = mat[i][left]
        mat[i][left] = prev
        prev = curr

    left += 1

return mat

# Utility Function
def printMatrix(mat):
    for row in mat:
        print row

# Test case 1
matrix =[
    [1, 2, 3, 4 ],
    [5, 6, 7, 8 ],
    [9, 10, 11, 12 ],
    [13, 14, 15, 16 ]
]

# Test case 2
"""

matrix =[

    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

"""

matrix = rotateMatrix(matrix)
# Print modified matrix
printMatrix(matrix)

```

5 1 2 3
9 10 6 4
13 11 7 8

14 15 16 12

Find sum of all elements in a matrix except the elements in row and/or column of given cell?

Given a 2D matrix and a set of cell indexes e.g., an array of (i, j) where i indicates row and j column. For every given cell index (i, j), find sums of all matrix elements except the elements present in ith row and/or jth column.

Example:

```
mat[][] = { {1, 1, 2}
            {3, 4, 6}
            {5, 3, 2} }
Array of Cell Indexes: {(0, 0), (1, 1), (0, 1)}
Output: 15, 10, 16
```

Source: <http://qa.geeksforgeeks.org/622/select-column-matrix-then-find-remaining-elements-matrices?show=625#a625>

A **Naive Solution** is to one by once consider all given cell indexes. For every cell index (i, j), find the sum of matrix elements that are not present either at ith row or at jth column. Below is C++ implementation of the Naive approach.

```
#include<bits/stdc++.h>
#define R 3
#define C 3
using namespace std;

// A structure to represent a cell index
struct Cell
{
    int r; // r is row, varies from 0 to R-1
    int c; // c is column, varies from 0 to C-1
};

// A simple solution to find sums for a given array of cell indexes
void printSums(int mat[][C], struct Cell arr[], int n)
{
    // Iterate through all cell indexes
    for (int i=0; i<n; i++)
    {
        int sum = 0, r = arr[i].r, c = arr[i].c;

        // Compute sum for current cell index
        for (int j=0; j<R; j++)
            for (int k=0; k<C; k++)
                if (j != r && k != c)
                    sum += mat[j][k];
        cout << sum << endl;
    }
}

// Driver program to test above
int main()
{
    int mat[][C] = {{1, 1, 2}, {3, 4, 6}, {5, 3, 2}};
    struct Cell arr[] = {{0, 0}, {1, 1}, {0, 1}};
    int n = sizeof(arr)/sizeof(arr[0]);
    printSums(mat, arr, n);
    return 0;
}
```

Output:

```
15
10
16
```

Time complexity of the above solution is $O(n * R * C)$ where n is number of given cell indexes and $R \times C$ is matrix size.

An **Efficient Solution** can compute all sums in $O(R \times C + n)$ time. The idea is to precompute total sum, row and column sums before processing the given array of indexes. Below are details

1. Calculate sum of matrix, call it sum
2. Calculate sum of individual rows and columns. (row[] and col[])
3. For a cell index (i, j), the desired sum will be sum - row[i] - col[j] + arr[i][j]

Below is C++ implementation of above idea.

```
// An efficient C++ program to compute sum for given array of cell indexes
#include<bits/stdc++.h>
#define R 3
```

```

#define C 3
using namespace std;

// A structure to represent a cell index
struct Cell
{
    int r; // r is row, varies from 0 to R-1
    int c; // c is column, varies from 0 to C-1
};

void printSums(int mat[][][C], struct Cell arr[], int n)
{
    int sum = 0;
    int row[R] = {};
    int col[C] = {};

    // Compute sum of all elements, sum of every row and sum every column
    for (int i=0; i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            sum += mat[i][j];
            col[j] += mat[i][j];
            row[i] += mat[i][j];
        }
    }

    // Compute the desired sum for all given cell indexes
    for (int i=0; i<n; i++)
    {
        int ro = arr[i].r, co = arr[i].c;
        cout << sum - row[ro] - col[co] + mat[ro][co] << endl;
    }
}

// Driver program to test above function
int main()
{
    int mat[][][C] = {{1, 1, 2}, {3, 4, 6}, {5, 3, 2}};
    struct Cell arr[] = {{0, 0}, {1, 1}, {0, 1}};
    int n = sizeof(arr)/sizeof(arr[0]);
    printSums(mat, arr, n);
    return 0;
}

```

Output:

```

15
10
16

```

Time Complexity: $O(R \times C + n)$

Auxiliary Space: $O(R + C)$

Thanks to [Gaurav Ahirwar](#) for suggesting this efficient solution [here](#).

Find a common element in all rows of a given row-wise sorted matrix

Given a matrix where every row is sorted in increasing order. Write a function that finds and returns a common element in all rows. If there is no common element, then returns -1.

Example:

```
Input: mat[4][5] = { {1, 2, 3, 4, 5},
                     {2, 4, 5, 8, 10},
                     {3, 5, 7, 9, 11},
                     {1, 3, 5, 7, 9},
                   };
Output: 5
```

A **O(m*n*n)** simple solution is to take every element of first row and search it in all other rows, till we find a common element. Time complexity of this solution is $O(m*n*n)$ where m is number of rows and n is number of columns in given matrix. This can be improved to $O(m*n*\log n)$ if we use [Binary Search](#) instead of linear search.

We can solve this problem in **O(mn)** time using the approach similar to merge of [Merge Sort](#). The idea is to start from the last column of every row. If elements at all last columns are same, then we found the common element. Otherwise we find the minimum of all last columns. Once we find a minimum element, we know that all other elements in last columns cannot be a common element, so we reduce last column index for all rows except for the row which has minimum value. We keep repeating these steps till either all elements at current last column dont become same, or a last column index reaches 0.

Below is C implementation of above idea.

```
// A C program to find a common element in all rows of a
// row wise sorted array
#include<stdio.h>

// Specify number of rows and columns
#define M 4
#define N 5

// Returns common element in all rows of mat[M][N]. If there is no
// common element, then -1 is returned
int findCommon(int mat[M][N])
{
    // An array to store indexes of current last column
    int column[M];
    int min_row; // To store index of row whose current
                 // last element is minimum

    // Initialize current last element of all rows
    int i;
    for (i=0; i<M; i++)
        column[i] = N-1;

    min_row = 0; // Initialize min_row as first row

    // Keep finding min_row in current last column, till either
    // all elements of last column become same or we hit first column.
    while (column[min_row] >= 0)
    {
        // Find minimum in current last column
        for (i=0; i<M; i++)
        {
            if (mat[i][column[i]] < mat[min_row][column[min_row]] )
                min_row = i;
        }

        // eq_count is count of elements equal to minimum in current last
        // column.
        int eq_count = 0;

        // Travers current last column elements again to update it
        for (i=0; i<M; i++)
        {
            // Decrease last column index of a row whose value is more
            // than minimum.
            if (mat[i][column[i]] > mat[min_row][column[min_row]])
            {
                if (column[i] == 0)
                    return -1;

                column[i] -= 1; // Reduce last column index by 1
            }
        }
    }
}
```

```

        else
            eq_count++;
    }

    // If equal count becomes M, return the value
    if (eq_count == M)
        return mat[min_row][column[min_row]];
}
return -1;
}

// driver program to test above function
int main()
{
    int mat[M][N] = { {1, 2, 3, 4, 5},
                      {2, 4, 5, 8, 10},
                      {3, 5, 7, 9, 11},
                      {1, 3, 5, 7, 9},
                  };
    int result = findCommon(mat);
    if (result == -1)
        printf("No common element");
    else
        printf("Common element is %d", result);
    return 0;
}

```

Output:

Common element is 5

Explanation for working of above code

Let us understand working of above code for following example.

Initially entries in last column array are N-1, i.e., {4, 4, 4, 4}

{1, 2, 3, 4, **5**},
 {2, 4, 5, 8, **10**},
 {3, 5, 7, 9, **11**},
 {1, 3, 5, 7, **9**},

The value of min_row is 0, so values of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 3, 3, 3}.

{1, 2, 3, 4, **5**},
 {2, 4, 5, **8**, 10},
 {3, 5, 7, **9**, 11},
 {1, 3, 5, 7, 9},

The value of min_row remains 0 and and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 2, 2}.

{1, 2, 3, 4, **5**},
 {2, 4, **5**, 8, 10},
 {3, 5, 7, 9, 11},
 {1, 3, **5**, 7, 9},

The value of min_row remains 0 and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 1, 2}.

{1, 2, 3, 4, **5**},
 {2, 4, **5**, 8, 10},
 {3, **5**, 7, 9, 11},
 {1, 3, **5**, 7, 9},

Now all values in current last columns of all rows is same, so 5 is returned.

A Hashing Based Solution

We can also use hashing. This solution works even if the rows are not sorted. It can be used to print all common elements.

Step1: Create a Hash Table with all key as distinct elements of row1. Value for all these will be 0.

Step2:

For i = 1 to M-1
 For j = 0 to N-1
 If (mat[i][j] is already present in Hash Table)
 If (And this is not a repetition in current row.
 This can be checked by comparing HashTable value with

```
row number)
Update the value of this key in HashTable with current
row number
```

Step3: Iterate over HashTable and print all those keys for
which value = M

Time complexity of the above hashing based solution is $O(MN)$ under the assumption that search and insert in HashTable take $O(1)$ time. Thanks to Nishant for suggesting this solution in a comment below.

Exercise: Given n sorted arrays of size m each, find all common elements in all arrays in $O(mn)$ time.