Contents

Chapter 1

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
Time Complexity: Depends on what sorting algorithm we use.  If we use Merge Sort or Heap Sort then (-)(nlogn)

Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is O(n) for merge sort and
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)    increment l.  Now l = 1

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

A[l] + A[r] ( 4 + 10)     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
```

Implementation:

C

```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)
```

```c
{
    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
# Python program to check for the sum condition to be satisified
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
```

```python
    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 is 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__
```

Output:

```
 Array has two elements with the given sum
```

METHOD 2 (Use Hash Map)
Thanks to Bindu for suggesting this method and thanks to Shekhufor 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++


```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
// 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__
```

Time Complexity: O(n)
Output:

```
 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.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whet

Category: Arrays Tags: Hashing

Post navigation

← Next Power of 2 Majority Element →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 2

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*n).
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, 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(nlogn)

Auxiliary Space: O(n)

METHOD 3 (Using Moore's 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 Moore's 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 Moore's 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).

Source

http://www.geeksforgeeks.org/majority-element/

Chapter 3

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 Solutio**n 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:

```
 5
```

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-the-number-occurring-odd-number-of-times/

Chapter 4

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.

Kadane's 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
Explanation:

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_endin

    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
Program:
```

C++

```cpp
// C++ program to print largest contiguous array sum
#include<iostream>
using namespace std;

int maxSubArraySum(int a[], int size)
{
```

```cpp
    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
# 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_
```

Output:

```
 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++

```cpp
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

```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.

C++

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

int maxSubArraySum(int a[], int size)
{
   int max_so_far = a[0];
   int curr_max = a[0];

   for (int i = 1; i < size; i++)
   {
       curr_max = max(a[i], curr_max+a[i]);
       max_so_far = max(max_so_far, curr_max);
   }
   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 " << max_sum;
   return 0;
}
```

Python

```python
# Python program to find maximum contiguous subarray

def maxSubArraySum(a,size):

    max_so_far =a[0]
    curr_max = a[0]

    for i in range(1,size):
        curr_max = max(a[i], curr_max + a[i])
        max_so_far = max(max_so_far,curr_max)

    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_
```

Output:

```
 Maximum contiguous sum is 7
```

Now try below question

Given an array of integers (possibly some of the elements negative), write a C program to find out the *maximum product* possible by adding 'n' consecutive integers in the array, n

References:

http://en.wikipedia.org/wiki/Kadane%27s_Algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Category: Arrays Tags: Dynamic Programming

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Source

http://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

Chapter 5

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:

```c
#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 elemets in arary */
    int x2 = 1; /* For xor of all the elemets 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.

Source

http://www.geeksforgeeks.org/find-the-missing-number/

Category: Arrays

Chapter 6

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.

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 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(logn) 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 If element is found in selected sub-array then return index
     Else return -1.
```

Implementation:

```c
/* 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(Logn) time in all cases when duplicates are allowed. For example consider searching 0 in {2, 2, 2, 2, 2, 2, 2, 2, 0, 2} and {2, 0, 2, 2, 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.

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-pivoted-array/

Category: Arrays Tags: array, Binary-Search

Post navigation

← Function to check if a singly linked list is palindrome The Great Tree-List Recursion Problem. →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 7

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:


```c
#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 emenet from N[]
    {
      mPlusN[k] = N[j];
      k++;
      j++;
    }
  }
}
```

```c
/* 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.

Source

http://www.geeksforgeeks.org/merge-one-array-of-size-n-into-another-one-of-size-mn/

Category: Arrays Tags: array

Chapter 8

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.
    Median = (max(ar1[0], ar2[0]) + min(ar1[1], 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:

```c
#include<stdio.h>

int max(int, int); /* to get maximum of two integers */
int min(int, int); /* to get minimum of two integeres */
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] 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] >= ar2[j] and ar1[i]

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 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:

```c
#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 ds3etph5wn

Asked by Snehal

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/median-of-two-sorted-arrays/

Category: Arrays Tags: Divide and Conquer

Post navigation

← Database Management Systems | Set 1 C Language | Set 1 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 9

Write a program to reverse an array

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
/* 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);
  getchar();
  return 0;
}
```

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.

```
/* 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);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Please write comments if you find any bug in the above programs or other ways to solve the same problem.

Source

Chapter 10

Program for array rotation

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

Rotation of the above array by 2 will make array

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
To rotate by one, store arr[0] in a temporary variable temp,  move arr[1] to arr[0], arr[2] to arr[1] …and fin
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}


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

/*Fuction 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]);
}

/*Fuction 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

Please write comments if you find any bug in above programs/algorithms.

Source

http://www.geeksforgeeks.org/array-rotation/

Category: Arrays Tags: array

Post navigation

← Program to count leaf nodes in a binary tree Reversal algorithm for array rotation →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 11

Reversal algorithm for array rotation

Write a function rotate(arr[], 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 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[], l, 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
/*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]);
  printf("%\n ");
}

/*Function to reverse arr[] from index start to end*/
void rvereseArray(int arr[], int start, int end)
{
  int i;
  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};
   leftRotate(arr, 2, 7);
   printArray(arr, 7);
   getchar();
   return 0;
}
```

Time Complexity: O(n)

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

Source

Chapter 12

Block swap algorithm for array rotation

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

Rotation of the above array by 2 will make array

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 ABlBr into BrBlA. 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 AlArB into BArAl. 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:

```c
#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

Please write comments if you find any bug in the above programs/algorithms or want to share any additional information about the block swap algorithm.

Source

Category: Arrays Tags: array

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 13

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:

```c
#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.

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

Source

Chapter 14

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++

```cpp
#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__
```

Output:

```
 17 5 2
```

Time Complexity: O(n*n)

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 it's 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__
```

Output

```
 2 5 17
```

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

Chapter 15

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, 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(nlogn)
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(nlogn).
```

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.

```cpp
// 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 If previous element is not equal to current
           so set the count to 1 */
        else 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 Ahirwarfor 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(nlogn) if aSelf 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/sort-elements-by-frequency/

Category: Arrays Tags: array

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 16

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 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 progra to test above functions */
int main(int argv, char** args)
{
  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().

# of inversions in each half
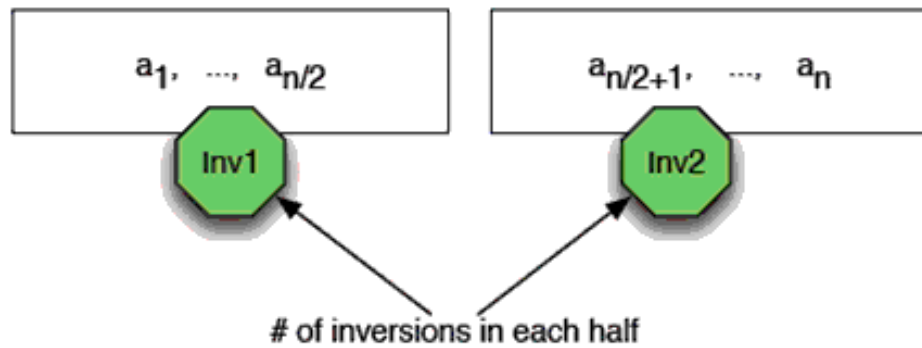
How to get number of inversions in merge()?

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] ... a[mid]) will be greater than a[j]
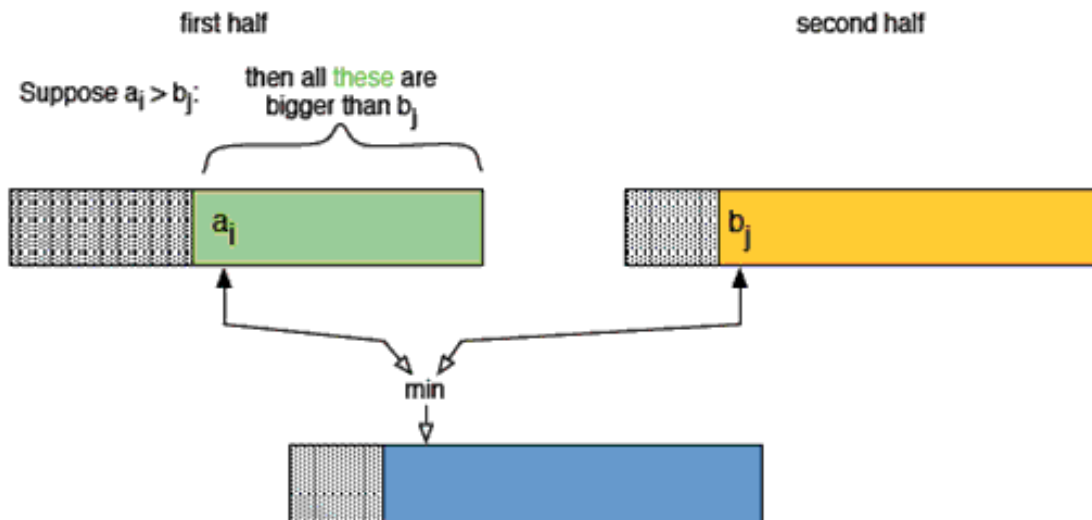
The complete picture:
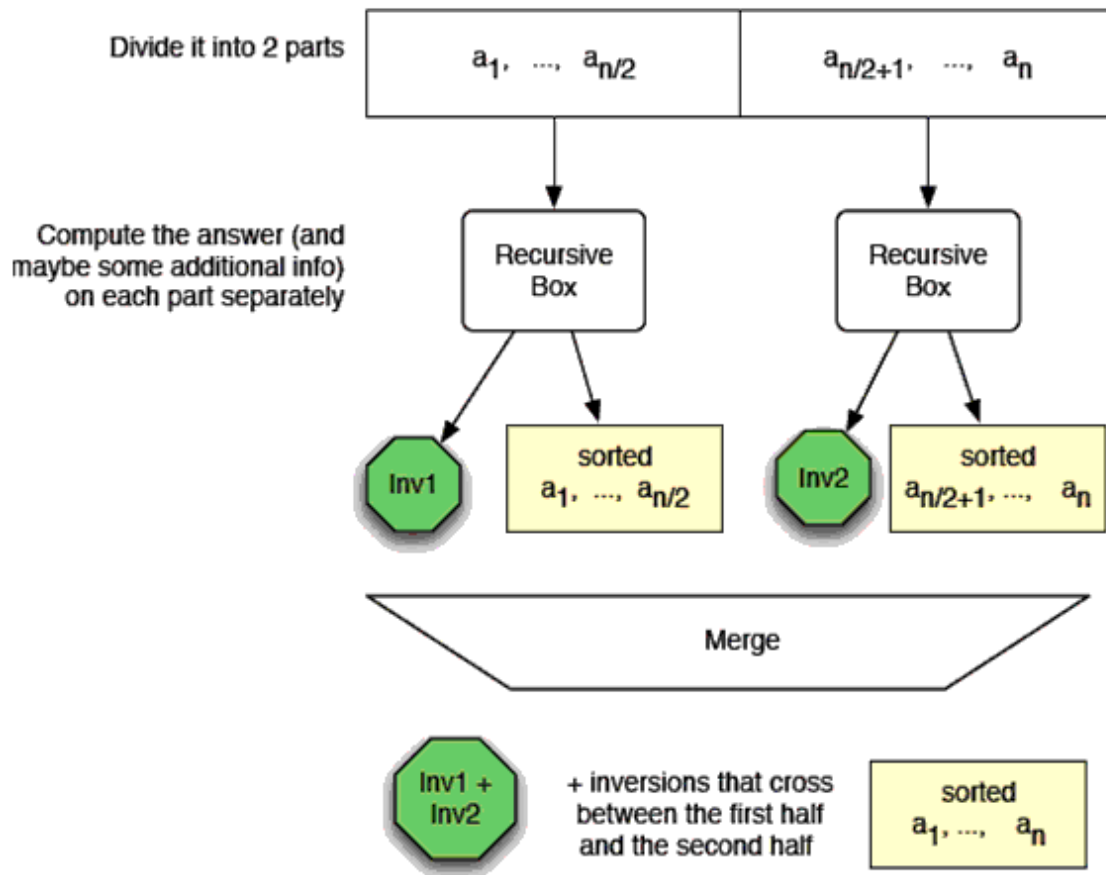
Implementation:

```c
#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;
```

```c
  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 argv, 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

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/counting-inversions/

Category: Arrays Tags: Divide and Conquer

Post navigation

← Sort elements by frequency | Set 1 Operating Systems | Set 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 17

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 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;
```

```c
  // 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(nlogn) + O(n) = O(nlogn)

Asked by Vineet

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/two-elements-whose-sum-is-closest-to-zero/

Category: Arrays Tags: array

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 18

Find the smallest and second smallest element in an array

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

Difficulty Level: Rookie

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:

```
#include <stdio.h>
#include <limits.h> /* For INT_MAX */

/* Function to print first smallest and second smallest elements */
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;
}
```

Output:

```
 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)

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/to-find-smallest-and-second-smallest-element-in-an-array/

Category: Arrays Tags: array

Chapter 19

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[]), array's 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, 3, 10}, x = 3
Output: True (x appears more than n/2 times in the given array)

Input: arr[] = {1, 1, 2, 4, 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
/* 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);
```

```c
  /* 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);

  getchar();
  return 0;
}
```

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 */
```

```c
    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;
}
```

Time Complexity: O(Logn)
Algorithmic Paradigm: Divide and Conquer

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

Source

Category: Arrays Tags: Divide and Conquer

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 20

Maximum and minimum of an array using minimum number of comparisons

Write a C function to return minimum and maximum in an array. You 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;
};
```

```c
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 the two parts to get the maximum and the minimum of the the whole array.

```
Pair MaxMin(array, array_size)
   if array_size = 1
      return element as both max and min
   else if arry_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

```c
/* 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 on 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;

  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

```
  T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2
  T(2) = 1
  T(1) = 0
```

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.

```c
#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 startung 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.

Please write comments if you find any bug in the above programs/algorithms or a better way to solve the same problem.

Source

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 21

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, 0, 1, 1, 1, 1, 1]
```

Method 1 (Count 0s or 1s)
Thanks to Naveenfor 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 // 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)

Please write comments if you find any of the above algorithms/code incorrect, or a better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/segregate-0s-and-1s-in-an-array-by-traversing-array-once/

Category: Arrays

Chapter 22

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 + klogk)

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)
1) Sort the elements in descending order in O(nLogn)
2) Print the first k numbers of the sorted array O(k).

Time complexity: O(nlogn)

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(klogn)

Time complexity: O(n + klogn)

Method 5(Use Oder 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(kLogk). This step is needed only if sorted output is required.

Time complexity: O(n) if we don't need the sorted output, otherwise O(n+kLogk)

Thanks to Shilpifor 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 geek4ufor 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 heapifyfor MH
......b) Else ignore it.
// The step 2 is O((n-k)*logk)

3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: O(k + (n-k)Logk) without sorted output. If sorted output is needed then O(k + (n-k)Logk + kLogk)

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

Please write comments if you find any of the above explanations/algorithms incorrect, or find better ways to solve the same problem.

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

Asked by geek4u

Source

http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/

Chapter 23

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 problems 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)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem

Source

http://www.geeksforgeeks.org/maximum-difference-between-two-elements/

Chapter 24

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++;
    }
  }
```

```c
  /* 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.

```c
#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;
  while (i < m && j < n)
  {
    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(mLogn, nLogm)). 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

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/

Chapter 25

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 than 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

```c
#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 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)

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem, or want to share code for floor implementation.

Source

http://www.geeksforgeeks.org/search-floor-and-ceil-in-a-sorted-array/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 26

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;
```

```c
  /* 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.

```c
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)

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/a-product-array-puzzle/

Category: Arrays

Chapter 27

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 lef
Implementation:

#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 = 7, i = 0;

  segregateEvenOdd(arr, arr_size);

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

  getchar();
  return 0;
}
```

Time Complexity: O(n)

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

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/segregate-even-and-odd-numbers/

Category: Arrays

Chapter 28

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*n)
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.

```c
#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 − 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 − 15 = 6

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

```c
#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 1's 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.

```c
#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 thiscomments

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-the-two-repeating-elements-in-a-given-array/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 29

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, 0, 1, 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-] 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
   (a) Invariant: a[1..Lo-1]=0 and a[Lo..Mid-1]=1 and a[Hi+1..N]=2; a[Mid..Hi] are unknown.
   (b) 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;
}
```

Time Complexity: O(n)

The above code performs unnecessary swaps for inputs like 0 0 0 0 1 1 1 2 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 http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Flag/

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/sort-an-array-of-0s-1s-and-2s/

Chapter 30

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 *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 *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 *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:

```c
#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)

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/minimum-length-unsorted-subarray-sorting-which-makes-the-complete-array-sorted/

Chapter 31

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)

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-duplicates-in-on-time-and-constant-extra-space/

Category: Arrays

Chapter 32

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 arrya 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 element 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).

```c
#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;
```

```c
    /* 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.

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/equilibrium-index-of-an-array/

Chapter 33

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 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/linked-list-vs-array/

Chapter 34

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 it's 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/which-sorting-algorithm-makes-minimum-number-of-writes/

Category: Arrays

Chapter 35

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 it's 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 x n) matrix into (n x m) matrix,
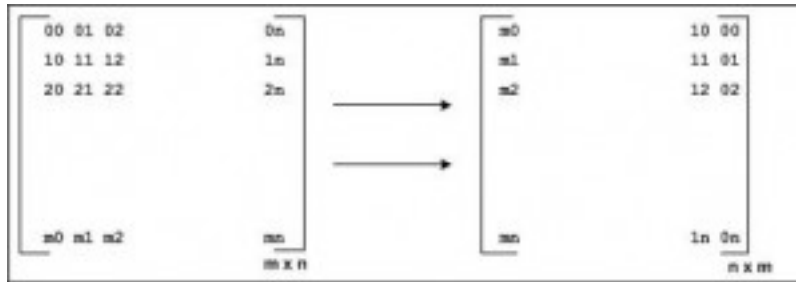
Transformations

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);
        }
    }
}
```

Compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

Chapter 36

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 Sachinfor providing following code.

C/C++

```
// Simple C program to print next greater elements
```

```c
// 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

```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
```

140

Output:


```
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 pchildfor 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.

C


```c
// A Stack based C program to find next greater element
// for all array elements.
#include<stdio.h>
#include<stdlib.h>
#define STACKSIZE 100

// stack structure
struct stack
{
    int top;
    int items[STACKSIZE];
};

// Stack Functions to be used by printNGE()
void push(struct stack *ps, int x)
{
    if (ps->top == STACKSIZE-1)
    {
        printf("Error: stack overflow\n");
        getchar();
        exit(0);
    }
    else
    {
        ps->top += 1;
        int top = ps->top;
        ps->items [top] = x;
```

```
    }
}

bool isEmpty(struct stack *ps)
{
    return (ps->top == -1)? true : false;
}

int pop(struct stack *ps)
{
    int temp;
    if (ps->top == -1)
    {
        printf("Error: stack underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        int top = ps->top;
        temp = ps->items [top];
        ps->top -= 1;
        return temp;
    }
}

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int i = 0;
    struct stack s;
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    push(&s, arr[0]);

    // iterate for rest of the elements
    for (i=1; i<n; i++)
    {
        next = arr[i];

        if (isEmpty(&s) == false)
        {
            // if stack is not empty, then pop an element from stack
            element = pop(&s);

            /* If the popped element is smaller than next, then
                a) print the pair
                b) keep popping while elements are smaller and
                stack is not empty */
            while (element < next)
            {
```

```c
                printf("\n %d --> %d", element, next);
                if(isEmpty(&s) == true)
                    break;
                element = pop(&s);
            }

            /* If element is greater than next, then push
               the element back */
            if (element > next)
                push(&s, element);
        }

        /* push next to stack so that we can find
           next greater for it */
        push(&s, next);
    }

    /* After iterating over the loop, the remaining
       elements in stack do not have the next greater
       element, so print -1 for them */
    while (isEmpty(&s) == false)
    {
        element = pop(&s);
        next = -1;
        printf("\n %d -- %d", element, next);
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Python

```python
# Python program to print next greater element using stack

# Stack Functions to be used by printNGE()
def createStack():
    stack = []
    return stack

def isEmpty(stack):
    return len(stack) == 0

def push(stack, x):
    stack.append(x)
```

```python
def pop(stack):
    if isEmpty(stack):
        print("Error : stack underflow")
    else:
        return stack.pop()

'''prints element and NGE pair for all elements of
   arr[] '''
def printNGE(arr):
    s = createStack()
    element = 0
    next = 0

    # push the first element to stack
    push(s, arr[0])

    # iterate for rest of the elements
    for i in range(1, len(arr), 1):
        next = arr[i]

        if isEmpty(s) == False:

            # if stack is not empty, then pop an element from stack
            element = pop(s)

            '''If the popped element is smaller than next, then
                 a) print the pair
                 b) keep popping while elements are smaller and
                    stack is not empty '''
            while element < next :
                print(str(element)+ " -- " + str(next))
                if isEmpty(s) == True :
                    break
                element = pop(s)

            '''If element is greater than next, then push
               the element back '''
            if  element > next:
                push(s, element)

        '''push next to stack so that we can find
          next greater for it '''
        push(s, next)

    '''After iterating over the loop, the remaining
       elements in stack do not have the next greater
       element, so print -1 for them '''

    while isEmpty(s) == False:
            element = pop(s)
            next = -1
            print(str(element) + " -- " + str(next))
```

```
# Driver program to test above functions
arr = [11, 13, 21, 3]
printNGE(arr)

# This code is contributed by Sunny Karira
```

Output:

```
 11 -- 13
 13 -- 21
  3 -- -1
 21 -- -1
```

Time Complexity: O(n). The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.
a) Initially pushed to the stack.
b) Popped from the stack when next element is being processed.
c) Pushed back to the stack because next element is smaller.
d) Popped from the stack in step 3 of algo.

Source:

http://geeksforgeeks.org/forum/topic/next-greater-element#post-60

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Category: Arrays

Post navigation

← Reverse alternate K nodes in a Singly Linked List Practice Questions for Recursion | Set 5 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Source

http://www.geeksforgeeks.org/next-greater-element/

Chapter 37

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 #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 repitition 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

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

Chapter 38

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 Ravichandrafor 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.

```c
#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/commvault-interview-question-for-software-engineerdeveloper-2-5-years-abou

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-the-first-missing-number/

Chapter 39

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]
```

```c
  /* 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

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/count-number-of-occurrences-in-a-sorted-array/

Category: Arrays Tags: Divide and Conquer

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 40

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

Source

http://www.geeksforgeeks.org/g-fact-84/

Category: Arrays

Chapter 41

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))
```

```c
            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 RMa[] 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.

```c
#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);
```

157

```
        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)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/given-an-array-arr-find-the-maximum-j-i-such-that-arrj-arri/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 42

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 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.

```c
#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(kLogk). Time Complexity of steps 2(a), 2(b) and 2(c) is O(Logk). 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(kLogk + (n-k+1)*Logk) which can also be written as O(nLogk).

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 *Qi* is the smallest of current window. Thanks to Aashishfor suggesting this method.

Following is C++ implementation of this method.


```cpp
#include <iostream>
#include <deque>

using namespace std;

// A Dequeue (Double ended queue) based method for printing maixmum 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 very 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)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

162

Source

Category: Arrays Tags: Queue, Stack-Queue, StackAndQueue

Post navigation

← XOR Linked List – A Memory Efficient Doubly Linked List | Set 1 Time Complexity of building a heap →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 43

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;
```

```c
    }

    /* 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.
```

```c
#include<stdio.h>

/* Fucntion 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;
    }
```

165

```
    /* 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(mLogm + nLogm). Please note that this will be the complexity if an mLogm algorithm is used for sorting which is not the case in above code. In above code Quick Sort is sued and worst case time complexity of Quick Sort is O(m^2)

Method 3 (Use Sorting and Merging)
1) Sort both arrays: arr1[] and arr2[] O(mLogm + nLogn)
2) Use Merge type of process to see if all elements of sorted arr2[] are present in sorted arr1[].

Thanks to Parthsarthifor 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(mLogm + nLogn) which is better than method 2. Please note that this will be the complexity if an nLogn algorithm is used for sorting both arrays which is not the case in above code. In above code Quick Sort is sued and worst case time complexity of Quick Sort is O(nˆ2)

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

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-whether-an-array-is-subset-of-another-array-set-1/

Category: Arrays

Post navigation

← A Boolean Array Puzzle Dynamic Programming | Set 4 (Longest Common Subsequence) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 44

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, 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 fnction */
int main()
{
    int arr[] = {3, 5, 4, 2, 6, 5, 6, 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 wgpshashankfor 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 occurence of any of the two numbers (x or y)
    // and store the index of this occurence in prev
    for (i = 0; i < n; i++)
    {
        if (arr[i] == x || arr[i] == y)
        {
            prev = i;
            break;
        }
    }
```

```
    // Traverse after the first occurence
    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 minimm 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 fnction */
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)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 45

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>
```

```c
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 – x + y
2) Get product of all numbers.
Product of array computed P = 1*2*3*…*n * y / x
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.

```c
#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.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-a-repeating-and-a-missing-number/

Chapter 46

Median in a stream of integers (running integers)

Given that integers are read from a data stream. Find median of elements read so for 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.

```cpp
#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];
```

```cpp
            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();
    }
```

```cpp
    // 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
        {
```

```
            // Remore 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.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/

Category: Arrays

Post navigation

← Advanced C++ | Virtual Constructor Advanced C++ | Virtual Copy Constructor →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 47

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 pmfor 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)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-a-fixed-point-in-a-given-array/

Category: Arrays Tags: Divide and Conquer

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 48

Maximum Length Bitonic Subarray

Given an array A[0 ... n-1] containing n positive integers, a subarray A[i ... j] is bitonic if there is a k with i = A[k + 1] >= .. A[j − 1] > = 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 bitnoic, 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 soultion.
1) Construct an auxiliary array inc[] from left to right such that inc[i] contains length of the nondecreaing subarray ending at arr[i].
For 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.

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

int bitonic(int arr[], int n)
{
    int i;
    int *inc = new int[n];
    int *dec = new int[n];
```

```
    int max;
    inc[0] = 1; // The length of increasing sequence ending at first index is 1
    dec[n-1] = 1; // The length of increasing sequence starting at first index is 1

    // Step 1) Construct increasing sequence array
    for(i = 1; i < n; i++)
    {
        if (arr[i] > arr[i-1])
            inc[i] = inc[i-1] + 1;
        else
            inc[i] = 1;
    }

    // Step 2) Construct decreasing sequence array
    for (i = n-2; i >= 0; i--)
    {
        if (arr[i] > arr[i+1])
            dec[i] = dec[i+1] + 1;
        else
            dec[i] = 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;
        }
    }

    // free dynamically allocated memory
    delete [] inc;
    delete [] dec;

    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("\n Length of max length Bitnoic Subarray is %d", bitonic(arr, n));
    getchar();
    return 0;
}
```

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.

188

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/maximum-length-bitonic-subarray/

Chapter 49

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 and 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++)
   {
   {
```

```c
        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}


```c
#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}.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-maximum-element-in-an-array-which-is-first-increasing-and-then-decreasing/

Category: Arrays Tags: Divide and Conquer

Post navigation

← Connect nodes at same level using constant extra space Sorted Linked List to Balanced BST →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 50

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 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
```

```c
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 rotted 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 2 0 0
```

Time Complexity: O(nLogn)
Auxiliary Space: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/count-smaller-elements-on-right-side/

Category: Arrays

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 51

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.

minJumps(start, end) = Min (minJumps(k, end)) for all k reachable from start

```c
#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, minJumps(3, 9) will be called two times as arr[3] is reachable from arr[1] and 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 jumps[] array from left to right such that jumps[i] indicates the minimum number of jumps needed to reach arr[i] from arr[0]. Finally, we return 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 parasfor 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 direcly 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 Ashishfor suggesting this solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/

Category: Arrays Tags: Dynamic Programming

Post navigation

← Count smaller elements on right side Populate Inorder Successor for all nodes →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 52

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 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
{
```

```cpp
    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
```

```cpp
        {
            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
class twoStacks:
    arr = [None]
    size,top1,top2 = 0,0,0

    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
```

```python
    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()

    # 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()))
```

Output:

```
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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/implement-two-stacks-in-an-array/

Chapter 53

Find subarray with given sum

Given an unsorted array of nonnegative integers, find a continous subarray which adds to a given number.

Examples:

```
Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Ouptut: Sum found between indexes 2 and 4

Input: arr[] = {1, 4, 0, 0, 3, 10, 5}, sum = 7
Ouptut: 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 the 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];
```

```c
        // 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 elemnents while curr_sum is greater than sum.

Following is C implementation of the above approach.

```c
/* An efficient program to print subarray with sum as given sum */
#include<stdio.h>

/* Returns true if the 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)
{
```

```c
    /* 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).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-subarray-with-given-sum/

Category: Arrays

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 54

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 intgers 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])
```

```c
            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

```python
# Dynamic Programming bsed 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/dynamic-programming-set-14-maximum-sum-increasing-subsequence/

Chapter 55

Longest Monotonically 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 monotonically 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 monotone 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 can't 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 doesn't 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 learnt 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,

```
#include <iostream>
#include <string.h>
#include <stdio.h>
```

```cpp
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) {
    int m;

    while( r - l > 1 ) {
        m = l + (r - l)/2;
        (A[m] >= key ? r : l) = m; // ternary expression returns an l-value
    }

    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] )
            // 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;
}
```

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** monotonically increasing list**s of equal longest size**.

3. Is the above algorithm an *online* algorithm?

4. Design an algorithm to construct the longest *decreasing* list..

— Venki. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 56

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 thispost.

```
# 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
```

222

```c
        // 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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-a-triplet-that-sum-to-a-given-value/

Category: Arrays

Post navigation

← Longest Monotonically Increasing Subsequence Size (N log N) Find the smallest positive number missing from an unsorted array →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 57

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.

```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++)
```

```c
    if (arr[i] > 0)
        return i+1;  // 1 is added becuase 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 loose 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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-smallest-positive-number-missing-from-an-unsorted-array/

Chapter 58

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 thispost. 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 ˆ n = 0
2) XOR of any number n with 0 gives us n, i.e., n ˆ 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 number of times will result in 0 in its set. And the xor of a set will be one of the odd occurring elements.

```c
// 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)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-two-numbers-with-odd-occurences-in-an-unsorted-array/

Category: Arrays

Post navigation

← Find the smallest positive number missing from an unsorted array Pattern Searching | Set 7 (Boyer Moore Algorithm – Bad Character Heuristic) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 59

The Celebrity Problem

Another classical problem.

*In a party of N people, only one person is known to everyone. Such a person **may be 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 *Polya's 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 verity 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.

```cpp
#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 HaveAcquiantance(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( HaveAcquiantance(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( HaveAcquiantance(C, B) )
        C = B;

    if( HaveAcquiantance(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( HaveAcquiantance(C, i) )
            return -1;

        // i must know C
        if( !HaveAcquiantance(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 *HaveAcquiantance(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 *HaveAcquiantance* 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 the *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,

```
 L0 0->0->1->0
              |
L1            0->0->1->0
                      |
L2                    0->0->1->0
                              |
L3                            0->0->1->0
```
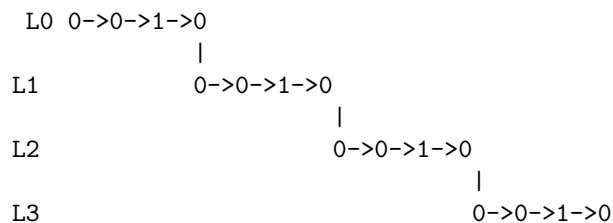
The function *HaveAcquanintance(i, j)* will search in the list for *j-th* node in the *i-th* level. Out goal is to minimize calls to *HaveAcquanintance* function.

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()*)?

— Venki. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/the-celebrity-problem/

Chapter 60

Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

Given an array arr[0 … n-1] containing n positive integers, a subsequenceof 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.

```
/* 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 ) );

  getchar();
  return 0;
}
```

Output:

```
Length of LBS is 7
```

Time Complexity: O(nˆ2)
Auxiliary Space: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/dynamic-programming-set-15-longest-bitonic-subsequence/

Category: Arrays Tags: Dynamic Programming

Post navigation

← Use of explicit keyword in C++ Dynamic Programming | Set 16 (Floyd Warshall Algorithm) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 61

Find a sorted subsequence of size 3 in linear time

Given an array of n integers, find the 3 elements such that a[i]

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 fund 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;
```

```cpp
    // 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 alloced 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)
Auxliary 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 arr[i] arr[k].
2. Find a sorted subsequence of size 4 in linear time

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-a-sorted-subsequence-of-size-3-in-linear-time/

Category: Arrays

Post navigation

← Dynamic Programming | Set 16 (Floyd Warshall Algorithm) Operating Systems | Set 6 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 62

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.

```c
// 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;
```

```c
    // 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.

```c
// 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 sunmleft[]. 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
```

```c
        // 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)

244

Thanks to Aashish Barnwal for suggesting this solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/largest-subarray-with-equal-number-of-0s-and-1s/

Category: Arrays

Post navigation

← Computer Networks | Set 6 Computer Networks | Set 7 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 63

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])



// A recursive solution 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 findPartiion (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 (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;
}
```

Output:

```
 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 (sum/2)*(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
```

```
// A Dynamic Programming solution 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;

    // Caculcate sun 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 botton up manner
     for (i = 1; i <= sum/2; i++)
     {
```

248

```
      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 funtion
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;
}
```

Output:

```
 Can be divided into two subsets of equal sum
```

Following diagram shows the values in partition table. The diagram is taken form 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/

Category: Arrays Tags: Dynamic Programming

Post navigation

← Automata Theory | Set 5 Database Management Systems | Set 11 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 64

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 ouput. 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.

```c
#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 throught 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 alway 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()
{
```

252

```
    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)

This article is compiled by **Dheeraj Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/maximum-product-subarray/

Category: Arrays

Post navigation

← Count numbers that don't contain 3 Magic Square →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 65

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(nLogn). 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 arr[i], binary search for arr[i] + n in arr[i+1..n-1]. If the element is found, return the pair.
Both first and second steps take O(nLogn). So overall complexity is O(nLogn).

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 arr[j] – arr[i] is smaller than n, we need to look for greater arr[j], so increment j. If arr[j] – arr[i] is greater than n, we need to look for greater 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 + arr[i] in HT.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-a-pair-with-the-given-difference/

Category: Arrays

Chapter 66

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*n).
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.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/replace-every-element-with-the-greatest-on-right-side/

Category: Arrays

Chapter 67

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 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 problemand can be solved in (nLogn) 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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/dynamic-programming-set-20-maximum-length-chain-of-pairs/

Chapter 68

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
```

```c
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(nˆ3) 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(nˆ3) solution.

```c
# 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(nˆ3)

This problem can also be solved in O(nˆ2Logn) complexity. We will soon be publishing the O(nˆ2Logn) solution as a separate post.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-four-numbers-with-sum-equal-to-given-sum/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 69

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^2Logn) time with the help of auxiliary space.

Thanks to itsnimishfor 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 ofthis postto 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[] don't 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^2Logn) time using merge sort or heap sort or any other O(nLogn) algorithm. The third step takes O(n^2) time. So overall complexity is O(n^2Logn).

Auxiliary Space: O(n^2). The big size of auxiliary array can be a concern in this method.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-four-elements-that-sum-to-a-given-value-set-2/

Chapter 70

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 Logk time. So overall complexity will be O(k) + O((n-k)*logK)

```cpp
#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
```

```cpp
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;
}

// 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(nLogk) 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(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed 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.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/nearly-sorted-algorithm/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 71

Maximum circular subarray sum

Given n numbers (both +ve and -ve), arranged in a circle, fnd 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 Kadane's 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 ashishdey0 for suggesting this solution. Following is C implementation of the above method.

```c
// 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);

// The function returns maximum circular contiguous sum in a[]
int maxCircularSum (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.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/maximum-contiguous-circular-sum/

Category: Arrays

Post navigation

← What happens when a function is called before its declaration in C? Amazon Interview | Set 7 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 72

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 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(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.

276

...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.

```cpp
// 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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-row-with-maximum-number-1s/

Chapter 73

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 – 1], 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 10, the median is 12.
So, in this case, find the median of three elements B[M / 2 − 1], 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.

```c
// 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)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

Category: Arrays Tags: Divide and Conquer

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 74

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".
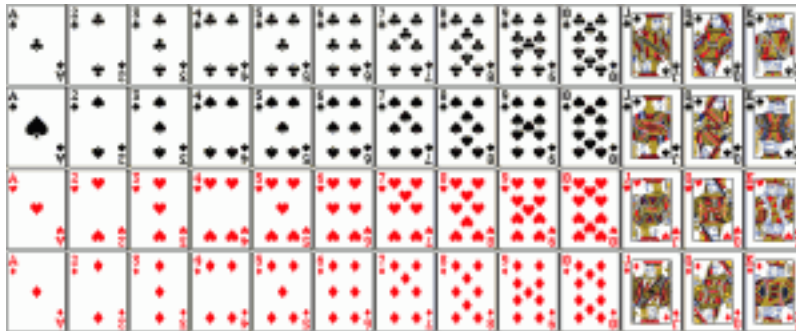
Let the given array be *arr[]*. A simple solution is to create an auxiliary array *temp[]* which is initially a copy of *arr[]*. Randomly select an element from *temp[]*, copy the randomly selected element to *arr[0]* and remove the selected element from *temp[]*. Repeat the same process n times and keep copying elements to *arr[1]*, *arr[2]*, ... The time complexity of this solution will be O(n^2).

Fisher–Yates 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
Following is C++ implementation of this algorithm.

// C Program to shuffle a given array

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

```c
// A utility function to swap to 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 ith element (including the last one) goes to last position is 1/n, because we randomly pick an element in first iteration.

The probability that ith 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) x (probability that the index picked in previous step is picked again so that the last element is swapped)

So the probability = ((n-1)/n) x (1/(n-1)) = 1/n

Case 2: 0 :

The probability of ith element going to second position = (probability that ith element is not picked in previous iteration) x (probability that ith element is picked in this iteration)

So the probability = ((n-1)/n) x (1/(n-1)) = 1/n

We can easily generalize above proof for any other position.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/shuffle-a-given-array/

Chapter 75

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] 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.

```c
// 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-number-of-triangles-possible/

Chapter 76

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.

```c
// 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 + 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 ];
```

```c
    // 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

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/iterative-quick-sort/

Category: Arrays

Post navigation

← Output of C++ Program | Set 17 Longest Palindromic Substring | Set 1 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 77

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.
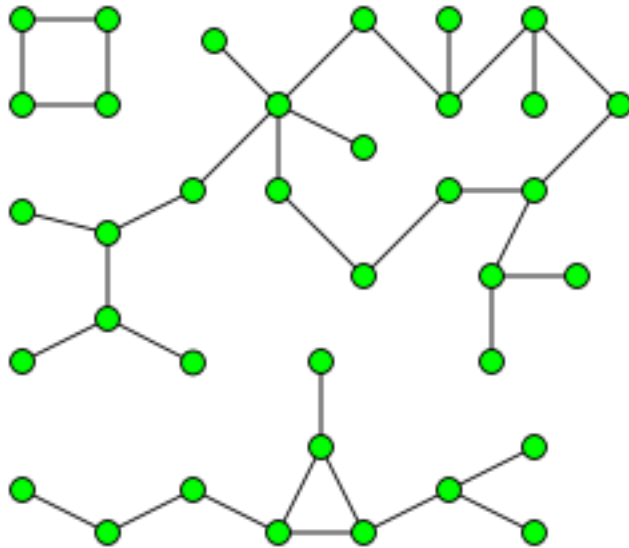
A graph where all vertices are connected with each other, has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

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
// 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])
{
    return (row >= 0) && (row < ROW) &&     // row number is in range
           (col >= 0) && (col < COL) &&     // column number is in range
           (M[row][col] && !visited[row][col]); // value is 1 and not yet visited
}

// A utility function to do DFS for a 2D boolean matrix. It only considers
// the 8 neighbors 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 neighbors
    // 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)
        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[][COL])
{
    // 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 travese 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;
}
```

Output:

```
 Number of islands is: 5
```

Time complexity: O(ROW x COL)

Reference:

http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-number-of-islands/

Category: Arrays Graph Tags: Graph

Post navigation

← Floor and Ceil from a BST Reservoir Sampling →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 78

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 condidate 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?

— Venki. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/construction-of-longest-monotonically-increasing-subsequence-n-log-n/

Category: Arrays

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 dequeueing 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
```

```c
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 curremt 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)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-a-tour-that-visits-all-stations/

Chapter 80

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 vectoris used instead of normal array.

```cpp
// 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);
```

```cpp
    // 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 funtion. 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];
}

// driverr 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 777776
    /*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
```

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/given-an-array-of-numbers-arrange-the-numbers-to-form-the-biggest-number/

Chapter 81

Pancake sorting

Given an 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 it's 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_szie-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.

[iframe width="340" height="252" src="http://www.youtube.com/embed/kk-_DDgoXfk" frameborder="0]

```cpp
/* 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;
```

```
}
```

Output:

```
 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/pancake-sorting/

Category: Arrays

Chapter 82

A Pancake Sorting Problem

We have discussed Pancake Sorting in the previous post. Following is a problem based on Pancake Sorting. Given an 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 find 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(Logi) 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  O(n)
{
  int key = arr[i];

  // Find index of ceiling of arr[i] in arr[0..i-1] using binary search
  j = celiSearch(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(nlogn). 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

308

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.

```c
#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);
}
```

```c
/* 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
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/a-pancake-sorting-question/

Category: Arrays

Post navigation

← Amazon Interview | Set 22 Swap all odd and even bits →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 83

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 solutions 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 boolen 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
```

This article is compiled by Ashish Anand and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/tug-of-war/

Category: Arrays Tags: Backtracking

Post navigation

← Interview Experiences at D. E. Shaw & Co. Print Matrix Diagonally →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 84

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(nLogn) 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 funtion to find maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// A utility funtion 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[] auch that arr[m] is part of it
int maxCrossingSum(int arr[], int l, int m, int h)
```

```c
{
    // 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 maxium 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) + Θ(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 Θ(nLogn).

The Kadane's Algorithm for this problem takes O(n) time. Therefore the Kadane's 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(nLogn).

References:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/divide-and-conquer-maximum-sum-subarray/

Chapter 85

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
// C Program for counting sort
#include <stdio.h>
#include <string.h>
#define RANGE 255

// The main function that sort the given string str in alphabatical order
void countSort(char *str)
```

```c
{
    // The output character array that will have sorted str
    char output[strlen(str)];

    // Create a count array to store count of inidividul 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. It 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.

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/counting-sort/

Category: Arrays

Chapter 86

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 interval[i] doesn't overlap with interval[i-1], then interval[i+1] cannot overlap with interval[i-1] because starting time of interval[i+1] must be greater than or equal to 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
```

```cpp
struct Interval
{
    int start, end;
};

// Compares two intervals according to their staring 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(nLogn) 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.

This article is compiled by Ravi Chandra Enaganti. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/merging-intervals/

Category: Arrays Tags: stack

Post navigation

← Construct Complete Binary Tree from its Linked List Representation Amazon Interview | Set 23 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 87

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 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(nˆ2).*

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.

```
#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;
}
```

Output:

```
 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.

This article is compiled by Ashish Anand and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-maximum-repeating-number-in-ok-time/

Chapter 88

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 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)

This article is compiled by Ashish Anand and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/stock-buy-sell/

Category: Arrays

Chapter 89

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.

```
// 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)
        {
```

```c
            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;
}
```

Output:

```
   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.

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

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.

Source

http://www.geeksforgeeks.org/rearrange-positive-and-negative-numbers-publish/

Category: Arrays

Post navigation

← Amazon Interview | Set 24 HCL Interview | Set 1 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 90

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(nLogn) 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(nLohn) time if a O(nLogn) 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(nLogn) if we use a O(nLogn) sorting algorithm and use a self balancing BST with O(Logn) insert operation.

Following is C++ implementation of the above algorithm.

```cpp
// 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
{
```

```cpp
    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 substree
    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, 3, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortByFrequency(arr, n);
    printArray(arr, n);
    return 0;
}
```

Output:

```
 3 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.

This article is compiled by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/sort-elements-by-frequency-set-2/

Category: Arrays

Post navigation

← Amazon Interview | Set 28 Amazon Interview | Set 29 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 91

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 is C implementation of this approach.

```c
// A divide and conquer solution to find a peak element element
#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 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()
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;
}
```

Output:

```
 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=HtSuA80QTyo

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-a-peak-in-a-given-array/

Chapter 92

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.
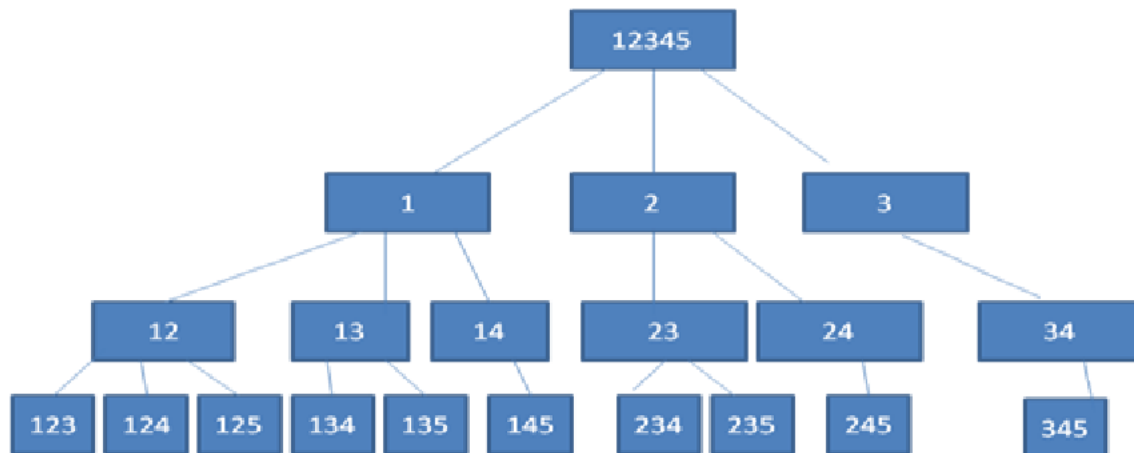
Following is C++ implementation of above approach.

```
// 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);
```

```c
// 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 temprary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/* arr[]  ---> Input Array
   data[] ---> Temporary array to store current combination
   start & end ---> Staring 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);
}
```

Output:

```
 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 Pascal's Identity, i.e. $\mathbf{n_{c_r} = n\text{-}1_{c_r} + n\text{-}1_{c_{r-1}}}$

Following is C++ implementation of method 2.

```
// 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 temprary 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 cobination 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;
}
```

Output:

```
 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 following 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.

This article is contributed by **Bateesh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/

Category: Arrays Tags: MathematicalAlgo, Recursion

Post navigation

← Fab.com Pune Interview Experience Goldman Sachs Interview Experience →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 93

Given an array of 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(nLogn) 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(nLogn) + O(n) which is O(nLogn).

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 temp[] (add/remove an element or increase/decrease count) for every traversed element. The array 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 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 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.

```cpp
// 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(nLogk) time with the help of more appropriate data structures than array for auxiliary storage of k-1 elements. Suggest a O(nLogk) approach.

This article is contributed by **Kushagra Jaiswal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-an-array-of-of-size-n-finds-all-the-elements-that-appear-more-than-nk-times/

Category: Arrays

Post navigation

← Goldman Sachs Interview Experience Adobe Interview | Set 5 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 94

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 can't directly apply Binary Search as we don't 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*O(Logn) which is O(Logn).

```c
#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 monotonocally 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
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-the-point-where-a-function-becomes-negative/

Chapter 95

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(n²).

We can do this **in O(nLogn) 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 fill LSL[] in O(nLogn) 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 thispost. 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(nLogn) + O(n) + O(n) which is O(nLogn). 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.

This article is contributed by**Amit Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/increasing-subsequence-of-length-three-with-maximum-product/

Chapter 96

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(Logn) 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 C implementation.

```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];
```

```c
    // 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;
}
```

Output:

```
 The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
```

How to handle duplicates?
It turned out that duplicates can't 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, 2, 0, 1, 1, 2} and {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't 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 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));
```

```c
    // 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, 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, 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, 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, 2};
    int n7 = sizeof(arr7)/sizeof(arr7[0]);
    printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

    return 0;
}
```

Output:

```
 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
```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-minimum-element-in-a-sorted-and-rotated-array/

Category: Arrays Tags: Divide and Conquer

Post navigation

← Print ancestors of a given binary tree node without recursion Ford-Fulkerson Algorithm for Maximum Flow Problem →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 97

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 m1 and m2 and two women w1 and w2.
Let m1's list of preferences be {w1, w2}
Let m2's list of preferences be {w1, w2}
Let w1's list of preferences be {m1, m2}
Let w2's list of preferences be {m1, m2}

The matching { {m1, w2}, {w1, m2} } is not stable because m1 and w1 would prefer each other over their assigned partners. The matching {m1, w1} and {m2, w2} 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 Gale–Shapley 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.

```cpp
// 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 engagment m1
    for (int i = 0; i < N; i++)
    {
        // If m1 comes before m in lisr of w, then w prefers her
        // cirrent engagement, don't do anything
        if (prefer[w][i] == m1)
             return true;

        // If m cmes 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 numbereed as N to 2N-1.
void stableMarriage(int prefer[2*N][N])
{
    // Stores partner of women. This is our output array that
    // stores paing information.  The value of wPartner[i]
    // indicates the partner assigned to woman N+i.  Note that
    // the woman numbers between N and 2*N-1. The value -1
    // indicates that (N+i)'th woman is free
    int wPartner[N];

    // An array to store availability of men.  If mFree[i] is
    // false, then man 'i' is free, otherwise engaged.
    bool mFree[N];

    // Initialize all men and women as free
    memset(wPartner, -1, sizeof(wPartner));
    memset(mFree, false, sizeof(mFree));
    int freeCount = N;
```

```cpp
    // While there are free men
    while (freeCount > 0)
    {
        // Pick the first free man (we could pick any)
        int m;
        for (m = 0; m < N; m++)
            if (mFree[m] == false)
                break;

        // One by one go to all women according to m's preferences.
        // Here m is the picked free man
        for (int i = 0; i < N && mFree[m] == false; i++)
        {
            int w = prefer[m][i];

            // The woman of preference is free, w and m become
            // partners (Note that the partnership maybe changed
            // later). So we can say they are engaged not married
            if (wPartner[w-N] == -1)
            {
                wPartner[w-N] = m;
                mFree[m] = true;
                freeCount--;
            }

            else  // If w is not free
            {
                // Find current engagement of w
                int m1 = wPartner[w-N];

                // If w prefers m over her current engagement m1,
                // then break the engagement between w and m1 and
                // engage m with w.
                if (wPrefersM1OverM(prefer, w, m, m1) == false)
                {
                    wPartner[w-N] = m;
                    mFree[m] = true;
                    mFree[m1] = false;
                }
            } // End of Else
        } // End of the for loop that goes to all women in m's list
    } // End of main while loop


    // Print the solution
    cout << "Woman   Man" << endl;
    for (int i = 0; i < N; i++)
        cout << " " << i+N << "\t" << wPartner[i] << endl;
}

// Driver program to test above functions
int main()
{
    int prefer[2*N][N] = { {7, 5, 6, 4},
```

```
        {5, 4, 6, 7},
        {4, 5, 6, 7},
        {4, 5, 6, 7},
        {0, 1, 2, 3},
        {0, 1, 2, 3},
        {0, 1, 2, 3},
        {0, 1, 2, 3},
    };
    stableMarriage(prefer);

    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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/stable-marriage-problem/

Category: Arrays

Post navigation

← Find minimum s-t cut in a flow network Convex Hull | Set 2 (Graham Scan) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 98

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(nLogn) sorting algorithm. This approach takes O(nkLognk) time.

We can merge arrays in O(nk*Logk) 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 doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Following is C++ implementation of the above algorithm.

```cpp
// 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
```

```cpp
    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 sonn be discussing them as separate posts.

Thanks to vigneshfor suggesting this problem and initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/merge-k-sorted-arrays/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 99

Radix Sort

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

Counting sort is a linear tine sorting algorithm that sort 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 c where c is a constant. In that case, the complexity becomes $O(nLog_b(n))$. 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++ 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 curent 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;
}
```

Output:

```
 2 24 45 66 75 90 170 802
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/radix-sort/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 100

Move all zeroes to end of array

Given an array of random numbers, Push all the zero's 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, 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.

```cpp
// 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;
}
```

Output:

```
 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)

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/move-zeroes-end-array/

Chapter 101

Find number of pairs 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(nLogn + mLogn)** 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 don't follow the general rule that x smaller than y means x^y is greater than 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
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

X

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.

```cpp
#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 pais is equal to number of
```

```cpp
    // 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).

This article is contributed by **Shubham Mittal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-number-pairs-xy-yx/

Category: Arrays

Post navigation

← How to count set bits in a floating point number in C? Analysis of Algorithms | Set 3 (Asymptotic Notations) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 102

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++)
```

381

```
                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(nLogn) time using a O(nLogn) 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(nLogn) time. The second step runs binary search n times, so the time complexity of second step is also O(nLogn). Therefore, overall time complexity is O(nLogn). The second step can be optimized to O(n), see this.

Method 3 (Use Self-balancing BST)
We can also 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(nLogn) as search and delete operations take O(Logn) 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)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/count-pairs-difference-equal-k/

Category: Arrays

Post navigation

← DELL Interview | Set 1 (On-Campus) How to compare two arrays in Java? →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 103

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.
```

We strongly recommend to minimize the browser and try this yourself first.

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 form 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 sero 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.

This article is contributed by **Chirag Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-if-there-is-a-subarray-with-0-sum/

Category: Arrays Tags: Hashing

Chapter 104

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.

```cpp
# 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)
{
    //  Initilize length of smallest subarray as n+1
```

```cpp
    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 curremt 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 thispost. Thanks to Ankit and Nitin for suggesting this optimized solution.

```cpp
// 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
```

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/minimum-length-subarray-sum-greater-given-value/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 105

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)

We strongly recommend to minimize the browser and try this yourself first.

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 vivekfor suggesting this method. Following is C++ implementation of above algorithm.

```cpp
// 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 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 self balancing 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 self balancing 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 toAnurag Sigh 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.

```cpp
// 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 accesed 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 comapre 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.

This article is compiled by **Piyush**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/sort-array-according-order-defined-another-array/

Category: Arrays

Post navigation

← Microsoft Interview | Set 32 (On-Campus for Internship) Amazon Interview | Set 115 (On-Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 106

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
```

We strongly recommend to minimize the browser and try this yourself first.

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++

```cpp
#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);

            // 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]:
```

401

```
        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__
```

Output:

```
 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).

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/maximum-sum-path-across-two-arrays/

402

Chapter 107

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 Θ(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
```

This article is contributed by **Anuj**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/check-given-array-contains-duplicate-elements-within-k-distance/

Category: Arrays Tags: Hashing

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 108

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[3] = ...

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
```

We strongly recommend to minimize your browser and try this yourself first.

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++

```cpp
// A C++ program to sort an array in wave form using a sorting function
#include<iostream>
#include<algorithm>
```

```cpp
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
# 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 progrM
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:

```
 2 1 10 5 49 23 90
```

The time complexity of the above solution is O(nLogn) if a O(nLogn) 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++

```cpp
// 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
```

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sort-array-wave-form-2/

Chapter 109

K'th 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 k'th smallest element in the given array. It is given that ll 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).

```cpp
// 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 k'th smallest element in time complexity better than O(nLogn). A simple optomization is to
create a Min Heapof 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)
    {
```

```cpp
        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 vakue.
    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 k'th 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 k'th 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
{
```

```cpp
    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 vakue.
    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 QuickSortis 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 k'th 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.

```cpp
#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-l == k-1)
            return arr[pos];
        if (pos-l > 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).

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

K'th 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/

Category: Arrays

Post navigation

← Time complexity of insertion sort when there are O(n) inversions? Amazon Interview Experience | Set 151 (For SDE) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 110

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th 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 k'th smallest element in the given array. It is given that ll 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 previouspost. 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.

```cpp
// 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-l == k-1)
            return arr[pos];
        if (pos-l > 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++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] arount 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.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 111

K'th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

We recommend to read following posts as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

K'th 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 k'th smallest element in the given array. It is given that ll 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 n/5rceil; 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  n/5  groups and find median
   of all groups. Create an auxiliary array 'median[]' and store medians
```

421

```
    of all n/5 groups in this median array.

// Recursively call this method to find median of median[0.. n/5 -1]
3) medOfMed = kthSmallest(median[0.. n/5 -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[l..pos-1], k)
7) If poa > k return kthSmallest(arr[pos+1..r], k-pos+l-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.

```cpp
// 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-l == k-1)
            return arr[pos];
        if (pos-l > 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 $7n/10 + 6$ 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) cn for some constant c and all n > 80. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

```
T(n)
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
Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. The
Sources:
```

424

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/~csli/graduate/algorithms/book6/chap10.htm
This article is contributed by Shivam. Please write comments if you find anything incorrect, or you want to sl

Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/

Category: Arrays

Post navigation

← Check a given sentence for a given set of simple grammer rules [TopTalent.in] Interview With Faraz Who Got Into MobiKwik →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 112

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, 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
```

We strongly recommend to minimize the browser and try this yourself first.

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 $O(n^2)$.

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 is C++ implementation of the above algorithm.

```
// C++ program to find Index of 0 to be replaced with 1 to get
// longest continuous sequence of 1s in a binary array
```

```cpp
#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, 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;
}
```

Output:

```
 Index of 0 to be replaced is 9
```

Time Complexity: O(n)
Auxiliary Space: O(1)

This article is contributed by **Ankur Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-index-0-replaced-1-get-longest-continuous-sequence-1s-binary-array/

Category: Arrays

Post navigation

← Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes
Check a given sentence for a given set of simple grammer rules →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 113

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
```

We strongly recommend to minimize your browser and try this yourself first.

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 = 0
       (a) If abs(ar1[l] + ar2[r] - sum)
Following is C++ implementation of this approach.

```cpp
// C++ program to find the pair from two sorted arays 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]
```

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-two-sorted-arrays-number-x-find-pair-whose-sum-closest-x/

Category: Arrays

Post navigation

← Adobe Interview | Set 17 (For MTS-1) Multiply two polynomials →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 114

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}
Outptu: 5, 5
```

A simple solution is to first find intersection of two arraysand store the intersection in a temporary array, then find the intersection of third array and temporary array. Time complexity of this solution is O(n1 + n2 + n3) where n1, n2 and n3 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 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++

```cpp
// 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
```

```python
        # Iterate through three arrays while all arrays have elements
        while (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__
```

Output:

```
 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.

This article is compiled by **Rahul Gupta** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-common-elements-three-sorted-arrays/

434

Chapter 115

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 $O(nLogn)$ time. Following are detailed steps.
1) Copy the given array to an auxiliary array temp[].
2) Sort the temp array using a $O(nLogn)$ 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 $O(nLogn)$ 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[]
```

```java
    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
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

Category: Arrays Tags: Hashing

Chapter 116

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
```

We strongly recommend to minimize the browser and try this yourself first.

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] – 1 be adding 'arr[i]' to all subsets that represent 1 to 'res')

Following is C++ implementation of above idea.

```cpp
// 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

438

```
4
5
10
```

Time Complexity of above program is O(n).

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-smallest-value-represented-sum-subset-given-array/

Category: Arrays

Post navigation

← [TopTalent.in] Exclusive Rapid Fire Interview with Gritika Who Got a job offer from Microsoft Nvidia Interview | Set 2 (On Campus for R & D Team, Pune) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 117

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[]
```

```c
// 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
```

442

```
// 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.

This article is contributed by **Arun Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/rearrange-array-arrj-becomes-arri-j/

Category: Arrays

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 118

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 don't know size of array.
If the array is infinite, that means we don't 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++

```cpp
// 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;
}
```

```cpp
// 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;
}
// 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
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 $O(Log\ p)$.

This article is contributed by **Gaurav Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-position-element-sorted-array-infinite-numbers/

Chapter 119

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 progran are copied from K'th 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-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left
            return kthSmallest(arr, l, pos-1, k);
```

```cpp
        // 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(nLogn), 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(nLogn) worst case time complexity.

This article is compiled by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/

Category: Arrays Tags: Sorting

Post navigation

← Belzabar Software Interview Experience | Set 4 Microsoft Interview Experience | Set 46 (Onsite) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 120

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]

We strongly recommend to minimize your browser and try this yourself first.

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.

```cpp
// 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;
```

```cpp
    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).

This article is contributed by **Rishabh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-the-largest-pair-sum-in-an-unsorted-array/

Category: Arrays

Chapter 121

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    // "aabaacaaba" is not palindrome
         a Yes   // "aabaacaabaa" is palindrome
```

Let input string be str[0..n-1]. A **Simple Solution** is to do following for every character str[i] in input string. Check if substring 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 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'
firstr' = hash("a"), 'second' = hash("b")

Start from second character, i.e.,
i = 1
a) Compare 'firstr' and 'second', they don't 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 don't 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 don't 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 hash("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
```

```c
#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 firstr and 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 'firstr' and 'second'
        // match, then only check individual characters
        if (firstr == second)
        {
            /* Check if 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 '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*n), 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".

This article is contributed by **Ajay**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

Chapter 122

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

460

.......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 + nLogm, mLogn + nLogn) 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.

```cpp
// 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 *tempp = arr1;
        arr1 = arr2;
        arr2 = tempp;

        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] << " ";
}
```

```cpp
// 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 *tempp = arr1;
        arr1 = arr2;
        arr2 = tempp;

        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.

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/find-union-and-intersection-of-two-unsorted-arrays/

Chapter 123

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.

```cpp
// A C++ program that returns true if there is a Pythagorean
// Triplet in a given aray.
#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;
}
```

Output:

```
 Yes
```

Time Complexity of the above solution is O(n³).

Method 2 (Use Sorting)
We can solve this in O(n²) 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(nLogn) time.

3) To find a triplet (a, b, c) such that a = b + c, do following.

1. Fix 'a' as last element of sorted array.
2. 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.
3. 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.

```
// 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;
}
```

Output:

```
  Yes
```

Time complexity of this method is $O(n^2)$.

This article is contributed by **Harshit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-pythagorean-triplet-in-an-unsorted-array/

Category:

Post navigation

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 124

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.
```

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to 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 aubarray 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++

```cpp
// 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
```

```cpp
        //     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

```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):
```

```python
            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__
```

Output:

```
 Maximum Profit = 100
```

Time complexity of the above solution is O(n).

Algorithmic Paradigm: Dynamic Programming

This article is contributed by **Amit Jaiswal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-twice/