Contents

Chapter 1

Advanced master theorem for divide and conquer recurrences

Advanced master theorem for divide and conquer recurrences - GeeksforGeeks

Master Theorem is used to determine running time of algorithms (divide and conquer algorithms) in terms of asymptotic notations.
Consider a problem that be solved using recursion.

```
function f(input x size n)
if(n > k)
solve x directly and return
else
divide x into a subproblems of size n/b
call f recursively to solve each subproblem
Combine the results of all sub-problems
```

The above algorithm divides the problem into **a** subproblems, each of size n/b and solve them recursively to compute the problem and the extra work done for problem is given by f(n), i.e., the time to create the subproblems and combine their results in the above procedure.

So, according to master theorem the runtime of the above algorithm can be expressed as:

```
T(n) = aT(n/b) + f(n)
```

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

Not all recurrence relations can be solved with the use of the master theorem i.e. if

- T(n) is not monotone, ex: T(n) = sin n
- f(n) is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$

This theorem is an advance version of master theorem that can be used to determine running time of divide and conquer algorithms if the recurrence is of the following form :-

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = (n^{\log_b a})$
2. if $a = b^k$, then
   (a) if p > -1, then $T(n) = (n^{\log_b a} \log^{p+1} n)$
   (b) if p = -1, then $T(n) = (n^{\log_b a} \log\log n)$
   (c) if p < -1, then $T(n) = (n^{\log_b a})$
3. if $a < b^k$, then
   (a) if p >= 0, then $T(n) = (n^k \log^p n)$
   (b) if p < 0, then $T(n) = (n^k)$

Time Complexity Analysis –

- **Example-1: Binary Search** – T(n) = T(n/2) + O(1)
  a = 1, b = 2, k = 0 and p = 0
  $b^k = 1$. So, $a = b^k$ and p > -1 [Case 2.(a)]
  $T(n) = (n^{\log_b a} \log^{p+1} n)$
  T(n) = (logn)
- **Example-2: Merge Sort** – T(n) = 2T(n/2) + O(n)
  a = 2, b = 2, k = 1, p = 0
  $b^k = 1$. So, $a = b^k$ and p > -1 [Case 2.(a)]
  $T(n) = (n^{\log_b a} \log^{p+1} n)$
  T(n) = (nlogn)

- **Example-3:** $T(n) = 3T(n/2) + n^2$
  $a = 3, b = 2, k = 2, p = 0$
  $b^k = 4$. So, $a < b^k$ and $p = 0$ [Case 3.(a)]
  $T(n) = \ (n^k \log^p n)$
  $T(n) = \ (n^2)$
- **Example-4:** $T(n) = 3T(n/2) + \log^2 n$
  $a = 3, b = 2, k = 0, p = 2$
  $b^k = 1$. So, $a > b^k$ [Case 1]
  $T(n) = \ (n^{\log_b a})$
  $T(n) = \ (n^{\log_2 3})$
- **Example-5:** $T(n) = 2T(n/2) + n\log^2 n$
  $a = 2, b = 2, k = 1, p = 2$
  $b^k = 1$. So, $a = b^k$ [Case 2.(a)]
  $T(n) = \ (n^{\log_b a}\log^{p+1} n)$
  $T(n) = \ (n^{\log_2 2}\log^3 n)$
  $T(n) = \ (n\log^3 n)$
- **Example-6:** $T(n) = 2^n T(n/2) + n^n$
  This recurrence can't be solved using above method since function is not of form $T(n) = aT(n/b) + \ (n^k \log^p n)$

GATE Practice questions –

- GATE-CS-2017 (Set 2) | Question 56
- GATE IT 2008 | Question 42
- GATE CS 2009 | Question 35

Source

https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/

Chapter 2

Allocate minimum number of pages

Allocate minimum number of pages - GeeksforGeeks

Given number of pages in n different books and m students. The books are arranged in ascending order of number of pages. Every student is assigned to read some consecutive books. The task is to assign books in such a way that the maximum number of pages assigned to a student is minimum.

Example :

```
Input : pages[] = {12, 34, 67, 90}
        m = 2
Output : 113
Explanation:
There are 2 number of students. Books can be distributed
in following fashion :
  1) [12] and [34, 67, 90]
      Max number of pages is allocated to student
      2 with 34 + 67 + 90 = 191 pages
  2) [12, 34] and [67, 90]
      Max number of pages is allocated to student
      2 with 67 + 90 = 157 pages
  3) [12, 34, 67] and [90]
      Max number of pages is allocated to student
      1 with 12 + 34 + 67 = 113 pages

Of the 3 cases, Option 3 has the minimum pages = 113.
```

The idea is to use Binary Search. We fix a value for the number of pages as mid of current minimum and maximum. We initialize minimum and maximum as 0 and sum-of-all-pages respectively. If a current mid can be a solution, then we search on the lower half, else we search in higher half.

Now the question arises, how to check if a mid value is feasible or not? Basically, we need to check if we can assign pages to all students in a way that the maximum number doesn't exceed current value. To do this, we sequentially assign pages to every student while the current number of assigned pages doesn't exceed the value. In this process, if the number of students becomes more than m, then the solution is not feasible. Else feasible.

Below is an implementation of above idea.

C++

```
 // C++ program for optimal allocation of pages
#include<bits/stdc++.h>
using namespace std;

// Utility function to check if current minimum value
// is feasible or not.
bool isPossible(int arr[], int n, int m, int curr_min)
{
    int studentsRequired = 1;
    int curr_sum = 0;

    // iterate over all books
    for (int i = 0; i < n; i++)
    {
        // check if current number of pages are greater
        // than curr_min that means we will get the result
        // after mid no. of pages
        if (arr[i] > curr_min)
            return false;

        // count how many students are required
        // to distribute curr_min pages
        if (curr_sum + arr[i] > curr_min)
        {
            // increment student count
            studentsRequired++;

            // update curr_sum
            curr_sum = arr[i];

            // if students required becomes greater
            // than given no. of students,return false
            if (studentsRequired > m)
                return false;
        }
        // else update curr_sum
        else
            curr_sum += arr[i];
    }
    return true;
}

// function to find minimum pages
int findPages(int arr[], int n, int m)
{
    long long sum = 0;

    // return -1 if no. of books is less than
    // no. of students
    if (n < m)
        return -1;

    // Count total number of pages
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // initialize start as 0 pages and end as
    // total pages
    int start = 0, end = sum;
    int result = INT_MAX;

    // traverse until start <= end
    while (start <= end)
    {
        // check if it is possible to distribute
        // books by using mid as current minimum
        int mid = (start + end) / 2;
        if (isPossible(arr, n, m, mid))
        {
            // if yes then find the minimum distribution
            result = min(result, mid);

            // as we are finding minimum and books
            // are sorted so reduce end = mid -1
            // that means
            end = mid - 1;
        }

        else
            // if not possible means pages should be
            // increased so update start = mid + 1
            start = mid + 1;
    }
    // at-last return minimum no. of  pages
    return result;
}

// Drivers code
int main()
{
    //Number of pages in books
    int arr[] = {12, 34, 67, 90};
    int n = sizeof arr / sizeof arr[0];
    int m = 2; //No. of students

    cout << "Minimum number of pages = "
         << findPages(arr, n, m) << endl;
    return 0;
}
```

Java

```
 // Java program for optimal allocation of pages

public class GFG
{
    // Utility method to check if current minimum value
    // is feasible or not.
    static boolean isPossible(int arr[], int n, int m, int curr_min)
    {
        int studentsRequired = 1;
        int curr_sum = 0;

        // iterate over all books
        for (int i = 0; i < n; i++)
        {
            // check if current number of pages are greater
            // than curr_min that means we will get the result
            // after mid no. of pages
            if (arr[i] > curr_min)
                return false;

            // count how many students are required
            // to distribute curr_min pages
            if (curr_sum + arr[i] > curr_min)
            {
                // increment student count
                studentsRequired++;

                // update curr_sum
                curr_sum = arr[i];

                // if students required becomes greater
                // than given no. of students,return false
                if (studentsRequired > m)
                    return false;
            }

            // else update curr_sum
            else
                curr_sum += arr[i];
        }
        return true;
    }

    // method to find minimum pages
    static int findPages(int arr[], int n, int m)
    {
        long sum = 0;

        // return -1 if no. of books is less than
        // no. of students
        if (n < m)
            return -1;

        // Count total number of pages
        for (int i = 0; i < n; i++)
            sum += arr[i];

        // initialize start as 0 pages and end as
        // total pages
        int start = 0, end = (int) sum;
        int result = Integer.MAX_VALUE;

        // traverse until start <= end
        while (start <= end)
        {
            // check if it is possible to distribute
            // books by using mid is current minimum
            int mid = (start + end) / 2;
            if (isPossible(arr, n, m, mid))
            {
                // if yes then find the minimum distribution
                result = Math.min(result, mid);

                // as we are finding minimum and books
                // are sorted so reduce end = mid -1
                // that means
                end = mid - 1;
            }

            else
                // if not possible means pages should be
                // increased so update start = mid + 1
                start = mid + 1;
        }

        // at-last return minimum no. of  pages
        return result;
    }

    // Driver Method
    public static void main(String[] args)
    {
        //Number of pages in books
        int arr[] = {12, 34, 67, 90};

        int m = 2; //No. of students

        System.out.println("Minimum number of pages = " +
                          findPages(arr, arr.length, m));
    }
}
```

C#

```
 // C# program for optimal
// allocation of pages
using System;

class GFG
{

// Utility function to check
// if current minimum value
// is feasible or not.
static bool isPossible(int []arr, int n,
                       int m, int curr_min)
{
    int studentsRequired = 1;
    int curr_sum = 0;

    // iterate over all books
    for (int i = 0; i < n; i++)
    {
        // check if current number of
        // pages are greater than curr_min
        // that means we will get the
        // result after mid no. of pages
        if (arr[i] > curr_min)
            return false;

        // count how many students
        // are required to distribute
        // curr_min pages
        if (curr_sum + arr[i] > curr_min)
        {
            // increment student count
            studentsRequired++;

            // update curr_sum
            curr_sum = arr[i];

            // if students required becomes
            // greater than given no. of
            // students, return false
            if (studentsRequired > m)
                return false;
        }

        // else update curr_sum
        else
            curr_sum += arr[i];
    }
    return true;
}

// function to find minimum pages
static int findPages(int []arr,
                     int n, int m)
{
    long sum = 0;

    // return -1 if no. of books
    // is less than no. of students
    if (n < m)
        return -1;

    // Count total number of pages
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // initialize start as 0 pages
    // and end as total pages
    int start = 0, end = (int)sum;
    int result = int.MaxValue;

    // traverse until start <= end
    while (start <= end)
    {
        // check if it is possible to
        // distribute books by using
        // mid as current minimum
        int mid = (start + end) / 2;
        if (isPossible(arr, n, m, mid))
        {
            // if yes then find the
            // minimum distribution
            result = Math.Min(result, mid);

            // as we are finding minimum and
            // books are sorted so reduce
            // end = mid -1 that means
            end = mid - 1;
        }

        else
            // if not possible means pages
            // should be increased so update
            // start = mid + 1
            start = mid + 1;
    }

    // at-last return
    // minimum no. of pages
    return result;
}

// Drivers code
static public void Main ()
{

//Number of pages in books
int []arr = {12, 34, 67, 90};
int n = arr.Length;
int m = 2; // No. of students

Console.WriteLine("Minimum number of pages = " +
                          findPages(arr, n, m));
}
}
```

```
// This code is contributed by anuj_67.
```

Output :

```
Minimum number of pages = 113
```

Improved By : vt_m

Source

https://www.geeksforgeeks.org/allocate-minimum-number-pages/

Chapter 3

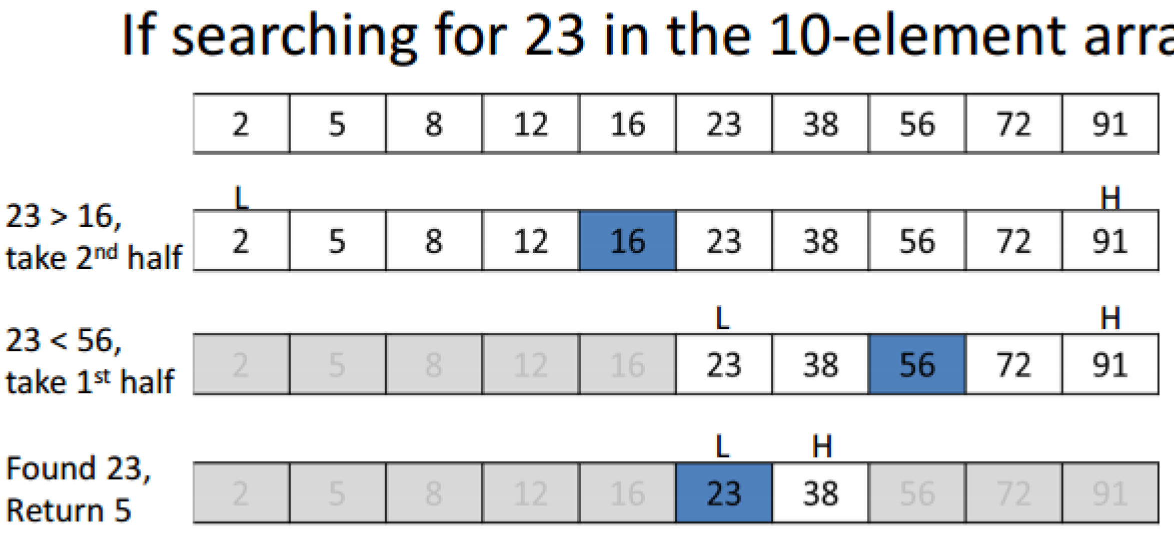
Binary Search

Binary Search - GeeksforGeeks

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Example :
Image Source : http://www.nyckidd.com/bob/Linear%20Search%20and%20Binary%20Search_WorkingCopy.pdf

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

C/C++

```
 // C program to implement recursive Binary Search
#include <stdio.h>

// 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;
}

int main(void)
```

```
{
   int arr[] = {2, 3, 4, 10, 40};
   int n = sizeof(arr)/ sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n-1, x);
   (result == -1)? printf("Element is not present in array")
                 : printf("Element is present at index %d",
                                                  result);
   return 0;
}
```

Java

```
 // Java implementation of recursive Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[l..
    // r], else return -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 method to test above
    public static void main(String args[])
    {
        BinarySearch ob = new BinarySearch();
        int arr[] = {2,3,4,10,40};
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr,0,n-1,x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index " +
                                                 result);
    }
}
/* This code is contributed by Rajat Mishra */
```

Python

```
 # Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l)/2

        # If 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
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid+1, r, x)

    else:
        # Element is not present in the array
        return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
```

```
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"
```

C#

```
 // C# implementation of recursive Binary Search
using System;

class GFG
{
    // Returns index of x if it is present in
    // arr[l..r], else return -1
    static 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 method to test above
    public static void Main()
    {

        int []arr = {2, 3, 4, 10, 40};
        int n = arr.Length;
        int x = 10;

        int result = binarySearch(arr, 0, n-1, x);

        if (result == -1)
            Console.WriteLine("Element not present");
        else
            Console.WriteLine("Element found at index "
                                            + result);
    }
}

// This code is contributed by Sam007.
```

PHP

```
 <?php
// PHP program to implement
// recursive Binary Search

// A recursive binary search
// function. It returns location
// of x in given array arr[l..r]
// is present, otherwise -1
function binarySearch($arr, $l, $r, $x)
{
if ($r >= $l)
{
        $mid = $l + ($r - $l) / 2;

        // If the element is present
        // at the middle itself
        if ($arr[$mid] == $x)
            return floor($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 Code
$arr = array(2, 3, 4, 10, 40);
$n = count($arr);
$x = 10;
$result = binarySearch($arr, 0, $n - 1, $x);
if(($result == -1))
echo "Element is not present in array";
else
echo "Element is present at index ",
                            $result;

// This code is contributed by anuj_67.
?>
```

Output :

```
Element is present at index 3
```

Iterative implementation of Binary Search

C/C++

```
 // C program to implement iterative Binary Search
#include <stdio.h>

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r-l)/2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;
        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present"
                                   " in array")
               : printf("Element is present at "
                                "index %d", result);
    return 0;
}
```

Java

```
 // Java implementation of iterative Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[],
    // else return -1
    int binarySearch(int arr[], int x)
    {
        int l = 0, r = arr.length - 1;
        while (l <= r)
        {
            int m = l + (r-l)/2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;

            // If x is smaller, ignore right half
            else
                r = m - 1;
        }

        // if we reach here, then element was
        // not present
        return -1;
    }

    // Driver method to test above
    public static void main(String args[])
    {
        BinarySearch ob = new BinarySearch();
        int arr[] = {2, 3, 4, 10, 40};
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at " +
                                   "index " + result);
    }
}
```

Python

```
 # Python code to implement iterative Binary
# Search.

# It returns location of x in given array arr
# if present, else returns -1
def binarySearch(arr, l, r, x):

    while l <= r:

        mid = l + (r - l)/2;

        # Check if x is present at mid
        if arr[mid] == x:
            return mid

        # If x is greater, ignore left half
        elif arr[mid] < x:
            l = mid + 1

        # If x is smaller, ignore right half
        else:
            r = mid - 1
    # If we reach here, then the element
    # was not present
    return -1


# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"
```

C#

```
 // C# implementation of iterative Binary Search
using System;

class GFG
{
    // Returns index of x if it is present in arr[],
    // else return -1
    static int binarySearch(int []arr, int x)
    {
        int l = 0, r = arr.Length - 1;
        while (l <= r)
        {
            int m = l + (r-l)/2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;

            // If x is smaller, ignore right half
            else
                r = m - 1;
        }

        // if we reach here, then element was
        // not present
        return -1;
    }

    // Driver method to test above
    public static void Main()
    {
        int []arr = {2, 3, 4, 10, 40};
        int n = arr.Length;
        int x = 10;
        int result = binarySearch(arr, x);
        if (result == -1)
            Console.WriteLine("Element not present");
        else
            Console.WriteLine("Element found at " +
                                "index " + result);
    }
}
// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP program to implement
// iterative Binary Search

// A iterative binary search
// function. It returns location
// of x in given array arr[l..r]
// if present, otherwise -1
function binarySearch($arr, $l,
                      $r, $x)
{
    while ($l <= $r)
    {
        $m = $l + ($r - $l) / 2;

        // Check if x is present at mid
        if ($arr[$m] == $x)
            return floor($m);

        // If x greater, ignore
        // left half
        if ($arr[$m] < $x)
            $l = $m + 1;

        // If x is smaller,
        // ignore right half
        else
```

```
            $r = $m - 1;
    }

    // if we reach here, then
    // element was not present
    return -1;
}

// Driver Code
$arr = array(2, 3, 4, 10, 40);
$n = count($arr);
$x = 10;
$result = binarySearch($arr, 0,
                       $n - 1, $x);
if(($result == -1))
echo "Element is not present in array";
else
echo "Element is present at index ",
                            $result;

// This code is contributed by anuj_67.
?>
```

Output :

```
Element is present at index 3
```

Time Complexity:
The time complexity of Binary Search can be written as

```
T(n) = T(n/2) + c
```

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is [illegible].

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Algorithmic Paradigm: Decrease and Conquer.

Interesting articles based on Binary Search.

- The Ubiquitous Binary Search
- Interpolation search vs Binary search
- Find the minimum element in a sorted and rotated array

- Find a peak element
- Find a Fixed Point in a given array
- Count the number of occurrences in a sorted array
- Median of two sorted arrays
- Floor and Ceiling in a sorted array
- Find the maximum element in an array which is first increasing and then decreasing

Coding Practice Questions on Binary Search
Recent Articles on Binary Search.

Improved By : vt_m

Source

https://www.geeksforgeeks.org/binary-search/

Chapter 4

Binary Search (bisect) in Python

Binary Search (bisect) in Python - GeeksforGeeks

Binary Search is a technique used to search element in a sorted list. In this article, we will looking at library functions to do Binary Search.

Finding first occurrence of an element.

> bisect.bisect_left(a, x, lo=0, hi=len(a)) : Returns leftmost insertion point of x in a sorted list. Last two parameters are optional, they are used to search in sublist.

```
 # Python code to demonstrate working
# of binary search in library
from bisect import bisect_left

def BinarySearch(a, x):
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    else:
        return -1

a  = [1, 2, 4, 4, 8]
x = int(4)
res = BinarySearch(a, x)
if res == -1:
    print(x, "is absent")
else:
    print("First occurrence of", x, "is present at", res)
```

Output:

```
First occurrence of 4 is present at 2
```

Finding greatest value smaller than x.

```
 # Python code to demonstrate working
# of binary search in library
from bisect import bisect_left

def BinarySearch(a, x):
    i = bisect_left(a, x)
    if i:
        return (i-1)
    else:
        return -1

# Driver code
a  = [1, 2, 4, 4, 8]
x = int(7)
res = BinarySearch(a, x)
if res == -1:
    print("No value smaller than ", x)
else:
    print("Largest value smaller than ", x, " is at index ", res)
```

Output:

```
Largest value smaller than  7  is at index  3
```

Finding rightmost occurrence

> bisect.bisect_right(a, x, lo=0, hi=len(a)) Returns rightmost insertion point of x in a sorted list a. Last two parameters are optional, they are used to search in sublist.

```
 # Python code to demonstrate working
# of binary search in library
from bisect import bisect_right

def BinarySearch(a, x):
    i = bisect_right(a, x)
    if i != len(a)+1 and a[i-1] == x:
        return (i-1)
    else:
        return -1

a  = [1, 2, 4, 4]
x = int(4)
res = BinarySearch(a, x)
if res == -1:
    print(x, "is absent")
else:
    print("Last occurrence of", x, "is present at", res)
```

Output:

```
Last occurrence of 4 is present at 3
```

Please refer Binary Search for writing your own Binary Search code.

Reference :
https://docs.python.org/3/library/bisect.html

Source

https://www.geeksforgeeks.org/binary-search-bisect-in-python/

Chapter 5

Binary Search on Singly Linked List

Binary Search on Singly Linked List - GeeksforGeeks

Given a singly linked list and a key, find key using binary search approach.
To perform a Binary search based on Divide and Conquer Algorithm, determination of the middle element is important. Binary Search is usually fast and efficient for arrays because accessing the middle index between two given indices is easy and fast(Time Complexity O(1)). But memory allocation for the singly linked list is dynamic and non-contiguous, which makes finding the middle element difficult. One approach could be of using skip list, one could be traversing the linked list using one pointer.

Prerequisite : Finding middle of a linked list.

Note: The approach and implementation provided below are to show how Binary Search can be implemented on a linked list. The implementation takes O(n) time.

Approach :

- Here, start node(set to Head of list), and the last node(set to NULL initially) are given.
- Middle is calculated using two pointers approach.
- If middle's data matches the required value of search, return it.
- Else if midele's data < value, move to upper half(setting last to midele's next).
- Else go to lower half(setting last to middle).
- The condition to come out is, either element found or entire list is traversed. When entire list is traversed, last points to start i.e. last -> next == start.

In main function, function **InsertAtHead** inserts value at the beginning of linked list. Inserting such values(for sake of simplicity) so that the list created is sorted.

Examples :

```
Input : Enter value to search : 7
Output : Found

Input : Enter value to search : 12
Output : Not Found
```

```
 // CPP code to implement binary search
// on Singly Linked List
#include<stdio.h>
#include<stdlib.h>

struct Node
{
    int data;
    struct Node* next;
};

Node *newNode(int x)
{
    struct Node* temp = new Node;
    temp->data = x;
    temp->next = NULL;
    return temp;
}

// function to find out middle element
struct Node* middle(Node* start, Node* last)
{
    if (start == NULL)
        return NULL;

    struct Node* slow = start;
    struct Node* fast = start -> next;

    while (fast != last)
    {
        fast = fast -> next;
        if (fast != last)
        {
            slow = slow -> next;
            fast = fast -> next;
        }
    }

    return slow;
}
```

```
// Function for implementing the Binary
// Search on linked list
struct Node* binarySearch(Node *head, int value)
{
    struct Node* start = head;
    struct Node* last = NULL;

    do
    {
        // Find middle
        Node* mid = middle(start, last);

        // If middle is empty
        if (mid == NULL)
            return NULL;

        // If value is present at middle
        if (mid -> data == value)
            return mid;

        // If value is more than mid
        else if (mid -> data < value)
            start = mid -> next;

        // If the value is less than mid.
        else
            last = mid;

    } while (last == NULL ||
             last -> next != start);

    // value not present
    return NULL;
}

// Driver Code
int main()
{
    Node *head = newNode(1);
    head->next = newNode(4);
    head->next->next = newNode(7);
    head->next->next->next = newNode(8);
    head->next->next->next->next = newNode(9);
    head->next->next->next->next->next = newNode(10);
    int value = 7;
    if (binarySearch(head, value) == NULL)
        printf("Value not present\n");
    else
```

```
        printf("Present");
    return 0;
}
```

Output:

```
Present
```

Time Complexity : O(n)

Improved By : sunny94

Source

https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/

Chapter 6

Binary Search using pthread

Binary Search using pthread - GeeksforGeeks

Binary search is a popular method of searching in a sorted array or list. It simply divides the list into two halves and discard the half which has zero probability of having the key. On dividing we check the mid point for the key and uses the lower half if key is less than mid point and upper half if key is greater than mid point. Binary search has time complexity of O(log(n)).

Binary search can also be implemented using **multi-threading** where we utilizes the cores of processor by providing each thread a portion of list to search for the key.

Number of threads depends upon the number of cores your processor has and its better to create one thread for each core.

Examples:

```
Input :  list = 1, 5, 7, 10, 12, 14, 15, 18, 20, 22, 25, 27, 30, 64, 110, 220
         key = 7
Output : 7 found in list

Input :  list = 1, 5, 7, 10, 12, 14, 15, 18, 20, 22, 25, 27, 30, 64, 110, 220
         key = 111
Output : 111 not found in list
```

Note – It is advised to execute the program in Linux based system.
Compile in linux using following code:

```
g++ -pthread program_name.cpp
```

```
 // CPP Program to perform binary search using pthreads
#include <iostream>
```

```
using namespace std;

// size of array
#define MAX 16

// maximum number of threads
#define MAX_THREAD 4

// array to be searched
int a[] = { 1, 5, 7, 10, 12, 14, 15, 18, 20, 22, 25, 27, 30, 64, 110, 220 };

// key that needs to be searched
int key = 110;
bool found = false;
int part = 0;

void* binary_search(void* arg)
{

    // Each thread checks 1/4 of the array for the key
    int thread_part = part++;
    int mid;

    int low = thread_part * (MAX / 4);
    int high = (thread_part + 1) * (MAX / 4);

    // search for the key until low < high
    // or key is found in any portion of array
    while (low < high && !found)  {

        // normal iterative binary search algorithm
        mid = (high - low) / 2 + low;

        if (a[mid] == key)  {
            found = true;
            break;
        }

        else if (a[mid] > key)
            high = mid - 1;
        else
            low = mid + 1;
    }
}


// Driver Code
int main()
```

```
{
    pthread_t threads[MAX_THREAD];

    for (int i = 0; i < MAX_THREAD; i++)
        pthread_create(&threads[i], NULL, binary_search, (void*)NULL);

    for (int i = 0; i < MAX_THREAD; i++)
        pthread_join(threads[i], NULL);

    // key found in array
    if (found)
        cout << key << " found in array" << endl;

    // key not found in array
    else
        cout << key << " not found in array" << endl;

    return 0;
}
```

Output:

```
110 found in array
```

Source

https://www.geeksforgeeks.org/binary-search-using-pthread/

Chapter 7

Check for Majority Element in a sorted array

Check for Majority Element in a sorted array - GeeksforGeeks

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

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

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

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

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

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

   return 0;
}
```

Java

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

class Majority {

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

        /* get last index according to n (even or odd) */
        last_index = (n%2==0)? n/2: n/2+1;
        /* search for first occurrence of x in arr[]*/
        for (i = 0; i < last_index; i++)
        {
            /* check if x is present and is present more
               than n/2 times */
            if (arr[i] == x && arr[i+n/2] == x)
                return true;
        }
        return false;
    }

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

Python

```
 '''Python3 Program to check for majority element in a sorted array'''

def isMajority(arr, n, x):
    # get last index according to n (even or odd) */
    last_index = (n//2 + 1) if n % 2 == 0 else (n//2)

    # search for first occurrence of x in arr[]*/
    for i in range(last_index):
        # check if x is present and is present more than n / 2 times */
        if arr[i] == x and arr[i + n//2] == x:
            return 1

# Driver program to check above function */
arr = [1, 2, 3, 4, 4, 4, 4]
n = len(arr)
x = 4
if (isMajority(arr, n, x)):
    print ("% d appears more than % d times in arr[]"
                                            %(x, n//2))
else:
    print ("% d does not appear more than % d times in arr[]"
                                                %(x, n//2))


# This code is contributed by shreyanshi_arun.
```

C#

```
 // C# Program to check for majority
// element in a sorted array
using System;

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

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

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

    // Driver code
    public static void Main()
    {
        int[] arr = { 1, 2, 3, 4, 4, 4, 4 };
        int n = arr.Length;
        int x = 4;
        if (isMajority(arr, n, x) == true)
            Console.Write(x + " appears more than " +
                            n / 2 + " times in arr[]");
        else
            Console.Write(x + " does not appear more than " +
                             n / 2 + " times in arr[]");
    }
}
```

```
// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP Program to check for
// majority element in a
// sorted array

// function returns majority
// element in a sorted array
function isMajority($arr, $n, $x)
{
    $i;

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

    // search for first occurrence
    // of x in arr[]
    for ($i = 0; $i < $last_index; $i++)
    {

        // check if x is present and
        // is present more than n/2
        // times
        if ($arr[$i] == $x && $arr[$i + $n / 2] == $x)
            return 1;
    }
    return 0;
}

    // Driver Code
    $arr = array(1, 2, 3, 4, 4, 4, 4);
    $n = sizeof($arr);
    $x = 4;
    if (isMajority($arr, $n, $x))
        echo $x, " appears more than "
             , floor($n / 2), " times in arr[]";

    else
        echo $x, "does not appear more than "
              , floor($n / 2), "times in arr[]";

// This code is contributed by Ajit
?>
```

Output :

```
4 appears more than 3 times in arr[]
```

Time Complexity : O(n)

METHOD 2 (Using Binary Search)
Use binary search methodology to find the first occurrence of the given number. The criteria for binary search is important here.

C

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

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

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

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

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

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

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

    return -1;
}

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

Java

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

class Majority {

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

            /* Check if arr[mid] is the first occurrence of x.
                arr[mid] is first occurrence if x is one of the following
                is true:
```

```
                (i)  mid == 0 and arr[mid] == x
                (ii) arr[mid-1] < x and arr[mid] == x
            */
            if ( (mid == 0 || x > arr[mid-1]) && (arr[mid] == x) )
                return mid;
            else if (x > arr[mid])
                return _binarySearch(arr, (mid + 1), high, x);
            else
                return _binarySearch(arr, low, (mid -1), x);
        }

        return -1;
    }


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

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

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

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

        int arr[] = {1, 2, 3, 3, 3, 3, 10};
        int n = arr.length;
        int x = 3;
        if (isMajority(arr, n, x)==true)
            System.out.println(x + " appears more than "+
                              n/2 + " times in arr[]");
        else
            System.out.println(x + " does not appear more than " +
                              n/2 + " times in arr[]");
    }
}
/*This code is contributed by Devesh Agrawal*/
```

Python

```
 '''Python3 Program to check for majority element in a sorted array'''

# This function returns true if the x is present more than n / 2
# times in arr[] of size n */
def isMajority(arr, n, x):

    # Find the index of first occurrence of x in arr[] */
    i = _binarySearch(arr, 0, n-1, x)

    # If element is not present at all, return false*/
    if i == -1:
        return False

    # check if the element is present more than n / 2 times */
    if ((i + n//2) <= (n -1)) and 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 */
def _binarySearch(arr, low, high, x):
    if high >= low:
        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 or x > arr[mid-1]) and (arr[mid] == x):
            return mid
        elif 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 */
arr = [1, 2, 3, 3, 3, 3, 10]
n = len(arr)
x = 3
if (isMajority(arr, n, x)):
    print ("% d appears more than % d times in arr[]"
                                            % (x, n//2))
else:
    print ("% d does not appear more than % d times in arr[]"
                                                    % (x, n//2))

# This code is contributed by shreyanshi_arun.
```

C#

```
 // C# Program to check for majority
// element in a sorted array */
using System;

class GFG {

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

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

            if ((mid == 0 || x > arr[mid - 1]) &&
                                 (arr[mid] == x))
                return mid;
            else if (x > arr[mid])
                return _binarySearch(arr, (mid + 1),
                                           high, x);
            else
                return _binarySearch(arr, low,
                                      (mid - 1), x);
        }

        return -1;
    }

    // This function returns true if the x is
    // present more than n/2 times in arr[]
    // of size n
    static bool isMajority(int[] arr, int n, int x)
    {

        // Find the index of first occurrence
        // of x in arr[]
        int i = _binarySearch(arr, 0, n - 1, x);

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

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

    //Driver code
    public static void Main()
    {

        int[] arr = { 1, 2, 3, 3, 3, 3, 10 };
        int n = arr.Length;
        int x = 3;
        if (isMajority(arr, n, x) == true)
           Console.Write(x + " appears more than " +
                             n / 2 + " times in arr[]");
        else
           Console.Write(x + " does not appear more than " +
                             n / 2 + " times in arr[]");
    }
}

// This code is contributed by Sam007
```

Output:

```
3 appears more than 3 times in arr[]
```

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

Improved By : jit_t

Source

https://www.geeksforgeeks.org/check-for-majority-element-in-a-sorted-array/

Chapter 8

Closest Pair of Points using Divide and Conquer algorithm

Closest Pair of Points using Divide and Conquer algorithm - GeeksforGeeks

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy. In this post, a O(n x (Logn)^2) approach is discussed. We will be discussing a O(nLogn) approach in a separate post.

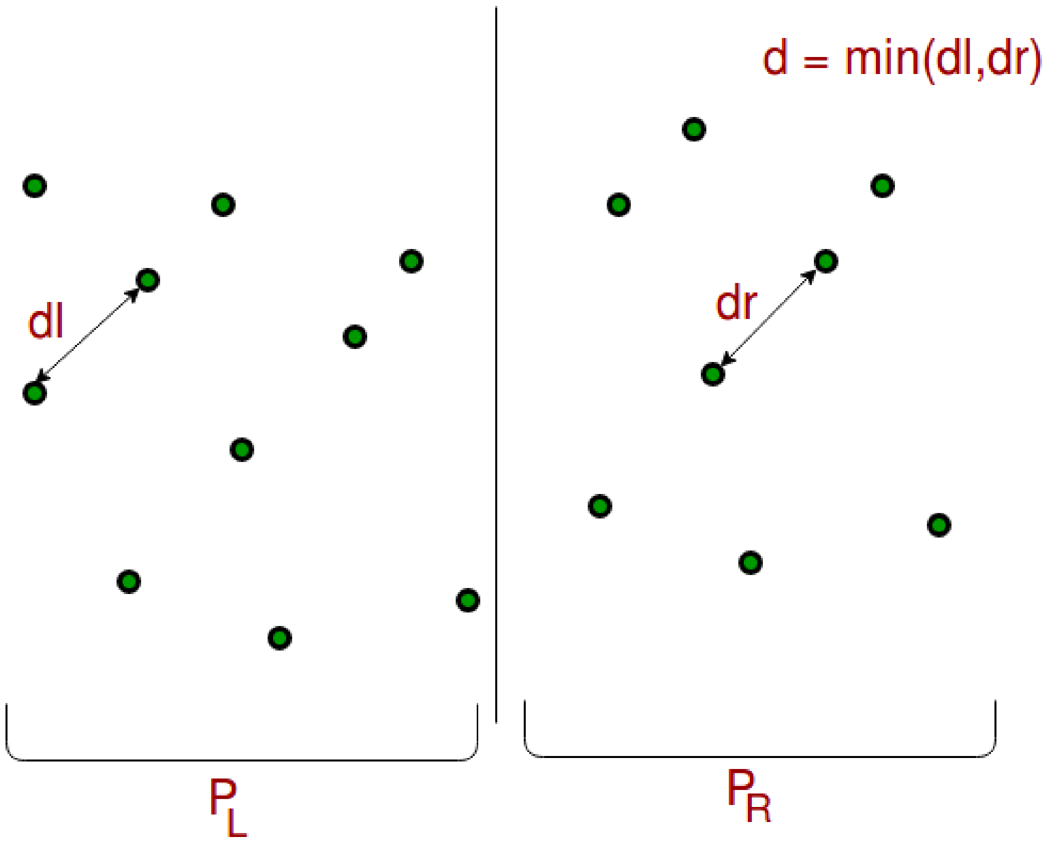
Algorithm
Following are the detailed steps of a O(n (Logn)^2) algortihm.
Input: An array of n points *P[]*
Output: The smallest distance between two points in the given array.

As a pre-processing step, input array is sorted according to x coordinates.

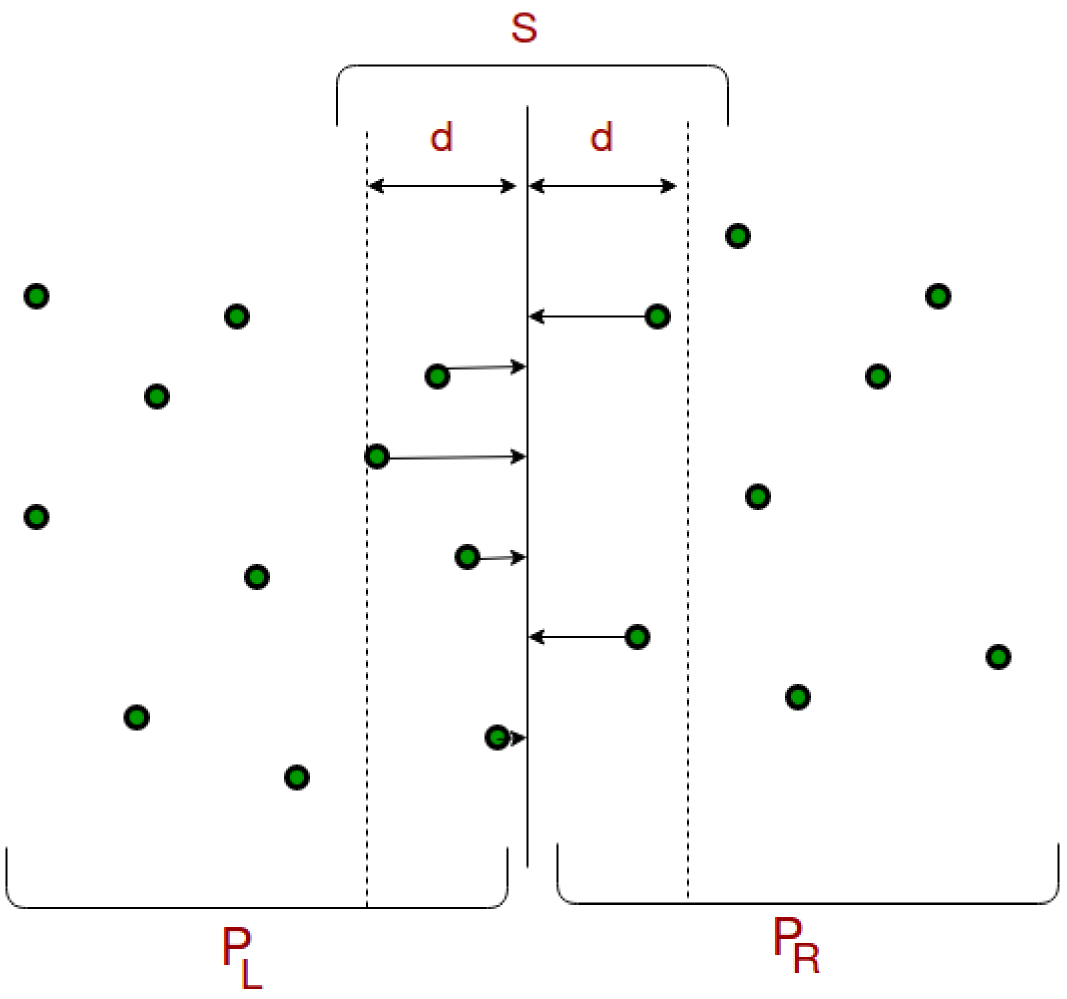
1) Find the middle point in the sorted array, we can take *P[n/2]* as middle point.

2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].

3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.



4) From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.



5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.

6) Find the smallest distance in strip[]. This is tricky. From first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See thisfor more analysis.

7) Finally return the minimum of d and distance calculated in above step (step 6)

Implementation
Following is C/C++ implementation of the above algorithm.

```
 // A divide and conquer program in C/C++ to find the smallest distance from a
// given set of points.

#include <stdio.h>
```

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

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};

/* Following two functions are needed for library function qsort().
   Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a,  *p2 = (Point *)b;
    return (p1->x - p2->x);
}
// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a,   *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
               );
}

// A Brute Force method to return the smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values
float min(float x, float y)
```

```
{
    return (x < y)? x : y;
}


// A utility function to find the distance beween the closest points of
// strip of given size. All points in strip[] are sorted accordint to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d;  // Initialize the minimum distance as d

    qsort(strip, size, sizeof(Point), compareY);

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array P contains
// all points sorted according to x coordinate
float closestUtil(Point P[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = P[mid];

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);
    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    // Find the closest points in strip.  Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main functin that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(P, n);
}

// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    printf("The smallest distance is %f ", closest(P, n));
    return 0;
}
```

Output:

```
The smallest distance is 1.414214
```

Time Complexity Let Time complexity of above algorithm be T(n). Let us assume that we use a O(nLogn) sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows
T(n) = 2T(n/2) + O(n) + O(nLogn) + O(n)
T(n) = 2T(n/2) + O(nLogn)
T(n) = T(n x Logn x Logn)

Notes
1) Time complexity can be improved to O(nLogn) by optimizing step 5 of the above algorithm. We will soon be discussing the optimized solution in a separate post.

2) The code finds smallest distance. It can be easily modified to find the points with smallest distance.
3) The code uses quick sort which can be O(n^2) in worst case. To have the upper bound as O(n (Logn)^2), a O(nLogn) sorting algorithm like merge sort or heap sort can be used

References:
http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf
http://www.youtube.com/watch?v=vS4Zn1a9KUc
http://www.youtube.com/watch?v=T3T7T8Ym20M
http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

Source

https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/

Chapter 9

Closest Pair of Points | O(nlogn) Implementation

Closest Pair of Points | O(nlogn) Implementation - GeeksforGeeks

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$||pq|| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

We have discussed a divide and conquer solution for this problem. The time complexity of the implementation provided in the previous post is O(n (Logn)^2). In this post, we discuss an implementation with time complexity as O(nLogn).

Following is a recap of the algorithm discussed in the previous post.

1) We sort all points according to x coordinates.

2) Divide all points in two halves.

3) Recursively find the smallest distances in both subarrays.

4) Take the minimum of two smallest distances. Let the minimum be d.

5) Create an array strip[] that stores all points which are at most d distance away from the middle line dividing the two sets.

6) Find the smallest distance in strip[].

7) Return the minimum of d and the smallest distance calculated in above step 6.

The great thing about the above approach is, if the array strip[] is sorted according to y coordinate, then we can find the smallest distance in strip[] in O(n) time. In the implementation discussed in previous post, strip[] was explicitly sorted in every recursive call that made the time complexity O(n (Logn)^2), assuming that the sorting step takes O(nLogn)

time.
In this post, we discuss an implementation where the time complexity is O(nLogn). The idea is to presort all points according to y coordinates. Let the sorted array be Py[]. When we make recursive calls, we need to divide points of Py[] also according to the vertical line. We can do that by simply processing every point and comparing its x coordinate with x coordinate of middle line.

Following is C++ implementation of O(nLogn) approach.

```
 // A divide and conquer program in C++ to find the smallest distance from a
// given set of points.

#include <iostream>
#include <float.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};


/* Following two functions are needed for library function qsort().
   Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a,  *p2 = (Point *)b;
    return (p1->x - p2->x);
}
// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a,   *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
               );
}
```

```
// A Brute Force method to return the smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}


// A utility function to find the distance beween the closest points of
// strip of given size. All points in strip[] are sorted accordint to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d;  // Initialize the minimum distance as d

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array Px contains
// all points sorted according to x coordinates and Py contains all points
// sorted according to y coordinates
float closestUtil(Point Px[], Point Py[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(Px, n);
    // Find the middle point
    int mid = n/2;
    Point midPoint = Px[mid];


    // Divide points in y sorted array around the vertical line.
    // Assumption: All x coordinates are distinct.
    Point Pyl[mid+1];   // y sorted points on left of vertical line
    Point Pyr[n-mid-1];  // y sorted points on right of vertical line
    int li = 0, ri = 0;  // indexes of left and right subarrays
    for (int i = 0; i < n; i++)
    {
      if (Py[i].x <= midPoint.x)
         Pyl[li++] = Py[i];
      else
         Pyr[ri++] = Py[i];
    }

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(Px, Pyl, mid);
    float dr = closestUtil(Px + mid, Pyr, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j] = Py[i], j++;

    // Find the closest points in strip.  Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main functin that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }

    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(Px, Py, n);
}

// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}
```

Output:

```
The smallest distance is 1.41421
```

Time Complexity: Let Time complexity of above algorithm be T(n). Let us assume that we use a O(nLogn) sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time. Also, it takes O(n) time to divide the Py array around the mid vertical line. Finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows
T(n) = 2T(n/2) + O(n) + O(n) + O(n)
T(n) = 2T(n/2) + O(n)
T(n) = T(nLogn)

References:
http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf
http://www.youtube.com/watch?v=vS4Zn1a9KUc
http://www.youtube.com/watch?v=T3T7T8Ym20M
http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

Source

https://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/

Chapter 10

Collect all coins in minimum number of steps

Collect all coins in minimum number of steps - GeeksforGeeks

Given many stacks of coins which are arranged adjacently. We need to collect all these coins in the minimum number of steps where in one step we can collect one horizontal line of coins or vertical line of coins and collected coins should be continuous.

Examples :

```
Input : height[] = [2 1 2 5 1]
Each value of this array corresponds to
the height of stack that is we are given
five stack of coins, where in first stack
2 coins are there then in second stack
1 coin is there and so on.
Output : 4
We can collect all above coins in 4 steps
which are shown in below diagram.
Each step is shown by different color.



First, we have collected last horizontal
line of coins after which stacks remains
as [1 0 1 4 0] after that, another horizontal
line of coins is collected from stack 3
and 4 then a vertical line from stack 4
and at the end a horizontal line from
stack 1. Total steps are 4.
```

We can solve this problem using divide and conquer method. We can see that it is always beneficial to remove horizontal lines from below. Suppose we are working on stacks from l index to r index in a recursion step, each time we will choose minimum height, remove those many horizontal lines after which stack will be broken into two parts, l to minimum and minimum +1 till r and we will call recursively in those subarrays. Another thing is we can also collect coins using vertical lines so we will choose minimum between the result of recursive calls and (r – l) because using (r – l) vertical lines we can always collect all coins. As each time we are calling each subarray and finding minimum of that, total time complexity of the solution will be O(N^2)

C++

```
 // C++ program to find minimum number of
// steps to collect stack of coins
#include <bits/stdc++.h>
using namespace std;

// recursive method to collect coins from
// height array l to r, with height h already
// collected
int minStepsRecur(int height[], int l, int r, int h)
{
    // if l is more than r, no steps needed
    if (l >= r)
        return 0;

    // loop over heights to get minimum height
    // index
    int m = l;
    for (int i = l; i < r; i++)
        if (height[i] < height[m])
            m = i;

    /* choose minimum from,
        1) collecting coins using all vertical
        lines (total r - l)
        2) collecting coins using lower horizontal
        lines and recursively on left and right
        segments */
    return min(r - l,
               minStepsRecur(height, l, m, height[m]) +
               minStepsRecur(height, m + 1, r, height[m]) +
               height[m] - h);
}

// method returns minimum number of step to
// collect coin from stack, with height in
// height[] array
```

```
int minSteps(int height[], int N)
{
    return minStepsRecur(height, 0, N, 0);
}

// Driver code to test above methods
int main()
{
    int height[] = { 2, 1, 2, 5, 1 };
    int N = sizeof(height) / sizeof(int);

    cout << minSteps(height, N) << endl;
    return 0;
}
```

Java

```
 // Java Code to Collect all coins in
// minimum number of steps
import java.util.*;

class GFG {

    // recursive method to collect coins from
    // height array l to r, with height h already
    // collected
    public static int minStepsRecur(int height[], int l,
                                          int r, int h)
    {
        // if l is more than r, no steps needed
        if (l >= r)
            return 0;

        // loop over heights to get minimum height
        // index
        int m = l;
        for (int i = l; i < r; i++)
            if (height[i] < height[m])
                m = i;

        /* choose minimum from,
            1) collecting coins using all vertical
            lines (total r - l)
            2) collecting coins using lower horizontal
            lines and recursively on left and right
            segments */
        return Math.min(r - l,
                        minStepsRecur(height, l, m, height[m]) +
                        minStepsRecur(height, m + 1, r, height[m]) +
                        height[m] - h);
    }

    // method returns minimum number of step to
    // collect coin from stack, with height in
    // height[] array
    public static int minSteps(int height[], int N)
    {
        return minStepsRecur(height, 0, N, 0);
    }

    /* Driver program to test above function */
    public static void main(String[] args)
    {

        int height[] = { 2, 1, 2, 5, 1 };
        int N = height.length;

        System.out.println(minSteps(height, N));
    }
}

// This code is contributed by Arnav Kr. Mandal.
```

Python 3

```
 # Python 3 program to find
# minimum number of steps
# to collect stack of coins

# recursive method to collect
# coins from height array l to
# r, with height h already
# collected
def minStepsRecur(height, l, r, h):

    # if l is more than r,
    # no steps needed
    if l >= r:
        return 0;

    # loop over heights to
    # get minimum height index
    m = l
    for i in range(l, r):
        if height[i] < height[m]:
            m = i
    # choose minimum from,
    # 1) collecting coins using
    # all vertical lines (total r - l)
    # 2) collecting coins using
    # lower horizontal lines and
    # recursively on left and
    # right segments
    return min(r - l,
            minStepsRecur(height, l, m, height[m]) +
            minStepsRecur(height, m + 1, r, height[m]) +
            height[m] - h)

# method returns minimum number
# of step to collect coin from
# stack, with height in height[] array
def minSteps(height, N):
    return minStepsRecur(height, 0, N, 0)

# Driver code
height = [ 2, 1, 2, 5, 1 ]
N = len(height)
print(minSteps(height, N))

# This code is contributed
# by ChitraNayal
```

C#

```
 // C# Code to Collect all coins in
// minimum number of steps
using System;

class GFG {

    // recursive method to collect coins from
    // height array l to r, with height h already
    // collected
    public static int minStepsRecur(int[] height, int l,
                                       int r, int h)
    {
        // if l is more than r, no steps needed
        if (l >= r)
            return 0;

        // loop over heights to
        // get minimum height index
        int m = l;
        for (int i = l; i < r; i++)
            if (height[i] < height[m])
                m = i;

        /* choose minimum from,
            1) collecting coins using all vertical
            lines (total r - l)
            2) collecting coins using lower horizontal
            lines and recursively on left and right
            segments */
        return Math.Min(r - l,
                        minStepsRecur(height, l, m, height[m]) +
                        minStepsRecur(height, m + 1, r, height[m]) +
                        height[m] - h);
    }

    // method returns minimum number of step to
    // collect coin from stack, with height in
    // height[] array
    public static int minSteps(int[] height, int N)
    {
        return minStepsRecur(height, 0, N, 0);
    }

    /* Driver program to test above function */
    public static void Main()
    {
        int[] height = { 2, 1, 2, 5, 1 };
        int N = height.Length;

        Console.Write(minSteps(height, N));
    }
}

// This code is contributed by nitin mittal
```

PHP

```
 <?php
// PHP program to find minimum number of
// steps to collect stack of coins

// recursive method to collect
// coins from height array l to
// r, with height h already
// collected
function minStepsRecur($height, $l,
                            $r, $h)
```

```
{

    // if l is more than r,
    // no steps needed
    if ($l >= $r)
        return 0;

    // loop over heights to
    // get minimum height
    // index
    $m = $l;
    for ($i = $l; $i < $r; $i++)
        if ($height[$i] < $height[$m])
            $m = $i;

    /* choose minimum from,
        1) collecting coins using
           all vertical lines
           (total r - l)
        2) collecting coins using
           lower horizontal lines
           and recursively on left
           and right segments */
    return min($r - $l,
           minStepsRecur($height, $l, $m, $height[$m]) +
           minStepsRecur($height, $m + 1, $r, $height[$m]) +
           $height[$m] - $h);
}

// method returns minimum number of step to
// collect coin from stack, with height in
// height[] array
function minSteps($height, $N)
{
    return minStepsRecur($height, 0, $N, 0);
}

    // Driver Code
    $height = array(2, 1, 2, 5, 1);
    $N = sizeof($height);

    echo minSteps($height, $N) ;

// This code is contributed by nitin mittal.
?>
```

Output:

```
4
```

Improved By : nitin mittal, ChitraNayal

Source

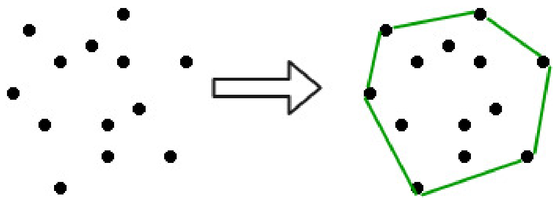
https://www.geeksforgeeks.org/collect-coins-minimum-number-steps/

Chapter 11

Convex Hull using Divide and Conquer Algorithm

Convex Hull using Divide and Conquer Algorithm - GeeksforGeeks

A convex hull is the smallest convex polygon containing all the given points.



Input is an array of points specified by their x and y coordinates. The output is the convex hull of this set of points.

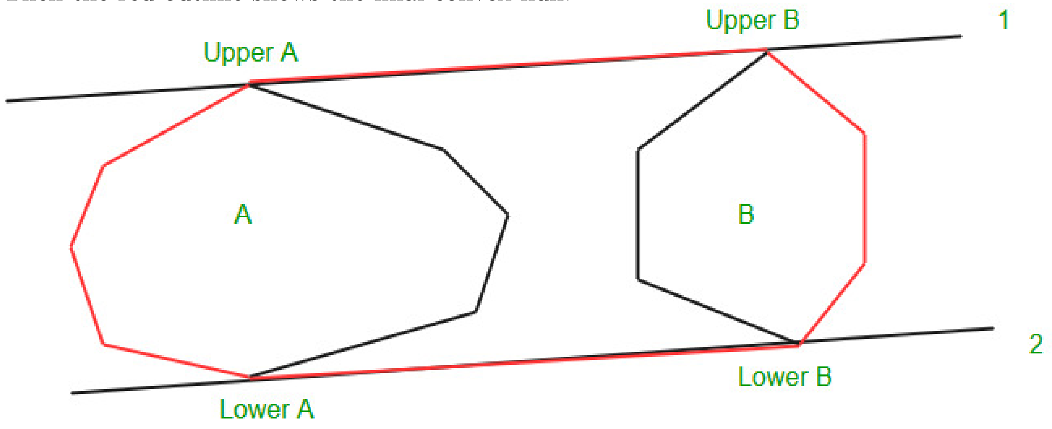
Examples:

```
Input : points[] = {(0, 0), (0, 4), (-4, 0), (5, 0),
                    (0, -6), (1, 0)};
Output : (-4, 0), (5, 0), (0, -6), (0, 4)
```

Pre-requisite:
Tangents between two convex polygons

Algorithm:
Given the set of points for which we have to find the convex hull. Suppose we know the convex hull of the left half points and the right half points, then the problem now is to merge these two convex hulls and determine the convex hull for the complete set.
This can be done by finding the upper and lower tangent to the right and left convex hulls. This is illustrated here Tangents between two convex polygons

Let the left convex hull be a and the right convex hull be b. Then the lower and upper tangents are named as 1 and 2 respectively, as shown in the figure.
Then the red outline shows the final convex hull.



Now the problem remains, how to find the convex hull for the left and right half. Now recursion comes into the picture, we divide the set of points until the number of points in the set is very small, say 5, and we can find the convex hull for these points by the brute algorithm. The merging of these halves would result in the convex hull for the complete set of points.

Note:
We have used the brute algorithm to find the convex hull for a small number of points and it has a time complexity of [illegible]. But some people suggest the following, the convex hull for 3 or fewer points is the complete set of points. This is correct but the problem comes when we try to merge a left convex hull of 2 points and right convex hull of 3 points, then the program gets trapped in an infinite loop in some special cases. So, to get rid of this problem I directly found the convex hull for 5 or fewer points by [illegible] algorithm, which is somewhat greater but does not affect the overall complexity of the algorithm.

```
 // A divide and conquer program to find convex
// hull of a given set of points.
#include<bits/stdc++.h>
using namespace std;

// stores the centre of polygon (It is made
// global because it is used in compare function)
pair<int, int> mid;

// determines the quadrant of a point
// (used in compare())
int quad(pair<int, int> p)
{
    if (p.first >= 0 && p.second >= 0)
        return 1;
    if (p.first <= 0 && p.second >= 0)
        return 2;
    if (p.first <= 0 && p.second <= 0)
        return 3;
    return 4;
}

// Checks whether the line is crossing the polygon
int orientation(pair<int, int> a, pair<int, int> b,
                pair<int, int> c)
{
    int res = (b.second-a.second)*(c.first-b.first) -
              (c.second-b.second)*(b.first-a.first);

    if (res == 0)
        return 0;
    if (res > 0)
        return 1;
    return -1;
}

// compare function for sorting
bool compare(pair<int, int> p1, pair<int, int> q1)
{
    pair<int, int> p = make_pair(p1.first - mid.first,
                                 p1.second - mid.second);
    pair<int, int> q = make_pair(q1.first - mid.first,
                                 q1.second - mid.second);

    int one = quad(p);
    int two = quad(q);

    if (one != two)
        return (one < two);
    return (p.second*q.first < q.second*p.first);
}

// Finds upper tangent of two polygons 'a' and 'b'
// represented as two vectors.
vector<pair<int, int>> merger(vector<pair<int, int> > a,
                              vector<pair<int, int> > b)
{
    // n1 -> number of points in polygon a
    // n2 -> number of points in polygon b
    int n1 = a.size(), n2 = b.size();

    int ia = 0, ib = 0;
    for (int i=1; i<n1; i++)
```

```
        if (a[i].first > a[ia].first)
            ia = i;

    // ib -> leftmost point of b
    for (int i=1; i<n2; i++)
        if (b[i].first < b[ib].first)
            ib=i;

    // finding the upper tangent
    int inda = ia, indb = ib;
    bool done = 0;
    while (!done)
    {
        done = 1;
        while (orientation(b[indb], a[inda], a[(inda+1)%n1]) >=0)
            inda = (inda + 1) % n1;

        while (orientation(a[inda], b[indb], b[(n2+indb-1)%n2]) <=0)
        {
            indb = (n2+indb-1)%n2;
            done = 0;
        }
    }

    int uppera = inda, upperb = indb;
    inda = ia, indb=ib;
    done = 0;
    int g = 0;
    while (!done)//finding the lower tangent
    {
        done = 1;
        while (orientation(a[inda], b[indb], b[(indb+1)%n2])>=0)
            indb=(indb+1)%n2;

        while (orientation(b[indb], a[inda], a[(n1+inda-1)%n1])<=0)
        {
            inda=(n1+inda-1)%n1;
            done=0;
        }
    }

    int lowera = inda, lowerb = indb;
    vector<pair<int, int>> ret;

    //ret contains the convex hull after merging the two convex hulls
    //with the points sorted in anti-clockwise order
    int ind = uppera;
    ret.push_back(a[uppera]);
    while (ind != lowera)
    {
        ind = (ind+1)%n1;
        ret.push_back(a[ind]);
    }

    ind = lowerb;
    ret.push_back(b[lowerb]);
    while (ind != upperb)
    {
        ind = (ind+1)%n2;
        ret.push_back(b[ind]);
    }
    return ret;

}

// Brute force algorithm to find convex hull for a set
// of less than 6 points
vector<pair<int, int>> bruteHull(vector<pair<int, int>> a)
{
    // Take any pair of points from the set and check
    // whether it is the edge of the convex hull or not.
    // if all the remaining points are on the same side
    // of the line then the line is the edge of convex
    // hull otherwise not
    set<pair<int, int> >s;

    for (int i=0; i<a.size(); i++)
    {
        for (int j=i+1; j<a.size(); j++)
        {
            int x1 = a[i].first, x2 = a[j].first;
            int y1 = a[i].second, y2 = a[j].second;

            int a1 = y1-y2;
            int b1 = x2-x1;
            int c1 = x1*y2-y1*x2;
            int pos = 0, neg = 0;
            for (int k=0; k<a.size(); k++)
            {
                if (a1*a[k].first+b1*a[k].second+c1 <= 0)
                    neg++;
                if (a1*a[k].first+b1*a[k].second+c1 >= 0)
                    pos++;
            }
            if (pos == a.size() || neg == a.size())
            {
                s.insert(a[i]);
                s.insert(a[j]);
            }
        }
    }

    vector<pair<int, int>>ret;
    for (auto e:s)
        ret.push_back(e);

    // Sorting the points in the anti-clockwise order
    mid = {0, 0};
    int n = ret.size();
    for (int i=0; i<n; i++)
    {
        mid.first += ret[i].first;
        mid.second += ret[i].second;
        ret[i].first *= n;
        ret[i].second *= n;
    }
    sort(ret.begin(), ret.end(), compare);
    for (int i=0; i<n; i++)
        ret[i] = make_pair(ret[i].first/n, ret[i].second/n);

    return ret;
}

// Returns the convex hull for the given set of points
vector<pair<int, int>> divide(vector<pair<int, int>> a)
{
    // If the number of points is less than 6 then the
    // function uses the brute algorithm to find the
    // convex hull
    if (a.size() <= 5)
        return bruteHull(a);

    // left contains the left half points
    // right contains the right half points
    vector<pair<int, int>>left, right;
    for (int i=0; i<a.size()/2; i++)
        left.push_back(a[i]);
    for (int i=a.size()/2; i<a.size(); i++)
        right.push_back(a[i]);

    // convex hull for the left and right sets
    vector<pair<int, int>>left_hull = divide(left);
    vector<pair<int, int>>right_hull = divide(right);
    // merging the convex hulls
    return merger(left_hull, right_hull);
}

// Driver code
int main()
{
    vector<pair<int, int> > a;
    a.push_back(make_pair(0, 0));
    a.push_back(make_pair(1, -4));
    a.push_back(make_pair(-1, -5));
    a.push_back(make_pair(-5, -3));
    a.push_back(make_pair(-3, -1));
    a.push_back(make_pair(-1, -3));
    a.push_back(make_pair(-2, -2));
    a.push_back(make_pair(-1, -1));
    a.push_back(make_pair(-2, -1));
    a.push_back(make_pair(-1, 1));

    int n = a.size();

    // sorting the set of points according
    // to the x-coordinate
    sort(a.begin(), a.end());
    vector<pair<int, int> >ans = divide(a);

    cout << "convex hull:\n";
    for (auto e:ans)
       cout << e.first << " "
            << e.second << endl;

    return 0;
}
```

Output:

```
Convex Hull:
-5 -3
-1 -5
1 -4
0 0
-1 1
```

Time Complexity: The merging of the left and the right convex hulls take O(n) time and as we are dividing the points into two equal parts, so the time complexity of the above algorithm is O(n * log n).

Related Articles :

- Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping)
- Convex Hull | Set 2 (Graham Scan)
- Quickhull Algorithm for Convex Hull

Source

https://www.geeksforgeeks.org/convex-hull-using-divide-and-conquer-algorithm/

Chapter 12

Count Inversions in an array | Set 1 (Using Merge Sort)

Count Inversions in an array | Set 1 (Using Merge Sort) - GeeksforGeeks

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.
Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j

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

METHOD 1 (Simple)
For each element, count number of elements which are on right side of it and are smaller than it.

C

```
 #include <bits/stdc++.h>
int getInvCount(int arr[], int n)
{
  int inv_count = 0;
  for (int i = 0; i < n - 1; i++)
    for (int 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};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf(" Number of inversions are %d \n", getInvCount(arr, n));
  return 0;
}
```

Java

```
 // Java program to count
// inversions in an array
class Test
{
    static int arr[] = new int[]{1, 20, 6, 4, 5};

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

      return inv_count;
    }

    // Driver method to test the above function
    public static void main(String[] args)
    {
        System.out.println("Number of inversions are "
                           + getInvCount(arr.length));

    }
}
```

Python3

```
 # Python3 program to count
# inversions in an array

def getInvCount(arr, n):

    inv_count = 0
    for i in range(n):
        for j in range(i+1, n):
            if (arr[i] > arr[j]):
                inv_count += 1
    return inv_count

# Driver Code
arr = [1, 20, 6, 4, 5]
n = len(arr)
print("Number of inversions are",
              getInvCount(arr, n))

# This code is contributed by Smitha Dinesh Semwal
```

C#

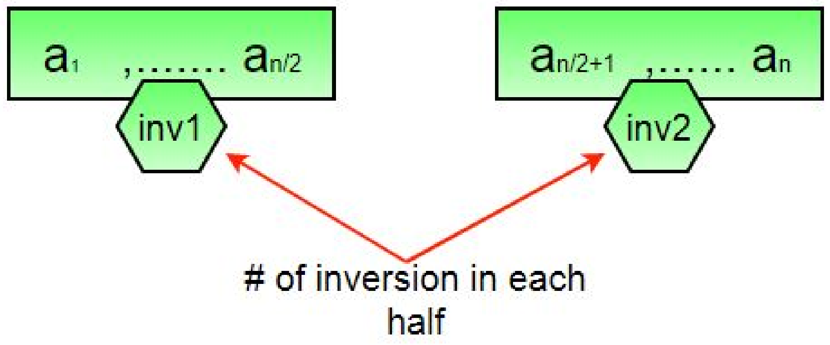
```
 // C# program to count inversions
// in an array
using System;
using System.Collections.Generic;

class GFG {

    static int []arr =
         new int[]{1, 20, 6, 4, 5};

    static int getInvCount(int n)
    {
        int inv_count = 0;

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

        return inv_count;
    }

    // Driver code
    public static void Main()
    {
        Console.WriteLine("Number of "
                   + "inversions are "
            + getInvCount(arr.Length));
    }
}

// This code is contributed by Sam007
```

Output:

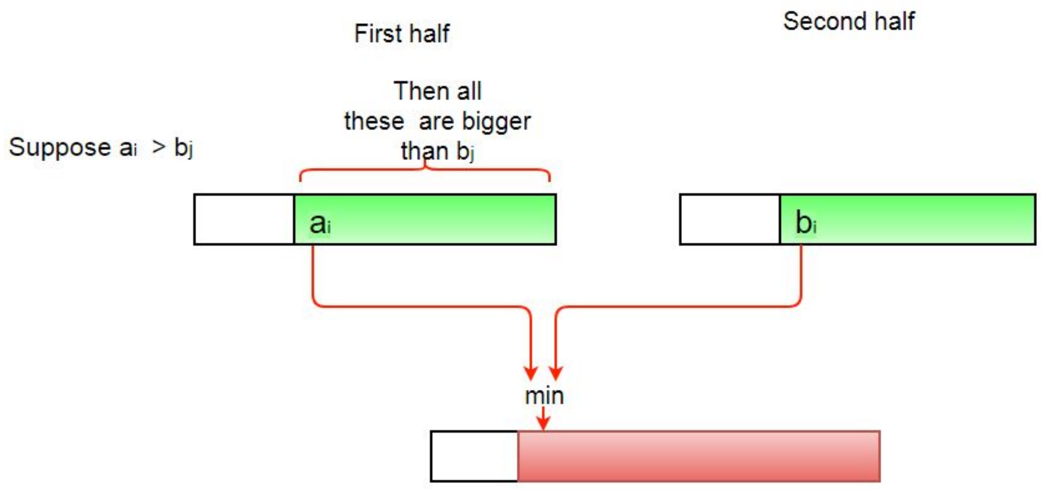
```
Number of inversions are 5
```

Time Complexity: O(n^2)

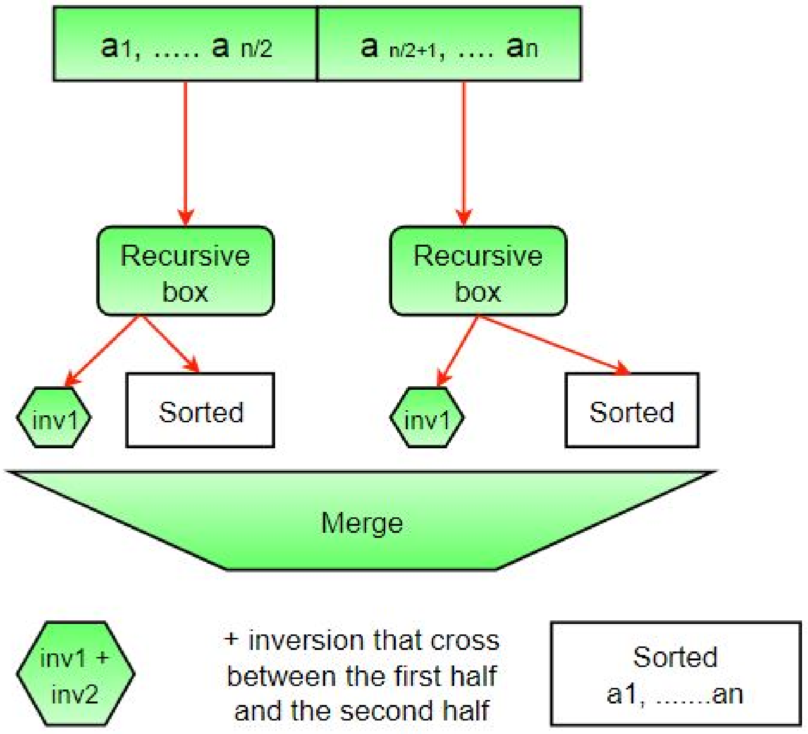
METHOD 2(Enhance Merge Sort)
Suppose we know the number of inversions in the left half and right half of the array (let be inv1 and inv2), what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().



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



The complete picture:



Implementation:

C

```
 #include <bits/stdc++.h>

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

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

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

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

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

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

  i = left; /* i is index for left subarray*/
  j = mid;  /* j is index for right subarray*/
  k = left; /* k 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 program 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;
}
```

Java

```
 // Java implementation of counting the
// inversion using merge sort

class Test
{

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

    /* An auxiliary recursive method that sorts the input array and
      returns the number of inversions in the array. */
    static int _mergeSort(int arr[], int temp[], int left, int right)
    {
      int mid, inv_count = 0;
      if (right > left)
      {
        /* Divide the array into two parts and call _mergeSortAndCountInv()
           for each of the parts */
        mid = (right + left)/2;
        /* Inversion count will be sum of inversions in left-part, right-part
          and number of inversions in merging */
        inv_count  = _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid+1, right);

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

    /* This method merges two sorted arrays and returns inversion count in
       the arrays.*/
    static 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;  /* j is index for right subarray*/
      k = left; /* k 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 method to test the above function
    public static void main(String[] args)
    {
        int arr[] = new int[]{1, 20, 6, 4, 5};
        System.out.println("Number of inversions are " + mergeSort(arr, 5));

    }
}
```

Output:

```
Number of inversions are 5
```

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

You may like to see.
Count inversions in an array | Set 2 (Using Self-Balancing BST)
Counting Inversions using Set in C++ STL
Count inversions in an array | Set 3 (Using BIT)

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

Improved By : Sam007

Source

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

Chapter 13

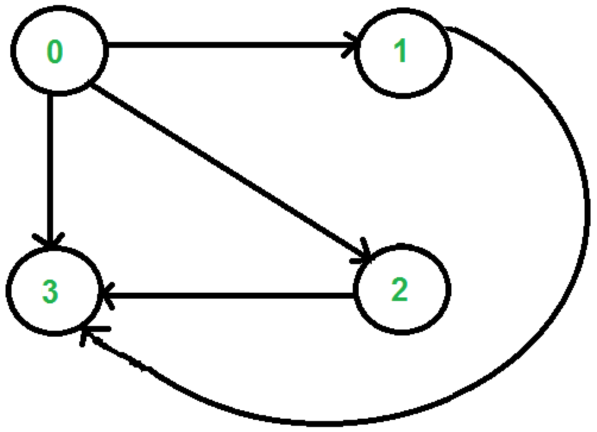
Count all possible walks from a source to a destination with exactly k edges

Count all possible walks from a source to a destination with exactly k edges - GeeksforGeeks

Given a directed graph and two vertices 'u' and 'v' in it, count all possible walks from 'u' to 'v' with exactly k edges on the walk.

The graph is given as adjacency matrix representation where value of graph[i][j] as 1 indicates that there is an edge from vertex i to vertex j and a value 0 indicates no edge from i to j.

For example consider the following graph. Let source 'u' be vertex 0, destination 'v' be 3 and k be 2. The output should be 2 as there are two walk from 0 to 3 with exactly 2 edges. The walks are {0, 2, 3} and {0, 1, 3}



A **simple solution** is to start from u, go to all adjacent vertices and recur for adjacent vertices with k as k-1, source as adjacent vertex and destination as v. Following is the implementation of this simple solution.

C++

```
 // C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A naive recursive function to count walks from u to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
   // Base cases
   if (k == 0 && u == v)      return 1;
   if (k == 1 && graph[u][v]) return 1;
   if (k <= 0)                return 0;

   // Initialize result
   int count = 0;

   // Go to all adjacents of u and recur
   for (int i = 0; i < V; i++)
       if (graph[u][i] == 1)  // Check if is adjacent of u
           count += countwalks(graph, i, v, k-1);

   return count;
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
     int graph[V][V] = { {0, 1, 1, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}
```

Java

```
 // Java program to count walks from u to v with exactly k edges
```

```
import java.util.*;
import java.lang.*;
import java.io.*;

class KPaths
{
    static final int V = 4; //Number of vertices

    // A naive recursive function to count walks from u
    // to v with k edges
    int countwalks(int graph[][], int u, int v, int k)
    {
        // Base cases
        if (k == 0 && u == v)           return 1;
        if (k == 1 && graph[u][v] == 1) return 1;
        if (k <= 0)                     return 0;

        // Initialize result
        int count = 0;

        // Go to all adjacents of u and recur
        for (int i = 0; i < V; i++)
            if (graph[u][i] == 1)  // Check if is adjacent of u
                count += countwalks(graph, i, v, k-1);

        return count;
    }

    // Driver method
    public static void main (String[] args) throws java.lang.Exception
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] =new int[][] { {0, 1, 1, 1},
            {0, 0, 0, 1},
            {0, 0, 0, 1},
            {0, 0, 0, 0}
        };
        int u = 0, v = 3, k = 2;
        KPaths p = new KPaths();
        System.out.println(p.countwalks(graph, u, v, k));
    }
}//Contributed by Aakash Hasija
```

Python3

```
 # Python3 program to count walks from
# u to v with exactly k edges
# Number of vertices in the graph
V = 4

# A naive recursive function to count
# walks from u to v with k edges
def countwalks(graph, u, v, k):

    # Base cases
    if (k == 0 and u == v):
        return 1
    if (k == 1 and graph[u][v]):
        return 1
    if (k <= 0):
        return 0

    # Initialize result
    count = 0

    # Go to all adjacents of u and recur
    for i in range(0, V):

        # Check if is adjacent of u
        if (graph[u][i] == 1):
            count += countwalks(graph, i, v, k-1)

    return count

# Driver Code

# Let us create the graph shown in above diagram
graph = [[0, 1, 1, 1,],
         [0, 0, 0, 1,],
         [0, 0, 0, 1,],
         [0, 0, 0, 0] ]

u = 0; v = 3; k = 2
print(countwalks(graph, u, v, k))

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 // C# program to count walks from u to
// v with exactly k edges
using System;

class GFG {
```

```
// Number of vertices
static int V = 4;

// A naive recursive function to
// count walks from u to v with
// k edges
static int countwalks(int [,]graph, int u,
                             int v, int k)
{

    // Base cases
    if (k == 0 && u == v)
        return 1;
    if (k == 1 && graph[u,v] == 1)
        return 1;
    if (k <= 0)
        return 0;

    // Initialize result
    int count = 0;

    // Go to all adjacents of u and recur
    for (int i = 0; i < V; i++)

        // Check if is adjacent of u
        if (graph[u,i] == 1)
            count +=
            countwalks(graph, i, v, k-1);

    return count;
}

// Driver method
public static void Main ()
{

    /* Let us create the graph shown
    in above diagram*/
    int [,]graph =
       new int[,] { {0, 1, 1, 1},
                    {0, 0, 0, 1},
                    {0, 0, 0, 1},
                    {0, 0, 0, 0} };

    int u = 0, v = 3, k = 2;

    Console.Write(
         countwalks(graph, u, v, k));
    }
}

// This code is contributed by nitin mittal.
```

PHP

```
 <?php
// PHP program to count walks from u
// to v with exactly k edges

// Number of vertices in the graph
$V = 4;

// A naive recursive function to count
// walks from u to v with k edges
function countwalks( $graph, $u, $v, $k)
{
    global $V;

    // Base cases
    if ($k == 0 and $u == $v)
        return 1;
    if ($k == 1 and $graph[$u][$v])
        return 1;
    if ($k <= 0)
        return 0;

    // Initialize result
    $count = 0;

    // Go to all adjacents of u and recur
    for ( $i = 0; $i < $V; $i++)

        // Check if is adjacent of u
        if ($graph[$u][$i] == 1)
            $count += countwalks($graph, $i,
                                $v, $k - 1);

    return $count;
}

    // Driver Code
    /* Let us create the graph
       shown in above diagram*/
    $graph = array(array(0, 1, 1, 1),
                   array(0, 0, 0, 1),
                   array(0, 0, 0, 1),
                  array(0, 0, 0, 0));
    $u = 0; $v = 3; $k = 2;
    echo countwalks($graph, $u, $v, $k);

// This code is contributed by anuj_67.
?>
```

Output:

```
2
```

The worst case time complexity of the above function is O(V^k) where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly n children.
We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other Dynamic Programming problems, we fill the 3D table in bottom up manner.

C++

```
 // C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A Dynamic programming based function to count walks from u
// to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value count[i][j][e] will
    // store count of possible walks from i to j with exactly k edges
    int count[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++)  // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                count[i][j][e] = 0;

                // from base cases
                if (e == 0 && i == j)
                    count[i][j][e] = 1;
                if (e == 1 && graph[i][j])
                    count[i][j][e] = 1;

                // go to adjacent only when number of edges is more than 1
                if (e > 1)
                {
                    for (int a = 0; a < V; a++) // adjacent of source i
                        if (graph[i][a])
                            count[i][j][e] += count[a][j][e-1];
                }
           }
        }
    }
    return count[u][v][k];
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
     int graph[V][V] = { {0, 1, 1, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}
```

Java

```
 // Java program to count walks from u to v with exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class KPaths
{
    static final int V = 4; //Number of vertices

    // A Dynamic programming based function to count walks from u
    // to v with k edges
    int countwalks(int graph[][], int u, int v, int k)
    {
        // Table to be filled up using DP. The value count[i][j][e]
        // will/ store count of possible walks from i to j with
        // exactly k edges
        int count[][][] = new int[V][V][k+1];

        // Loop for number of edges from 0 to k
        for (int e = 0; e <= k; e++)
        {
            for (int i = 0; i < V; i++)  // for source
            {
                for (int j = 0; j < V; j++) // for destination
                {
                    // initialize value
                    count[i][j][e] = 0;

                    // from base cases
                    if (e == 0 && i == j)
                        count[i][j][e] = 1;
                    if (e == 1 && graph[i][j]!=0)
                        count[i][j][e] = 1;

                    // go to adjacent only when number of edges
                    // is more than 1
                    if (e > 1)
                    {
                        for (int a = 0; a < V; a++) // adjacent of i
                            if (graph[i][a]!=0)
                                count[i][j][e] += count[a][j][e-1];
                    }
               }
            }
        }
        return count[u][v][k];
    }

    // Driver method
    public static void main (String[] args) throws java.lang.Exception
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] =new int[][] { {0, 1, 1, 1},
                                     {0, 0, 0, 1},
                                     {0, 0, 0, 1},
                                     {0, 0, 0, 0}
                                    };
        int u = 0, v = 3, k = 2;
        KPaths p = new KPaths();
        System.out.println(p.countwalks(graph, u, v, k));
    }
}//Contributed by Aakash Hasija
```

C#

```
 // C# program to count walks from u to v
// with exactly k edges
using System;

class GFG {
    static int V = 4; //Number of vertices

    // A Dynamic programming based function
    // to count walks from u to v with k edges
    static int countwalks(int [,]graph, int u,
                             int v, int k)
    {
        // Table to be filled up using DP. The
        // value count[i][j][e] will/ store
        // count of possible walks from i to
        // j with exactly k edges
        int [, ,]count = new int[V,V,k+1];

        // Loop for number of edges from 0 to k
        for (int e = 0; e <= k; e++)
        {

            // for source
            for (int i = 0; i < V; i++)
            {

                // for destination
                for (int j = 0; j < V; j++)
                {
                    // initialize value
                    count[i,j,e] = 0;

                    // from base cases
                    if (e == 0 && i == j)
                        count[i,j,e] = 1;
                    if (e == 1 && graph[i,j] != 0)
                        count[i,j,e] = 1;

                    // go to adjacent only when
                    // number of edges
                    // is more than 1
                    if (e > 1)
                    {
                        // adjacent of i
                        for (int a = 0; a < V; a++)
                            if (graph[i,a]!=0)
```

```
                                count[i,j,e] +=
                                     count[a,j,e-1];
                        }
                    }
                }
            }

            return count[u,v,k];
        }

        // Driver method
        public static void Main ()
        {
            /* Let us create the graph shown in
            above diagram*/
            int [,]graph = { {0, 1, 1, 1},
                             {0, 0, 0, 1},
                             {0, 0, 0, 1},
                             {0, 0, 0, 0} };
            int u = 0, v = 3, k = 2;

            Console.WriteLine(
                  countwalks(graph, u, v, k));
        }
    }

// This is Code Contributed by anuj_67.
```

Output:

```
2
```

Time complexity of the above DP based solution is O(V^3K) which is much better than the naive solution.

We can also use **Divide and Conquer** to solve the above problem in O(V^3Logk) time. The count of walks of length k from u to v is the [u][v]'th entry in (graph[V][V])^k. We can calculate power of by doing O(Logk) multiplication by using the divide and conquer technique to calculate power. A multiplication between two matrices of size V x V takes O(V^3) time. Therefore overall time complexity of this method is O(V^3Logk).

Improved By : nitin mittal, vt_m

Source

https://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/

Chapter 14

Count number of occurrences (or frequency) in a sorted array

Count number of occurrences (or frequency) in a sorted array - GeeksforGeeks

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.

C++

```
 // C++ program to count occurrences of an element
#include<bits/stdc++.h>
using namespace std;

// Returns number of times x occurs in arr[0..n-1]
```

```
int countOccurrences(int arr[], int n, int x)
{
    int res = 0;
    for (int i=0; i<n; i++)
        if (x == arr[i])
          res++;
    return res;
}

// Driver code
int main()
{
    int arr[] = {1, 2, 2, 2, 2, 3, 4, 7 ,8 ,8 };
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 2;
    cout << countOccurrences(arr, n, x);
    return 0;
}
```

Java

```
 // Java program to count occurrences
// of an element

class Main
{
    // Returns number of times x occurs in arr[0..n-1]
    static int countOccurrences(int arr[], int n, int x)
    {
        int res = 0;
        for (int i=0; i<n; i++)
            if (x == arr[i])
              res++;
        return res;
    }

    public static void main(String args[])
    {
        int arr[] = {1, 2, 2, 2, 2, 3, 4, 7 ,8 ,8 };
        int n = arr.length;
        int x = 2;
        System.out.println(countOccurrences(arr, n, x));
    }
}
```

Python3

```
 # Python3 program to count
```

```
# occurrences of an element

# Returns number of times x
# occurs in arr[0..n-1]
def countOccurrences(arr, n, x):
    res = 0
    for i in range(n):
        if x == arr[i]:
            res += 1
    return res

# Driver code
arr = [1, 2, 2, 2, 2, 3, 4, 7 ,8 ,8]
n = len(arr)
x = 2
print (countOccurrences(arr, n, x))
```

C#

```
 // C# program to count occurrences
// of an element
using System;

class GFG
{
    // Returns number of times x
    // occurs in arr[0..n-1]
    static int countOccurrences(int []arr,
                                int n, int x)
    {
        int res = 0;

        for (int i = 0; i < n; i++)
            if (x == arr[i])
            res++;

        return res;
    }

    // driver code
    public static void Main()
    {
        int []arr = {1, 2, 2, 2, 2, 3, 4, 7 ,8 ,8 };
        int n = arr.Length;
        int x = 2;

        Console.Write(countOccurrences(arr, n, x));
    }
}

// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP program to count occurrences
// of an element

// Returns number of times x
// occurs in arr[0..n-1]
function countOccurrences($arr, $n, $x)
{
    $res = 0;
    for ($i = 0; $i < $n; $i++)
        if ($x == $arr[$i])
        $res++;
    return $res;
}

    // Driver code
    $arr = array(1, 2, 2, 2, 2, 3, 4, 7 ,8 ,8 );
    $n = count($arr);
    $x = 2;
    echo countOccurrences($arr,$n, $x);

// This code is contributed by Sam007
?>
```

Output :

```
4
```

Time Complexity: O(n)

Method 2 (Better using Binary Search)
We first find an occurrence using binary search. Then we match toward left and right sides of the matched the found index.

C++

```
 // C++ program to count occurrences of an element
#include <bits/stdc++.h>
using namespace std;
```

```
// 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)
        return -1;

    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);
}

// Returns number of times x occurs in arr[0..n-1]
int countOccurrences(int arr[], int n, int x)
{
    int ind = binarySearch(arr, 0, n - 1, x);

    // If element is not present
    if (ind == -1)
        return 0;

    // Count elements on left side.
    int count = 1;
    int left = ind - 1;
    while (left >= 0 && arr[left] == x)
        count++, left--;

    // Count elements on right side.
    int right = ind + 1;
    while (right < n && arr[right] == x)
        count++, right++;

    return count;
}
```

```
// Driver code
int main()
{
    int arr[] = { 1, 2, 2, 2, 2, 3, 4, 7, 8, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 2;
    cout << countOccurrences(arr, n, x);
    return 0;
}
```

Java

```
 // Java program to count
// occurrences of an element
class GFG
{

    // A recursive binary search
    // function. It returns location
    // of x in given array arr[l..r]
    // is present, otherwise -1
    static int binarySearch(int arr[], int l,
                            int r, int x)
    {
        if (r < l)
            return -1;

        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);
    }

    // Returns number of times x
    // occurs in arr[0..n-1]
    static int countOccurrences(int arr[],
                                int n, int x)
    {
        int ind = binarySearch(arr, 0,
                               n - 1, x);

        // If element is not present
        if (ind == -1)
            return 0;

        // Count elements on left side.
        int count = 1;
        int left = ind - 1;
        while (left >= 0 &&
               arr[left] == x)
        {
            count++;
            left--;
        }

        // Count elements
        // on right side.
        int right = ind + 1;
        while (right < n &&
               arr[right] == x)
        {
            count++;
            right++;
        }

        return count;
    }


    // Driver code
    public static void main(String[] args)
    {
        int arr[] = {1, 2, 2, 2, 2,
                     3, 4, 7, 8, 8};
        int n = arr.length;
        int x = 2;
        System.out.print(countOccurrences(arr, n, x));
    }
}

// This code is contributed
// by ChitraNayal
```

Python 3

```
 # Python 3 program to count
# occurrences of an element

# A recursive binary search
# function. It returns location
# of x in given array arr[l..r]
# is present, otherwise -1
def binarySearch(arr, l, r, x):
    if (r < l):
        return -1

    mid = int( 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)

# Returns number of times
# x occurs in arr[0..n-1]
def countOccurrences(arr, n, x):
    ind = binarySearch(arr, 0, n - 1, x)

    # If element is not present
    if ind == -1:
        return 0

    # Count elements
    # on left side.
    count = 1
    left = ind - 1
    while (left >= 0 and
           arr[left] == x):
```

```
        count += 1
        left -= 1

    # Count elements on
    # right side.
    right = ind + 1;
    while (right < n and
           arr[right] == x):
        count += 1
        right += 1

    return count

# Driver code
arr = [ 1, 2, 2, 2, 2,
        3, 4, 7, 8, 8 ]
n = len(arr)
x = 2
print(countOccurrences(arr, n, x))

# This code is contributed
# by ChitraNayal
```

C#

```
 // C# program to count
// occurrences of an element
using System;

class GFG
{

    // A recursive binary search
    // function. It returns location
    // of x in given array arr[l..r]
    // is present, otherwise -1
    static int binarySearch(int[] arr, int l,
                            int r, int x)
    {
        if (r < l)
            return -1;

        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);
    }

    // Returns number of times x
    // occurs in arr[0..n-1]
    static int countOccurrences(int[] arr,
                                int n, int x)
    {
        int ind = binarySearch(arr, 0,
                               n - 1, x);

        // If element is not present
        if (ind == -1)
            return 0;

        // Count elements on left side.
        int count = 1;
        int left = ind - 1;
        while (left >= 0 &&
               arr[left] == x)
        {
            count++;
            left--;
        }

        // Count elements on right side.
        int right = ind + 1;
        while (right < n &&
               arr[right] == x)
        {
            count++;
            right++;
        }

        return count;
    }
    // Driver code
    public static void Main()
    {
        int[] arr = {1, 2, 2, 2, 2,
                     3, 4, 7, 8, 8};
        int n = arr.Length;
        int x = 2;
        Console.Write(countOccurrences(arr, n, x));
    }
}

// This code is contributed
// by ChitraNayal
```

PHP

```
 <?php
// PHP program to count
// occurrences of an element

// A recursive binary search
// function. It returns location
// of x in given array arr[l..r]
// is present, otherwise -1
function binarySearch(&$arr, $l,
                         $r, $x)
{
    if ($r < $l)
        return -1;

    $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);
}

// Returns number of times
// x occurs in arr[0..n-1]
function countOccurrences($arr, $n, $x)
{
    $ind = binarySearch($arr, 0,
                        $n - 1, $x);

    // If element is not present
    if ($ind == -1)
        return 0;

    // Count elements
    // on left side.
    $count = 1;
    $left = $ind - 1;
    while ($left >= 0 &&
           $arr[$left] == $x)
    {
        $count++;
        $left--;
    }

    // Count elements on right side.
    $right = $ind + 1;
    while ($right < $n &&
           $arr[$right] == $x)
    {
        $count++;
        $right++;
    }
    return $count;
}

// Driver code
$arr = array( 1, 2, 2, 2, 2,
              3, 4, 7, 8, 8 );
$n = sizeof($arr);
$x = 2;
echo countOccurrences($arr, $n, $x);

// This code is contributed
// by ChitraNayal
?>
```

Output :

```
4
```

Time Complexity : O(Log n + count) where count is number of occurrences.

Method 3 (Best using Improved 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);

C++

```
 // C++ program to count occurrences of an element
// in a sorted array.
# include <bits/stdc++.h>
using namespace std;

/* if x is present in arr[] then returns the count
    of occurrences of x, otherwise returns 0. */
int count(int arr[], int x, int n)
{
  /* get the index of first occurrence of x */
  int *low = lower_bound(arr, arr+n, x);

  // If element is not present, return 0
  if (low == (arr + n) || *low != x)
     return 0;

  /* Else get the index of last occurrence of x.
     Note that we  are only looking in the
     subarray after first occurrence */
  int *high = upper_bound(low, arr+n, x);

  /* return count */
  return high - low;
}

/* 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);
  return 0;
}
```

C

```
 # include <stdio.h>

/* 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;
}

/* if x is present in arr[] then returns the count
   of occurrences of x, otherwise returns -1. */
int count(int arr[], int x, int n)
```

```
{
  int i; // index of first occurrence of x in arr[0..n-1]
  int j; // index of last occurrence of x in arr[0..n-1]

  /* get the index of first occurrence of x */
  i = first(arr, 0, n-1, x, n);

  /* If x doesn't exist in arr[] then return -1 */
  if(i == -1)
    return i;

  /* Else get the index of last occurrence of x.
     Note that we are only looking in the subarray
     after first occurrence */
  j = last(arr, i, n-1, x, n);

  /* return count */
  return j-i+1;
}

/* 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;
}
```

Java

```
 // Java program to count occurrences
// of an element

class Main
{
    /* if x is present in arr[] then returns
       the count of occurrences of x,
       otherwise returns -1. */
    static int count(int arr[], int x, int n)
    {
      // index of first occurrence of x in arr[0..n-1]
      int i;

      // index of last occurrence of x in arr[0..n-1]
```

```
  int j;

  /* 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 */
static int first(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    /*low + (high - low)/2;*/
    int mid = (low + high)/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 */
static int last(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    /*low + (high - low)/2;*/
    int mid = (low + high)/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;
    }

    public static void main(String args[])
    {
        int arr[] = {1, 2, 2, 3, 3, 3, 3};

        // Element to be counted in arr[]
        int x =  3;
        int n = arr.length;
        int c = count(arr, x, n);
        System.out.println(x+" occurs "+c+" times");
    }
}
```

Python3

```
 # Python3 program to count
# occurrences of an element

# if x is present in arr[] then
# returns the count of occurrences
# of x, otherwise returns -1.
def count(arr, x, n):

    # 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 return
# the index of FIRST occurrence of x in
# arr[0..n-1], otherwise returns -1
def first(arr, low, high, x, n):
    if high >= low:

        # low + (high - low)/2
        mid = (low + high)//2

    if (mid == 0 or x > arr[mid-1]) and arr[mid] == x:
        return mid
    elif 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 return
# the index of LAST occurrence of x
# in arr[0..n-1], otherwise returns -1
def last(arr, low, high, x, n):
    if high >= low:

        # low + (high - low)/2
        mid = (low + high)//2;

    if(mid == n-1 or x < arr[mid+1]) and arr[mid] == x :
        return mid
    elif 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
arr = [1, 2, 2, 3, 3, 3, 3]
x = 3  # Element to be counted in arr[]
n = len(arr)
c = count(arr, x, n)
print ("%d occurs %d times "%(x, c))
```

C#

```
 // C# program to count occurrences
// of an element
using System;

class GFG
{
```

```
/* if x is present in arr[] then returns
the count of occurrences of x,
otherwise returns -1. */
static int count(int []arr, int x, int n)
{
// index of first occurrence of x in arr[0..n-1]
int i;

// index of last occurrence of x in arr[0..n-1]
int j;

/* 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 */
static int first(int []arr, int low, int high,
                                int x, int n)
{
if(high >= low)
{
    /*low + (high - low)/2;*/
    int mid = (low + high)/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 */
    static int last(int []arr, int low,
                        int high, int x, int n)
    {
    if(high >= low)
    {
        /*low + (high - low)/2;*/
        int mid = (low + high)/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;
    }

    public static void Main()
    {
        int []arr = {1, 2, 2, 3, 3, 3, 3};

        // Element to be counted in arr[]
        int x = 3;
        int n = arr.Length;
        int c = count(arr, x, n);

        Console.Write(x + " occurs " + c + " times");
    }
}
// This code is contributed by Sam007
```

Output:

```
3 occurs 4 times
```

Time Complexity: O(Logn)
Programming Paradigm: Divide & Conquer

Improved By : Sam007, ChitraNayal

Source

https://www.geeksforgeeks.org/count-number-of-occurrences-or-frequency-in-a-sorted-array/

Chapter 15

Cut all the rods with some length such that the sum of cut-off length is maximized

Cut all the rods with some length such that the sum of cut-off length is maximized - GeeksforGeeks

Given N rods of different lengths. The task is to cut all the rods with some maximum integer height 'h' such that sum of cut-off lengths of the rod is maximized and must be greater than M. Print -1 if no such cut is possible.

Note: A rod cannot be cut also.

Examples:

> **Input:** N = 7, M = 8, a[] = {1, 2, 3, 5, 4, 7, 6}
> **Output:** 3
> Rod 1 and 2 are untouched, and rod 3, 4, 5, 6, 7 are cut with the cut-off lengths being (3-3) + (4-3) + (5-3) + (7-3) + (6-3) which is equal to 10 which is greater than M = 8.
>
> **Input:** N = 4, M = 2, a[] = {1, 2, 3, 3}
> **Output:** 2

Approach:

- Sort the array in ascending order
- Run a binary searchwith values low=0 and high=length[n-1], such that mid=(low+high)/2.
- Run a loop from n-1 till 0 adding the height of the rod **cut-off** to the sum.
- If the sum is greater than or equal to m, assign low with mid+1 otherwise high will be updated with mid.
- After Binary search is completed the answer will be low-1.

Below is the implementation of the above approach:

C++

```
 // C++ program to find the maximum possible
// length of rod which will be cut such that
// sum of cut off lengths will be maximum
#include <bits/stdc++.h>
using namespace std;

// Function to run Binary Search to
// find maximum cut off length
int binarySearch(int adj[], int target, int length)
{

    int low = 0;
    int high = adj[length - 1];
    while (low < high) {

        // f is the flag varibale
        // sum is for the total length cutoff
        int f = 0, sum = 0;

        int mid = (low + high) / 2;

        // Loop from higer to lower
        // for optimization
        for (int i = length - 1; i >= 0; i--) {

            // Only if length is greater
            // than cut-off length
            if (adj[i] > mid) {
                sum = sum + adj[i] - mid;
            }

            // When total cut off length becomes greater
            // than desired cut off length
            if (sum >= target) {
                f = 1;
                low = mid + 1;
                break;
            }
        }

        // If flag variable is not set
        // Change high
        if (f == 0)
            high = mid;
    }
    // returning the maximum cut off length
    return low - 1;
}

// Driver Function
int main()
{
    int n1 = 7;
    int n2 = 8;

    int adj[] = { 1, 2, 3, 4, 5, 7, 6 };

    // Sorting the array in ascending order
    sort(adj, adj + n1);

    // Calling the binarySearch Function
    cout << binarySearch(adj, n2, n1);
}
```

Java

```
 // Java program to find the
// maximum possible length
// of rod which will be cut
// such that sum of cut off
// lengths will be maximum
import java.util.*;

class GFG
{
// Function to run Binary
// Search to find maximum
// cut off length
static int binarySearch(int adj[],
                        int target,
                        int length)
{
int low = 0;
int high = adj[length - 1];
while (low < high)
{

    // f is the flag varibale
    // sum is for the total
    // length cutoff
    int f = 0, sum = 0;
    int mid = (low + high) / 2;

    // Loop from higer to lower
    // for optimization
    for (int i = length - 1;
             i >= 0; i--)
    {

        // Only if length is greater
        // than cut-off length
        if (adj[i] > mid)
        {
            sum = sum + adj[i] - mid;
        }

        // When total cut off length
        // becomes greater than
        // desired cut off length
        if (sum >= target)
        {
            f = 1;
            low = mid + 1;
            break;
        }
    }

    // If flag variable is
    // not set Change high
    if (f == 0)
        high = mid;
}

// returning the maximum
// cut off length
return low - 1;
}

// Driver Code
public static void main(String args[])
{
    int n1 = 7;
    int n2 = 8;

    int adj[] = { 1, 2, 3, 4, 5, 7, 6 };

    // Sorting the array
    // in ascending order
    Arrays.sort(adj);
    // Calling the binarySearch Function
    System.out.println(binarySearch(adj, n2, n1));
}
}

// This code is contributed
// by Arnab Kundu
```

Python3

```
 # Python 3 program to find the
# maximum possible length of
# rod which will be cut such
# that sum of cut off lengths
# will be maximum

# Function to run Binary Search
# to find maximum cut off length
def binarySearch(adj, target, length) :
    low = 0
    high = adj[length - 1]

    while (low < high) :

        # f is the flag varibale
        # sum is for the total
        # length cutoff

        # multiple assignments
        f, sum = 0, 0

        # take integer value
        mid = (low + high) // 2

        # Loop from higer to lower
        # for optimization
        for i in range(length - 1, -1 , -1) :

            # Only if length is greater
            # than cut-off length
            if adj[i] > mid :
                sum = sum + adj[i] - mid

            # When total cut off length
            # becomes greater than
            # desired cut off length
            if sum >= target :
                f = 1
                low = mid + 1
                break

        # If flag variable is
        # not set. Change high
        if f == 0 :
            high = mid

    # returning the maximum
    # cut off length
    return low - 1

# Driver code
if __name__ == "__main__" :

    n1 = 7
    n2 = 8

    # adj = [1,2,3,3]
    adj = [ 1, 2, 3, 4, 5, 7, 6]

    # Sorting the array
    # in ascending order
    adj.sort()

    # Calling the binarySearch Function
    print(binarySearch(adj, n2, n1))

# This code is contributed
# by ANKITRAI1
```

C#

```
 // C# program to find the
// maximum possible length
// of rod which will be cut
// such that sum of cut off
// lengths will be maximum
using System;

class GFG
{
// Function to run Binary
// Search to find maximum
// cut off length
static int binarySearch(int []adj,
                        int target,
                        int length)
{
int low = 0;
int high = adj[length - 1];
while (low < high)
{

    // f is the flag varibale
    // sum is for the total
    // length cutoff
    int f = 0, sum = 0;

    int mid = (low + high) / 2;

    // Loop from higer to lower
    // for optimization
    for (int i = length - 1;
             i >= 0; i--)
    {

        // Only if length is greater
        // than cut-off length
        if (adj[i] > mid)
        {
            sum = sum + adj[i] - mid;
        }

        // When total cut off length
        // becomes greater than
        // desired cut off length
        if (sum >= target)
        {
            f = 1;
            low = mid + 1;
            break;
        }
    }

    // If flag variable is
    // not set Change high
    if (f == 0)
        high = mid;
}

// returning the maximum
// cut off length
return low - 1;
}
```

```
// Driver Code
public static void Main()
{
    int n1 = 7;
    int n2 = 8;

    int []adj = {1, 2, 3, 4, 5, 7, 6};

    // Sorting the array
    // in ascending order
    Array.Sort(adj);

    // Calling the binarySearch Function
    Console.WriteLine(binarySearch(adj, n2, n1));
}
}

// This code is contributed
// by Subhadeep Gupta
```

PHP

= 0; $i–)
{

// Only if length is greater
// than cut-off length
if ($adj[$i] > $mid)
{
$sum = $sum + $adj[$i] – $mid;
}

// When total cut off length becomes
// greater than desired cut off length
if ($sum >= $target)
{
$f = 1;
$low = $mid + 1;
break;
}
}

// If flag variable is not
// set Change high
if ($f == 0)
$high = $mid;
}

// returning the maximum cut off length

```
return $low – 1;
}

// Driver Code
$n1 = 7;
$n2 = 8;

$adj = array( 1, 2, 3, 4, 5, 7, 6 );

// Sorting the array in ascending order
sort($adj);

// Calling the binarySearch Function
echo (int)binarySearch($adj, $n2, $n1);

// This code is contributed by ChitraNayal
?>
```

Output:

```
3
```

Time Complexity: O(N * log N)
Auxiliary Space: O(1)

Improved By : andrew1234, tufan_gupta2000, ANKITRAI1, ChitraNayal

Source

https://www.geeksforgeeks.org/cut-all-the-rods-with-some-length-such-that-the-sum-of-cut-off-length-is-maximize

Chapter 16

Decrease and Conquer

Decrease and Conquer - GeeksforGeeks

As divide-and-conquer approach is already discussed, which include following steps:
Divide the problem into a number of subproblems that are smaller instances of the same problem.
Conquer the sub problems by solving them recursively. If the subproblem sizes are small enough, however, just solve the sub problems in a straightforward manner.
Combine the solutions to the sub problems into the solution for the original problem.

Similarly, the approach decrease-and-conquer works, it also include following steps:
Decrease or reduce problem instance to smaller instance of the same problem and extend solution.
Conquer the problem by solving smaller instance of the problem.
Extend solution of smaller instance to obtain solution to original problem .
Basic idea of the decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. This approach is also known as incremental or inductive approach.

"Divide-and-Conquer" vs "Decrease-and-Conquer":

As per Wikipedia, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more subproblems. The name decrease and conquer has been proposed instead for the single-subproblem class. According to this definition, Merge Sort and Quick Sort comes under divide and conquer (because there are 2 sub-problems) and Binary Search comes under decrease and conquer (because there is one sub-problem).

Implementations of Decrease and Conquer :

This approach can be either implemented as top-down or bottom-up.

Top-down approach : It always leads to the recursive implementation of the problem.
Bottom-up approach : It is usually implemented in iterative way, starting with a solution to the smallest instance of the problem.

Variations of Decrease and Conquer :

There are three major variations of decrease-and-conquer:

1. Decrease by a constant
2. Decrease by a constant factor
3. Variable size decrease

Decrease by a Constant : In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one , although other constant size reductions do happen occasionally. Below are example problems :

- Insertion sort
- Graph search algorithms: DFS, BFS
- Topological sorting
- Algorithms for generating permutations, subsets

Decrease by a Constant factor: This technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. A reduction by a factor other than two is especially rare.

Decrease by a constant factor algorithms are very efficient especially when the factor is greater than 2 as in the fake-coin problem. Below are example problems :

- Binary search
- Fake-coin problems
- Russian peasant multiplication

Variable-Size-Decrease : In this variation, the size-reduction pattern varies from one iteration of an algorithm to another.
As, in problem of finding gcd of two number though the value of the second argument is always smaller on the right-handside than on the left-hand side, it decreases neither by a constant nor by a constant factor. Below are example problems :

- Computing median and selection problem.
- Interpolation Search
- Euclid's algorithm

There may be a case that problem can be solved by decrease-by-constant as well as decrease-by-factor variations, but the implementations can be either recursive or iterative. The iterative implementations may require more coding effort, however they avoid the overload that accompanies recursion.

Reference :
Anany Levitin
Decrease and conquer

Source

https://www.geeksforgeeks.org/decrease-and-conquer/

Chapter 17

Distinct elements in subarray using Mo's Algorithm

Distinct elements in subarray using Mo's Algorithm - GeeksforGeeks

Given an array 'a[]' of size n and number of queries q. Each query can be represented by two integers l and r. Your task is to print the number of distinct integers in the subarray l to r.

Given a[i] <= 10^6
Examples :

```
Input : a[] = {1, 1, 2, 1, 2, 3}
        q = 3
        0 4
        1 3
        2 5
Output : 2
         2
         3
In query 1, number of distinct integers
in a[0...4] is 2 (1, 2)
In query 2, number of distinct integers
in a[1..3] is 2 (1, 2)
In query 3, number of distinct integers
in a[2..5] is 3 (1, 2, 3)

Input : a[] = {7, 3, 5, 9, 7, 6, 4, 3, 2}
        q = 4
        1 5
        0 4
        0 7
```

```
              1 8
output : 5
              4
              6
              7
```

Let a[0…n-1] be input array and q[0..m-1] be array of queries.
Approach :

1. Sort all queries in a way that queries with L values from 0 to $\sqrt{n} - 1$ are put together, then all queries from $\sqrt{n}$ to $2 \times \sqrt{n} - 1$, and so on. All queries within a block are sorted in increasing order of R values.
2. Initialize an array freq[] of size 10^6 with 0 . freq[] array keep count of frequencies of all the elements in lying in a given range.
3. Process all queries one by one in a way that every query uses number of different elements and frequency array computed in previous query and stores the result in structure.

- Let 'curr_Diff_element' be number of different elements of previous query.
- Remove extra elements of previous query. For example if previous query is [0, 8] and current query is [3, 9], then remove a[0], a[1] and a[2]
- Add new elements of current query. In the same example as above, add a[9].

Sort the queries in the same order as they were provided earlier and print their stored results

Adding elements()

- Increase the frequency of element to be added(freq[a[i]]) by 1.
- If frequency of element a[i] is 1.Increase curr_diff_element by 1 as 1 new element has been added in range.

Removing elements()

- Decrease frequency of element to be removed (a[i]) by 1.
- if frequency of an element a[i] is 0.Just decrease curr_diff_element by 1 as 1 element has been completely removed from the range.

Note : In this algorithm, in step 2, index variable for R change at most O(n * $\sqrt{n}$) times throughout the run and same for L changes its value at most O(m * $\sqrt{n}$) times. All these bounds are possible only because sorted queries first in blocks of $\sqrt{n}$ size.

The preprocessing part takes O(m Log m) time.

Processing all queries takes O(n * $\sqrt{n}$) + O(m * $\sqrt{n}$) = O((m+n) * $\sqrt{n}$) time.

Below is the implementation of above approach :

```
 // Program to compute no. of different elements
// of ranges for different range queries
#include <bits/stdc++.h>
using namespace std;

// Used in frequency array (maximum value of an
// array element).
const int MAX = 1000000;

// Variable to represent block size. This is made
// global so compare() of sort can use it.
int block;

// Structure to represent a query range and to store
// index and result of a particular query range
struct Query {
    int L, R, index, result;
};

// Function used to sort all queries so that all queries
// of same block are arranged together and within a block,
// queries are sorted in increasing order of R values.
bool compare(Query x, Query y)
{
    // Different blocks, sort by block.
    if (x.L / block != y.L / block)
        return x.L / block < y.L / block;

    // Same block, sort by R value
    return x.R < y.R;
}

// Function used to sort all queries in order of their
// index value so that results of queries can be printed
// in same order as of input
bool compare1(Query x, Query y)
{
    return x.index < y.index;
}

// calculate distinct elements of all query ranges.
// m is number of queries n is size of array a[].
void queryResults(int a[], int n, Query q[], int m)
{
    // Find block size
    block = (int)sqrt(n);

    // Sort all queries so that queries of same
    // blocks are arranged together.
    sort(q, q + m, compare);

    // Initialize current L, current R and current
    // different elements
    int currL = 0, currR = 0;
    int curr_Diff_elements = 0;

    // Initialize frequency array with 0
    int freq[MAX] = { 0 };

    // Traverse through all queries
    for (int i = 0; i < m; i++) {

        // L and R values of current range
        int L = q[i].L, R = q[i].R;

        // Remove extra elements of previous range.
        // For example if previous range is [0, 3]
        // and current range is [2, 5], then a[0]
        // and a[1] are subtracted
        while (currL < L) {

            // element a[currL] is removed
            freq[a[currL]]--;
            if (freq[a[currL]] == 0)
                curr_Diff_elements--;

            currL++;
        }

        // Add Elements of current Range
        // Note:- during addition of the left
        // side elements we have to add currL-1
        // because currL is already in range
        while (currL > L) {
            freq[a[currL - 1]]++;

            // include a element if it occurs first time
            if (freq[a[currL - 1]] == 1)
                curr_Diff_elements++;
            currL--;
        }
        while (currR <= R) {
            freq[a[currR]]++;

            // include a element if it occurs first time
            if (freq[a[currR]] == 1)
                curr_Diff_elements++;

            currR++;
        }

        // Remove elements of previous range. For example
        // when previous range is [0, 10] and current range
        // is [3, 8], then a[9] and a[10] are subtracted
        // Note:- Basically for a previous query L to R
        // currL is L and currR is R+1. So during removal
        // of currR remove currR-1 because currR was
        // never included
        while (currR > R + 1) {

            // element a[currL] is removed
            freq[a[currR - 1]]--;

            // if ocurrence of a number is reduced
            // to zero remove it from list of
            // different elements
            if (freq[a[currR - 1]] == 0)
                curr_Diff_elements--;

            currR--;
        }
        q[i].result = curr_Diff_elements;
    }
}

// print the result of all range queries in
// initial order of queries
void printResults(Query q[], int m)
{
    sort(q, q + m, compare1);
    for (int i = 0; i < m; i++) {
        cout << "Number of different elements" <<
               " in range " << q[i].L << " to "
             << q[i].R << " are " << q[i].result << endl;
    }
}
```

```
// Driver program
int main()
{
    int a[] = { 1, 1, 2, 1, 3, 4, 5, 2, 8 };
    int n = sizeof(a) / sizeof(a[0]);
    Query q[] = { { 0, 4, 0, 0 }, { 1, 3, 1, 0 },
                  { 2, 4, 2, 0 } };
    int m = sizeof(q) / sizeof(q[0]);
    queryResults(a, n, q, m);
    printResults(q, m);
    return 0;
}
```

Output:

```
Number of different elements in range 0 to 4 are 3
Number of different elements in range 1 to 3 are 2
Number of different elements in range 2 to 4 are 3
```

Source

https://www.geeksforgeeks.org/distinct-elements-subarray-using-mos-algorithm/

Chapter 18

Divide and Conquer Algorithm | Introduction

Divide and Conquer Algorithm | Introduction - GeeksforGeeks

Like Greedyand Dynamic Programming, Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide:* Break the given problem into subproblems of same type.
2. *Conquer:* Recursively solve these subproblems
3. *Combine:* Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

1) Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5) Strassen's Algorithm is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.

6) Cooley–Tukey Fast Fourier Transform (FFT) algorithm is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(nlogn) time.

7) Karatsuba algorithm for fast multiplication it does multiplication of two n-digit numbers in at most $3n^{\log_2 3} \approx 3n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3}$ when n is a power of 2). It is therefore faster than the classicalalgorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59{,}049$ and $(2^{10})^2 = 1{,}048{,}576$, respectively.

We will publishing above algorithms in separate posts.

Divide and Conquer (D & C) vs Dynamic Programming (DP)
Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How to choose one of them for a given problem? Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used. For example, Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for calculating nth Fibonacci number, Dynamic Programming should be preferred (See thisfor details).

References
Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.
http://en.wikipedia.org/wiki/Karatsuba_algorithm

Source

https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

Chapter 19

Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)

Divide and Conquer | Set 5 (Strassen's Matrix Multiplication) - GeeksforGeeks

Given two square matrices A and B of size n x n each, find their multiplication matrix.

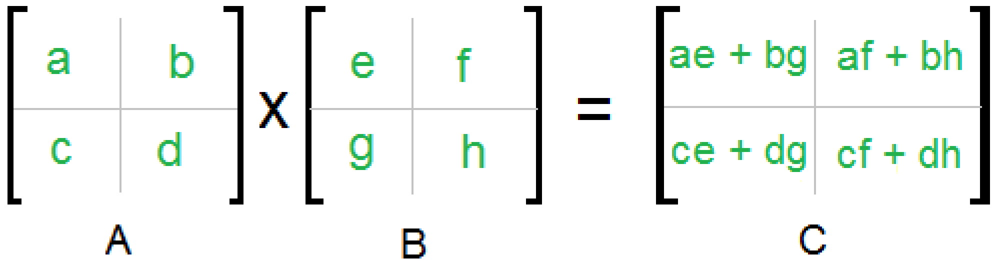
Naive Method
Following is a simple way to multiply two matrices.

```
 void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is O(N^3).

Divide and Conquer
Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below

diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

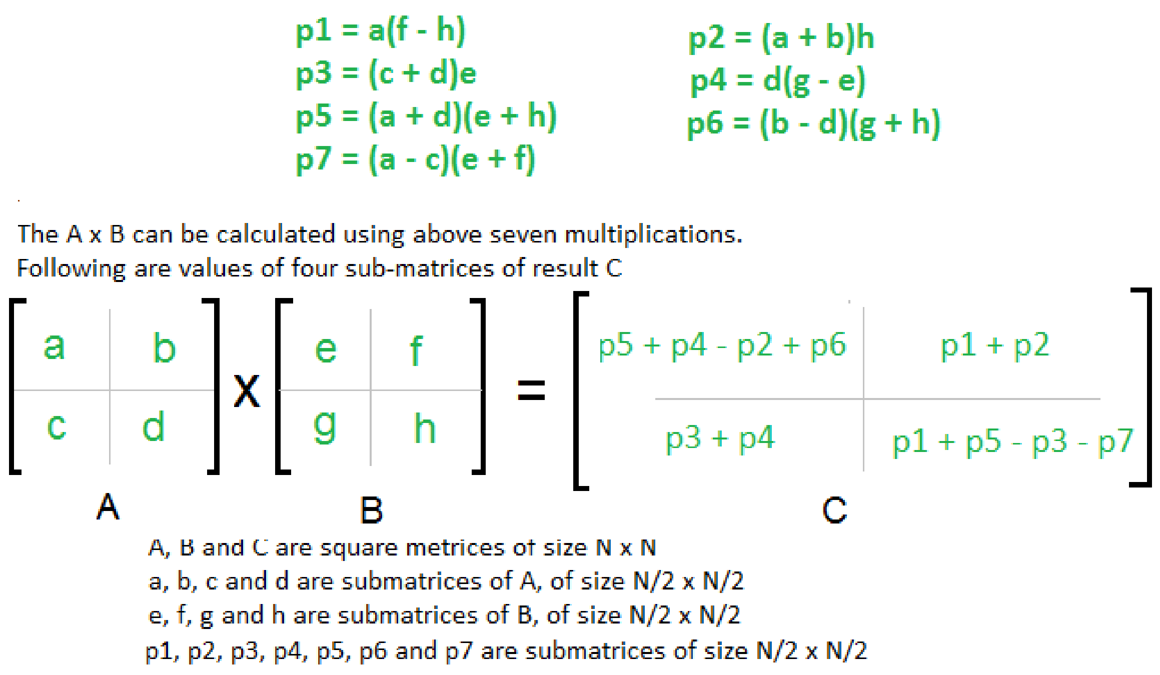


A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes O(N^2) time. So the time complexity can be written as

```
T(N) = 8T(N/2) + O(N2)

From Master's Theorem, time complexity of above method is O(N3)
which is unfortunately same as the above naive method.
```

Simple Divide and Conquer also leads to O(N^3), can there be a better way?
In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.



$p1 = a(f - h)$ $p2 = (a + b)h$
$p3 = (c + d)e$ $p4 = d(g - e)$
$p5 = (a + d)(e + h)$ $p6 = (b - d)(g + h)$
$p7 = (a - c)(e + f)$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method
Addition and Subtraction of two matrices takes O(N^2) time. So time complexity can be written as

```
T(N) = 7T(N/2) +  O(N2)

From Master's Theorem, time complexity of above method is
O(NLog7) which is approximately O(N2.8074)
```

Generally Strassen's Method is not preferred for practical applications for following reasons.
1) The constants used in Strassen's method are high and for a typical application Naive method works better.
2) For Sparse matrices, there are better methods especially designed for them.
3) The submatrices in recursion take extra space.
4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method (Source: CLRS Book)

Easy way to remember Strassen's Matrix Equation

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
https://www.youtube.com/watch?v=LOLebQ8nKHA
https://www.youtube.com/watch?v=QXY4RskLQcI

Source

https://www.geeksforgeeks.org/strassens-matrix-multiplication/

Chapter 20

Dynamic Programming vs Divide-and-Conquer

Dynamic Programming vs Divide-and-Conquer - GeeksforGeeks

TL;DR

In this article I'm trying to explain the difference/similarities between dynamic programing and divide and conquer approaches based on two examples: binary search and minimum edit distance (Levenshtein distance).

The Problem

When I started to learn algorithms it was hard for me to understand the main idea of dynamic programming (DP) and how it is different from divide-and-conquer (DC) approach. When it gets to comparing those two paradigms usually Fibonacci function comes to the rescue as great example. But when we're trying to solve the same problem using both DP and DC approaches to explain each of them, it feels for me like we may lose valuable detail that might help to catch the difference faster. And these detail tells us that each technique serves best for different types of problems.

I'm still in the process of understanding DP and DC difference and I can't say that I've fully grasped the concepts so far. But I hope this article will shed some extra light and help you to do another step of learning such valuable algorithm paradigms as dynamic programming and divide-and-conquer.

Dynamic Programming and Divide-and-Conquer Similarities

As I see it for now I can say that dynamic programming is an extension of divide and conquer paradigm.

I would not treat them as something completely different. Because they both work by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

So why do we still have different paradigm names then and why I called dynamic programming an extension. It is because dynamic programming approach may be applied to the problem only if the problem has certain restrictions or prerequisites. And after that dynamic programming extends divide and conquer approach with memoization or tabulation technique.

Let's go step by step…

Dynamic Programming Prerequisites/Restrictions

As we've just discovered there are two key attributes that divide and conquer problem must have in order for dynamic programming to be applicable:

1. Optimal substructure—optimal solution can be constructed from optimal solutions of its subproblems
2. Overlapping sub-problems—problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems

Once these two conditions are met we can say that this divide and conquer problem may be solved using dynamic programming approach.

Dynamic Programming Extension for Divide and Conquer

Dynamic programming approach extends divide and conquer approach with two techniques (memoization and tabulation) that both have a purpose of storing and re-using sub-problems solutions that may drastically improve performance. For example naive recursive implementation of Fibonacci function has time complexity of O(2^n) where DP solution doing the same with only O(n) time.

Memoization (top-down cache filling) refers to the technique of caching and reusing previously computed results. The memoized fib function would thus look like this:

```
memFib(n) {
    if (mem[n] is undefined)
        if (n < 2) result = n
        else result = memFib(n-2) + memFib(n-1)
        mem[n] = result
    return mem[n]
}
```

Tabulation (bottom-up cache filling) is similar but focuses on filling the entries of the cache. Computing the values in the cache is easiest done iteratively. The tabulation version of fib would look like this:

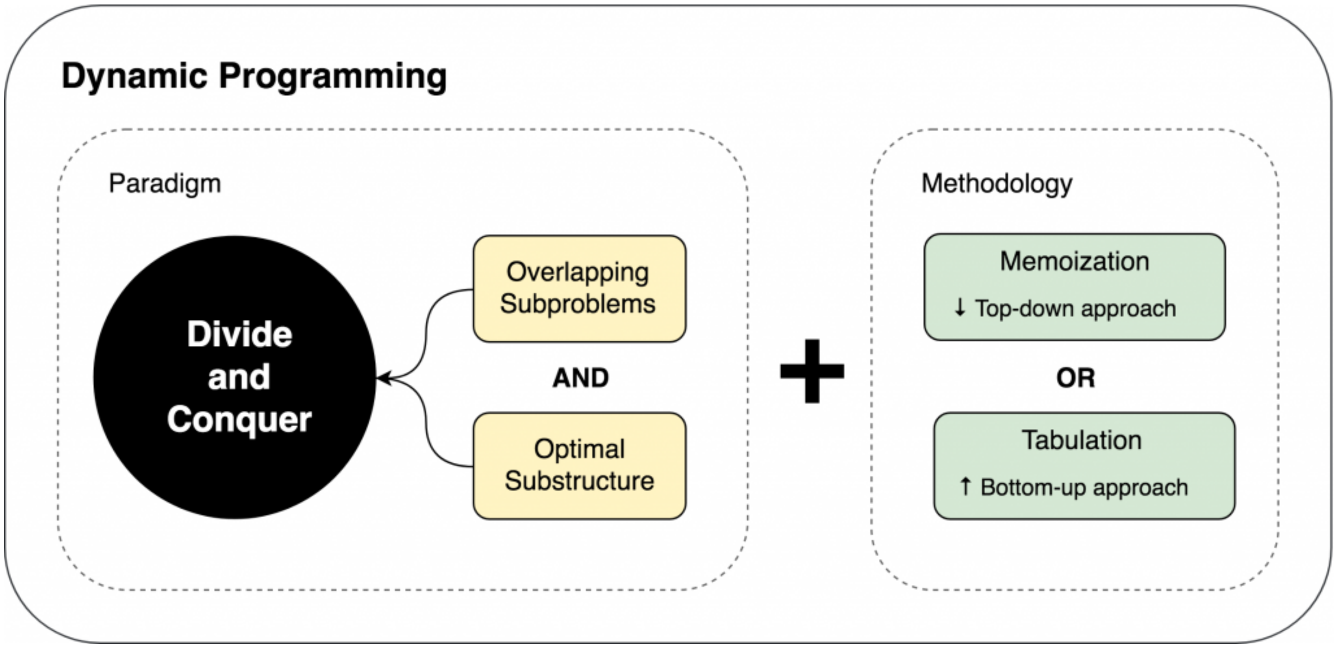


Figure 20.1:

```
tabFib(n) {
    mem[0] = 0
    mem[1] = 1
    for i = 2...n
        mem[i] = mem[i-2] + mem[i-1]
    return mem[n]
}
```

You may read more about memoization and tabulation comparison here.

The main idea you should grasp here is that because our divide and conquer problem has overlapping sub-problems the caching of sub-problem solutions becomes possible and thus memoization/tabulation step up onto the scene.

So What the Difference Between DP and DC After All

Since we're now familiar with DP prerequisites and its methodologies we're ready to put all that was mentioned above into one picture.

Let's go and try to solve some problems using DP and DC approaches to make this illustration more clear.

Divide and Conquer Example: Binary Search

Binary search algorithm, also known as half-interval search, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Example

Here is a visualization of the binary search algorithm where 4 is the target value.

Let's draw the same logic but in form of decision tree.

You may clearly see here a divide and conquer principle of solving the problem. We're iteratively breaking the original array into sub-arrays and trying to find required element in there.

Can we apply dynamic programming to it? No. It is because there are no overlapping sub-problems. Every time we split the array into completely independent parts. And according to divide and conquer prerequisites/restrictions the sub-problems must be overlapped somehow.

Normally every time you draw a decision tree and it is actually a tree (and not a decision graph) it would mean that you don't have overlapping sub-problems and this is not dynamic programming problem.

The Code

Here you may find complete source code of binary search function with test cases and explanations.

```
function binarySearch(sortedArray, seekElement) {
  let startIndex = 0;
  let endIndex = sortedArray.length - 1;
  while (startIndex <= endIndex) {
    const middleIndex = startIndex + Math.floor((endIndex - startIndex) / 2);
    // If we've found the element just return its position.
    if (sortedArray[middleIndex] === seekElement)) {
      return middleIndex;
    }
    // Decide which half to choose: left or right one.
    if (sortedArray[middleIndex] < seekElement)) {
      // Go to the right half of the array.
      startIndex = middleIndex + 1;
    } else {
      // Go to the left half of the array.
      endIndex = middleIndex - 1;
    }
  }
  return -1;
}
```

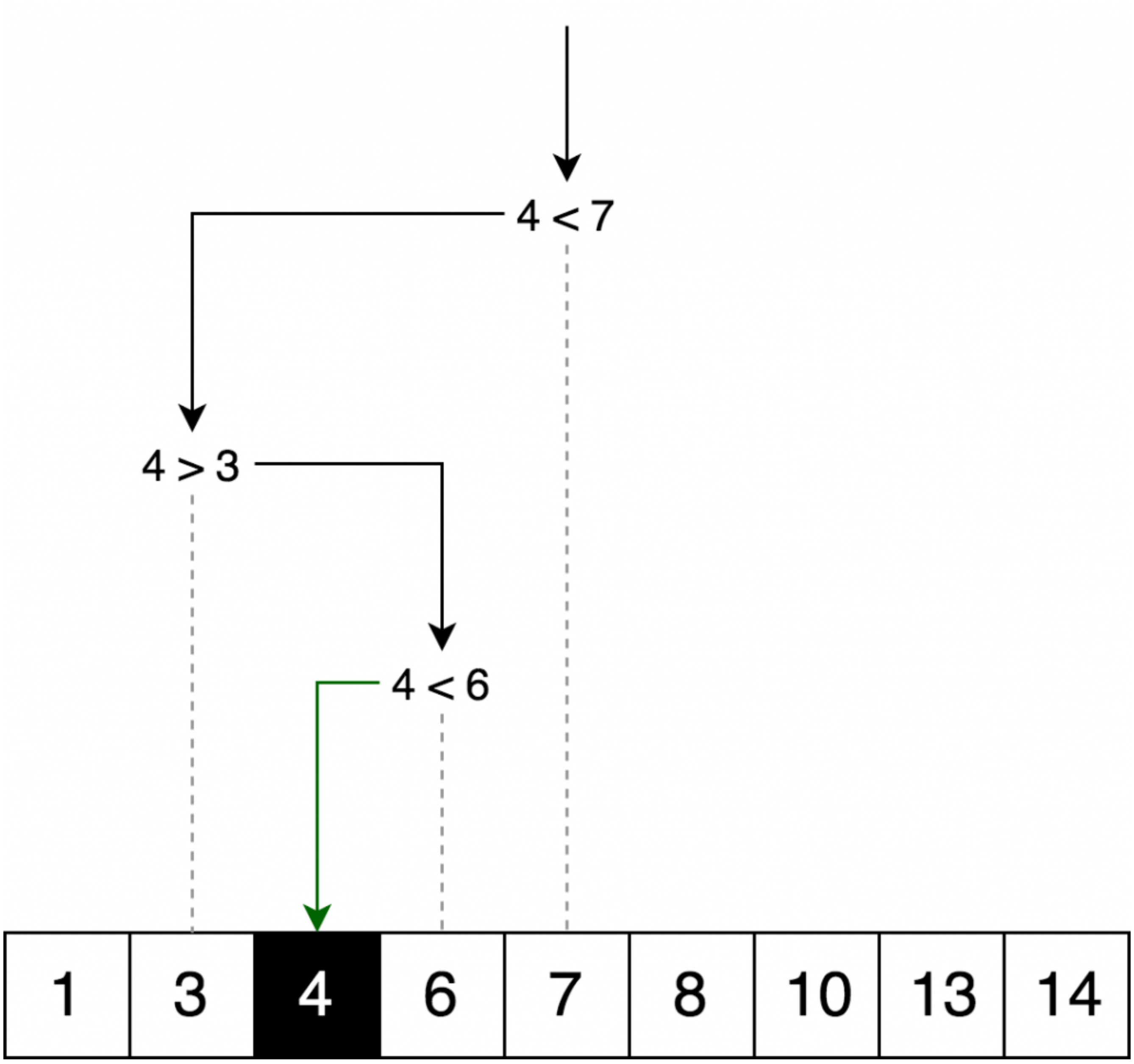


Figure 20.2:

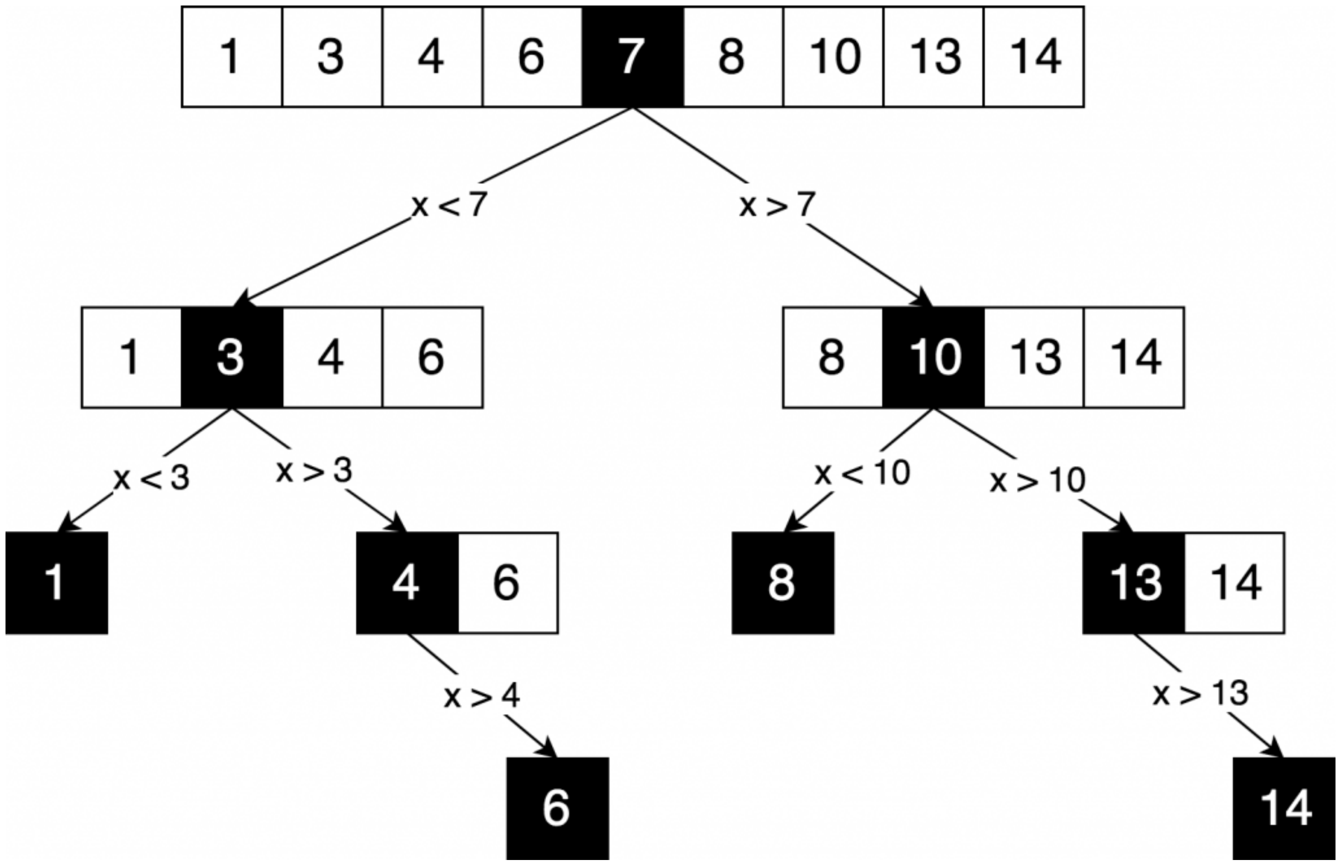

1 3 4 6 7 8 10 13 14
x < 7
x > 7
1 3 4 6
8 10 13 14
x < 3
x > 3
x < 10
x > 10
1
4 6
8
13 14
x > 4
x > 13
6
14

Figure 20.3:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure 20.4:

Dynamic Programming Example: Minimum Edit Distance

Normally when it comes to dynamic programming examples the Fibonacci number algorithm is being taken by default. But let's take a little bit more complex algorithm to have some kind of variety that should help us to grasp the concept.

Minimum Edit Distance (or Levenshtein Distance) is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

Example

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. kitten > sitten (substitution of "s" for "k")
2. sitten > sittin (substitution of "i" for "e")
3. sittin > sitting (insertion of "g" at the end).

Applications

This has a wide range of applications, for instance, spell checkers, correction systems for optical character recognition, fuzzy string searching, and software to assist natural language translation based on translation memory.

Mathematical Definition

Mathematically, the Levenshtein distance between two strings a, b (of length |a| and |b| respectively) is given by function lev(|a|, |b|) where

Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

Explanation

Ok, let's try to figure out what that formula is talking about. Let's take a simple example of finding minimum edit distance between strings ME and MY. Intuitively you already know that minimum edit distance here is 1 operation and this operation is "replace E with Y". But let's try to formalize it in a form of the algorithm in order to be able to do more complex examples like transforming Saturday into Sunday.

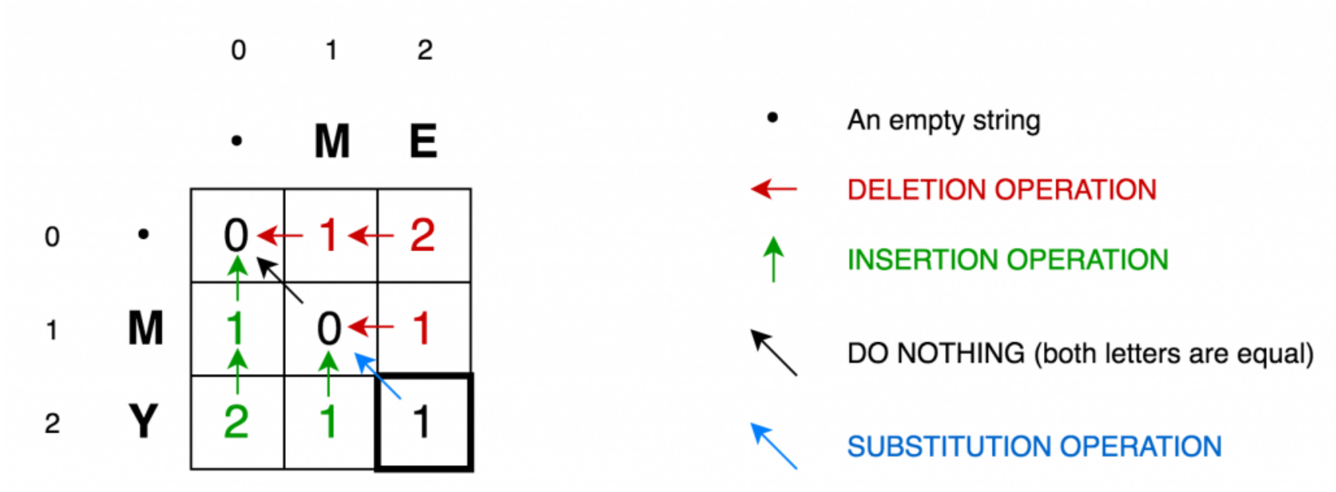


Figure 20.5: Simple example of finding minimum edit distance between ME and MY strings

To apply the formula to ME>MY transformation we need to know minimum edit distances of ME>M, M>MY and M>M transformations in prior. Then we will need to pick the minimum one and add +1 operation to transform last letters E?Y.

So we can already see here a recursive nature of the solution: minimum edit distance of ME>MY transformation is being calculated based on three previously possible transformations. Thus we may say that this is divide and conquer algorithm.

To explain this further let's draw the following matrix.

Cell (0, 1) contains red number 1. It means that we need 1 operation to transform M to empty string: delete M. This is why this number is red.

Cell (0, 2) contains red number 2. It means that we need 2 operations to transform ME to empty string: delete E, delete M.

Cell (1, 0) contains green number 1. It means that we need 1 operation to transform empty string to M: insert M. This is why this number is green.

Cell (2, 0) contains green number 2. It means that we need 2 operations to transform empty string to MY: insert Y, insert M.

Cell (1, 1) contains number 0. It means that it costs nothing to transform M to M.

Cell (1, 2) contains red number 1. It means that we need 1 operation to transform ME to M: delete E.

And so on…

This looks easy for such small matrix as ours (it is only 3×3). But how we could calculate all those numbers for bigger matrices (let's say 9×7 one, for Saturday>Sunday transformation)?

The good news is that according to the formula you only need three adjacent cells (i-1, j), (i-1, j-1), and (i, j-1) to calculate the number for current cell (i, j) . All we need to do is to

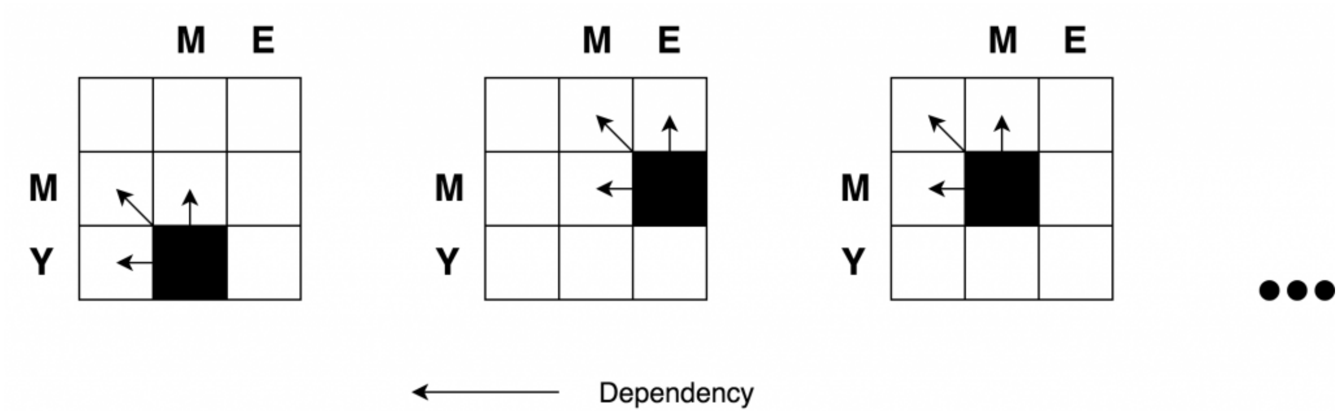


Figure 20.6: Recursive nature of minimum edit distance problem

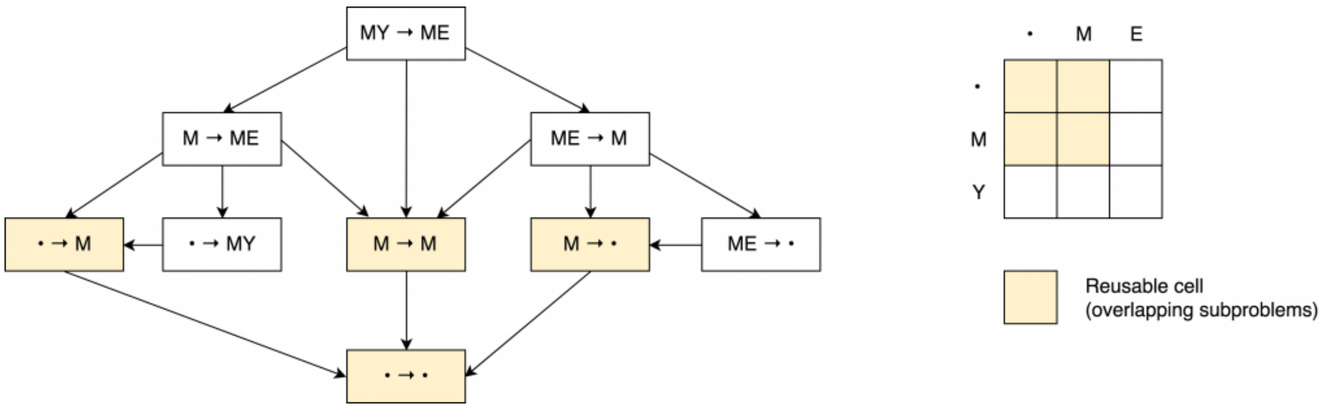


Figure 20.7: Decision graph for minimum edit distance with overlapping sub-problems

find the minimum of those three cells and then add +1 in case if we have different letters in i-s row and j-s column

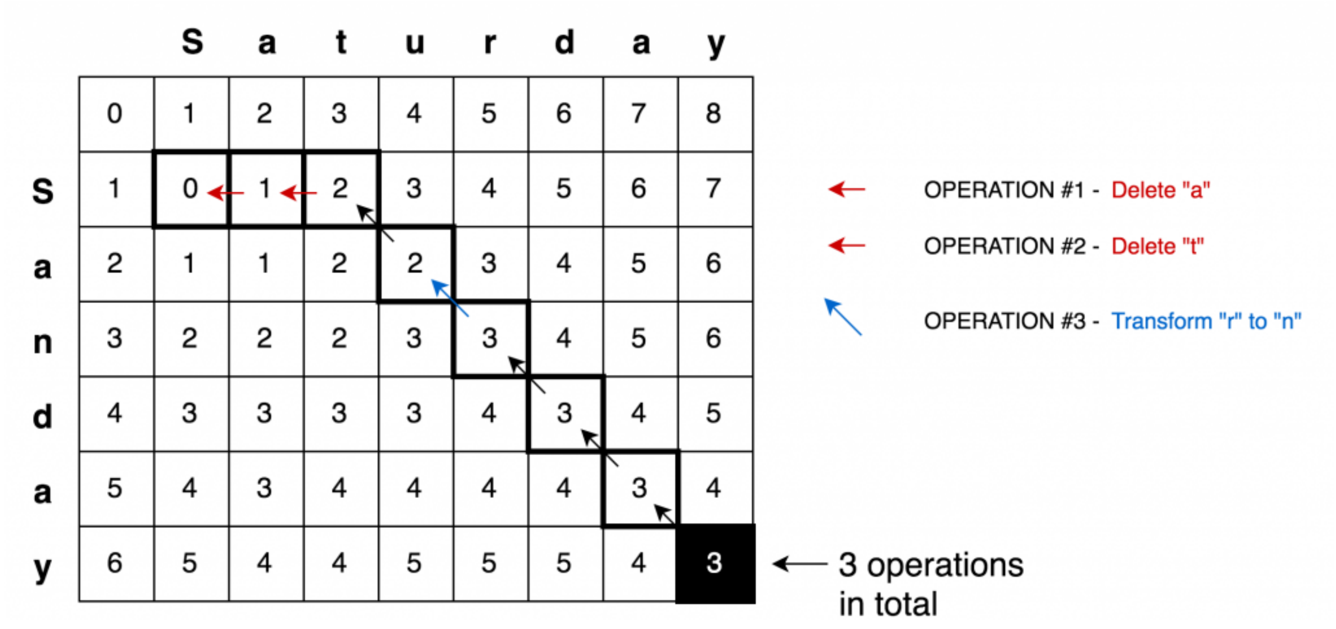
So once again you may clearly see the recursive nature of the problem.

Ok we've just found out that we're dealing with divide and conquer problem here. But can we apply dynamic programming approach to it? Does this problem satisfies our overlapping sub-problems and optimal substructure restrictions? Yes. Let's see it from decision graph.

First of all this is not a decision tree. It is a decision graph. You may see a number of overlapping subproblems on the picture that are marked with red. Also there is no way to reduce the number of operations and make it less then a minimum of those three adjacent cells from the formula.

Also you may notice that each cell number in the matrix is being calculated based on previous ones. Thus the tabulation technique (filling the cache in bottom-up direction) is being applied here. You'll see it in code example below.

Applying this principles further we may solve more complicated cases like with Saturday >


S a t u r d a y
S a n d a y
OPERATION #1 - Delete "a"
OPERATION #2 - Delete "t"
OPERATION #3 - Transform "r" to "n"
3 operations in total

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
a	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 20.8: Minimum edit distance to convert Saturday to Sunday

Sunday transformation.

The Code

Here you may find complete source code of minimum edit distance function with test cases and explanations.

```
function levenshteinDistance(a, b) {
  const distanceMatrix = Array(b.length + 1)
    .fill(null)
    .map(
      () => Array(a.length + 1).fill(null)
    );

  for (let i = 0; i <= a.length; i += 1) {
    distanceMatrix[0][i] = i;
  }

  for (let j = 0; j <= b.length; j += 1) {
    distanceMatrix[j][0] = j;
  }

  for (let j = 1; j <= b.length; j += 1) {
    for (let i = 1; i <= a.length; i += 1) {
      const indicator = a[i - 1] === b[j - 1] ? 0 : 1;
      distanceMatrix[j][i] = Math.min(
        distanceMatrix[j][i - 1] + 1, // deletion
        distanceMatrix[j - 1][i] + 1, // insertion
        distanceMatrix[j - 1][i - 1] + indicator, // substitution
      );
    }
  }

  return distanceMatrix[b.length][a.length];
}
```

Conclusion

In this article we have compared two algorithmic approaches such as dynamic programming and divide-and-conquer. We've found out that dynamic programing is based on divide and conquer principle and may be applied only if the problem has overlapping sub-problems and optimal substructure (like in Levenshtein distance case). Dynamic programming then is using memoization or tabulation technique to store solutions of overlapping sub-problems for later usage.

I hope this article hasn't brought you more confusion but rather shed some light on these two important algorithmic concepts!

You may find more examples of divide and conquer and dynamic programming problems with explanations, comments and test cases in JavaScript Algorithms and Data Structures repository.

Happy coding!

Source

https://www.geeksforgeeks.org/dynamic-programming-vs-divide-and-conquer/

Chapter 21

Easy way to remember Strassen's Matrix Equation

Easy way to remember Strassen's Matrix Equation - GeeksforGeeks

Strassen's matrix is a Divide and Conquer method that helps us to multiply two matrices(of size n X n).

You can refer to the link, for having the knowledge about Strassen's Matrix first :
Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)

But this method needs to cram few equations, so I'll tell you the simplest way to remember those :

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{X}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{Y}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

X , Y and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

You just need to remember 4 Rules :

- AHED (Learn it as 'Ahead')
- Diagonal
- Last CR
- First CR

Also, consider X as (Row +) and Y as (Column -) matrix

Follow the Steps :

- Write P1 = A; P2 = H; P3 = E; P4 = D
- For P5 we will use Diagonal Rule i.e.
  (Sum the Diagonal Elements Of Matrix X) * (Sum the Diagonal Elements Of Matrix Y), we get
  P5 = (A + D)* (E + H)
- For P6 we will use Last CR Rule i.e. Last Column of X and Last Row of Y and remember that Row+ and Column- so i.e. (B – D) * (G + H), we get
  P6 = (B – D) * (G + H)
- For P7 we will use First CR Rule i.e. First Column of X and First Row of Y and remember that Row+ and Column- so i.e. (A – C) * (E + F), we get
  P6 = (A – C) * (E + F)

- Come Back to P1 : we have A there and it's adjacent element in Y Matrix is E, since Y is Column Matrix so we select a column in Y such that E won't come, we find F H Column, so multiply A with (F – H)
  So, finally P1 = A * (F – H)
- Come Back to P2 : we have H there and it's adjacent element in X Matrix is D, since X is Row Matrix so we select a Row in X such that D won't come, we find A B Column, so multiply H with (A + B)
  So, finally P2 = H * (A + B)
- Come Back to P3 : we have E there and it's adjacent element in X Matrix is A, since X is Row Matrix so we select a Row in X such that A won't come, we find C D Column, so multiply E with (C + D)
  So, finally P3 = E * (C + D)
- Come Back to P4 : we have D there and it's adjacent element in Y Matrix is H, since Y is Column Matrix so we select a column in Y such that H won't come, we find G E Column, so multiply D with (G – E)
  So, finally P4 = D * (G – E)
- Remember Counting : Write P1 + P2 at C2
- Write P3 + P4 at its diagonal Position i.e. at C3
- Write P4 + P5 + P6 at 1st position and subtract P2 i.e. C1 = P4 + P5 + P6 – P2
- Write odd values at last Position with alternating – and + sign i.e. P1 P3 P5 P7 becomes
  C4 = P1 – P3 + P5 – P7

Improved By : BhavayAnand

Source

https://www.geeksforgeeks.org/easy-way-remember-strassens-matrix-equation/

Chapter 22

Fast Fourier Transformation for poynomial multiplication

Fast Fourier Transformation for poynomial multiplication - GeeksforGeeks

Given two polynomial A(x) and B(x), find the product C(x) = A(x)*B(x). There is already an O(n^2) naive approach to solve this problem. here. This approach uses the coefficient form of the polynomial to calculate the product.

A coefficient representation of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ is a = a0, a1, …, an-1.

- Example-

$$A(x) = 6x^3 + 7x^2 - 10x + 9$$
$$B(x) = -2x^3 + 4x - 5$$

Coefficient representation of A(x) = (9, -10, 7, 6)
Coefficient representation of B(x) = (-5, 4, 0, -2)

```
Input :
 A[] = {9, -10, 7, 6}
 B[] = {-5, 4, 0, -2}
Output :
```

We can do better, if we represent the polynomial in another form.

yes

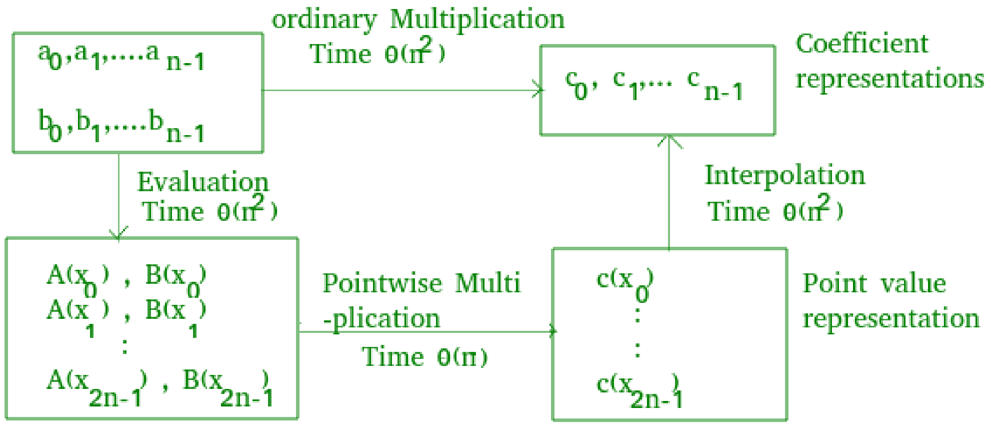
Idea is to represent polynomial in point-value form and then compute the product. A point-value representation of a polynomial A(x) of degree-bound n is a set of n point-value pairs is{ (x0, y0), (x1, y1), ..., (xn-1, yn-1)} such that all of the xi are distinct and yi = A(xi) for i = 0, 1, ..., n-1.
Example

```
xi    -- 0, 1, 2, 3
A(xi) -- 1, 0, 5, 22
```

Point-value representation of above polynomial is { (0, 1), (1, 0), (2, 5), (3, 22) }. Using Horner's method, (discussed here), n-point evaluation takes time O(n^2). It's just calculation of values of A(x) at some x for n different points, so time complexity is O(n^2). Now that the polynomial is converted into point value, it can be easily calculated C(x) = A(x)*B(x) again using horner's method. This takes O(n) time. An important point here is C(x) has degree bound 2n, then n points will give only n points of C(x), so for that case we need 2n different values of x to calculate 2n different values of y. Now that the product is calculated, the answer can to be converted back into coefficient vector form. To get back to coefficient vector form we use inverse of this evaluation. The inverse of evaluation is called interpolation. Interpolation using Lagrange's formula gives point value-form to coefficient vector form of the polynomial.Lagrange's formula is –

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x-x_j)}{\prod_{j\neq k}(x_k-x_j)}$$

So far we discussed,



This idea still solves the problem in O(n^2) time complexity. We can use any points we want as evaluation points, but by choosing the evaluation points carefully, we can convert between representations in only O(n log n) time. If we choose "complex roots of unity" as the evaluation points, we can produce a point-value representation by taking the discrete Fourier transform (DFT) of a coefficient vector. We can perform the inverse operation, interpolation, by taking the "inverse DFT" of point-value pairs, yielding a coefficient vector.

Fast Fourier Transform (FFT) can perform DFT and inverse DFT in time O(nlogn).
DFT

DFT is evaluating values of polynomial at n complex nth roots of unity [illegible] .

So, for [illegible] k = 0, 1, 2, …, n-1, y = (y0, y1, y2, …, yn-1) is Discrete fourier Transformation (DFT) of given polynomial.
The product of two polynomials of degree-bound n is a polynomial of degree-bound 2n. Before evaluating the input polynomials A and B, therefore, we first double their degree-bounds to 2n by adding n high-order coefficients of 0. Because the vectors have 2n elements, we use "complex 2nth roots of unity, " which are denoted by the W2n (omega 2n). We assume that n is a power of 2; we can always meet this requirement by adding high-order zero coefficients.
FFT
Here is the Divide-and-conquer strategy to solve this problem.

The problem of evaluating A(x) at [illegible] reduces to evaluating the degree-bound n/2 polynomials A0(x) and A1(x) at the points
[illegible]

The list [illegible] does not contain n distinct values, but n/2 complex n/2th roots of unity. Polynomials A0 and A1 are recursively evaluated at the n complex nth roots of unity. Subproblems have exactly the same form as the original problem, but are half the size. So recurrence formed is T(n) = 2T(n/2) + O(n), i.e complexity O(nlogn).

```
Algorithm
1. Add n higher-order zero coefficients to A(x) and B(x)
2. Evaluate A(x) and B(x) using FFT for 2n points
3. Pointwise multiplication of point-value forms
4. Interpolate C(x) using FFT to compute inverse DFT
```

Pseudo code of recursive FFT

```
Recursive_FFT(a){
n = length(a) // a is the input coefficient vector
if n = 1
  then return a

// wn is principle complex nth root of unity.
wn = e^(2*pi*i/n)
w = 1

// even indexed coefficients
A0 = (a0, a2, ..., an-2 )

// odd indexed coefficients
A1 = (a1, a3, ..., an-1 )

y0 = Recursive_FFT(A0) // local array
y1 = Recursive-FFT(A1) // local array

for k = 0 to n/2 - 1

  // y array stores values of the DFT
  // of given polynomial.
  do y[k] = y0[k] + w*y1[k]
     y[k+(n/2)] = y0[k] - w*y1[k]
     w = w*wn
return y
}
Recursion Tree of Above Execution-
```

Why does this work ?

[illegible]

since,

Thus, the vector y returned by Recursive-FFT is indeed the DFT of the input vector a.

```
 #include <bits/stdc++.h>
using namespace std;

// For storing complex values of nth roots
// of unity we use complex<double>
typedef complex<double> cd;

// Recursive function of FFT
vector<cd> fft(vector<cd>& a)
{
    int n = a.size();

    // if input contains just one element
    if (n == 1)
        return vector<cd>(1, a[0]);

    // For storing n complex nth roots of unity
    vector<cd> w(n);
    for (int i = 0; i < n; i++) {
        double alpha = 2 * M_PI * i / n;
        w[i] = cd(cos(alpha), sin(alpha));
    }

    vector<cd> A0(n / 2), A1(n / 2);
    for (int i = 0; i < n / 2; i++) {

        // even indexed coefficients
        A0[i] = a[i * 2];

        // odd indexed coefficients
        A1[i] = a[i * 2 + 1];
    }

    // Recursive call for even indexed coefficients
    vector<cd> y0 = fft(A0);

    // Recursive call for odd indexed coefficients
    vector<cd> y1 = fft(A1);

    // for storing values of y0, y1, y2, ..., yn-1.
    vector<cd> y(n);

    for (int k = 0; k < n / 2; k++) {
        y[k] = y0[k] + w[k] * y1[k];
        y[k + n / 2] = y0[k] - w[k] * y1[k];
    }
    return y;
}
```

```
// Driver code
int main()
{
    vector<cd> a{1, 2, 3, 4};
    vector<cd> b = fft(a);
    for (int i = 0; i < 4; i++)
        cout << b[i] << endl;

    return 0;
}
```

```
Input:  1 2 3 4
Output:
(10, 0)
(-2, -2)
(-2, 0)
(-2, 2)
```

Interpolation
Switch the roles of a and y.
Replace wn by wn^-1.
Divide each element of the result by n.
Time Complexity : O(nlogn).

Source

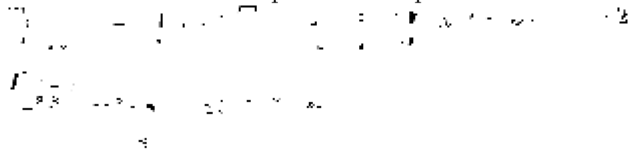
https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-multiplication/

Chapter 23

Find Nth term (A matrix exponentiation example)

Find Nth term (A matrix exponentiation example) - GeeksforGeeks

We are given a recursive function that describes Nth terms in form of other terms. In this article we have taken specific example.



Now you are given n, and you have to find out nth term using above formula.

Examples:

```
Input : n = 2
Output : 5

Input : n = 3
Output :13
```

Prerequisite :

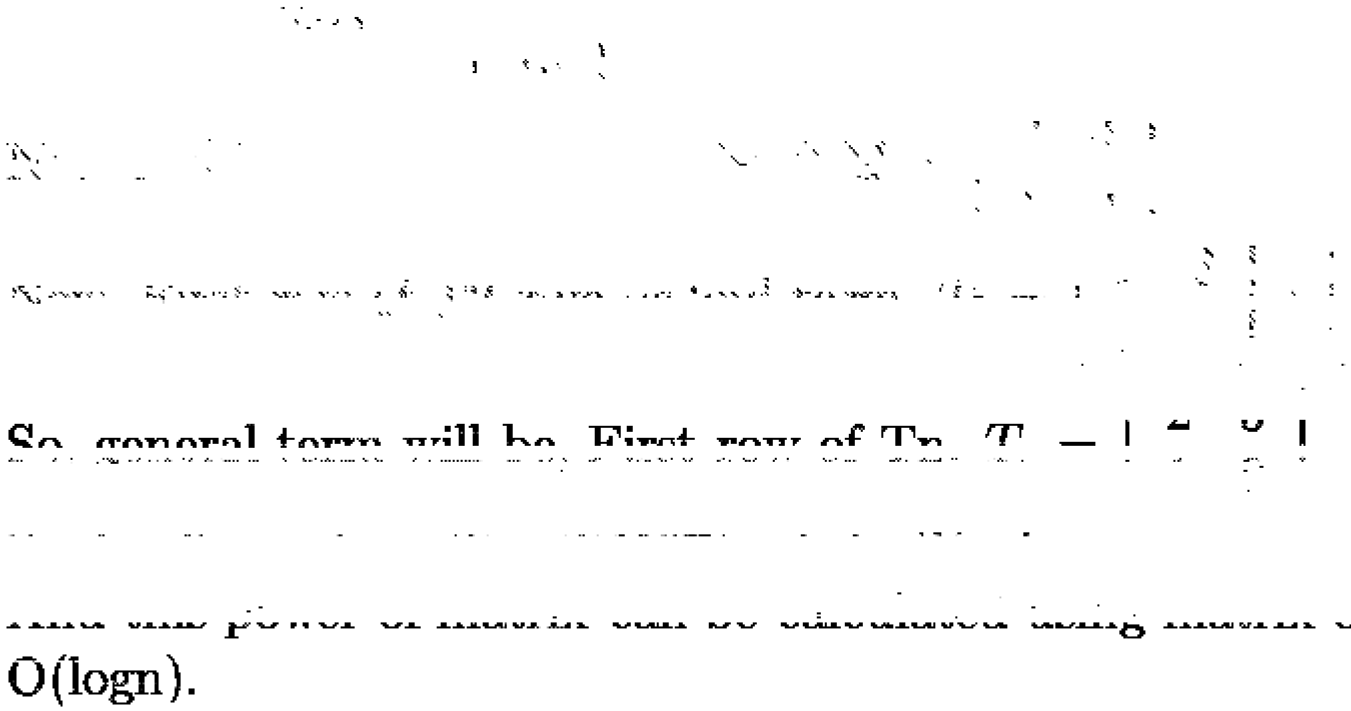
Basic Approach: This problem can be solved by simply just iterating over the n terms. Every time you find a term, using this term find next one and so on. But time complexity of this problem is of order O(n).

Optimized Approach
All such problem where a term is a function of other terms in linear fashion. Then these can be solved using Matrix (Please refer : Matrix Exponentiation). First we make transformation matrix and then just use matrix exponentiation to find Nth term.

Step by Step method includes:
Step 1. Determine k the number of terms on which T(i) depends.
In our example T(i) depends on two terms.so, k = 2

Step 2. Determine initial values
As in this article T0=1, T1=1 are given.

Step 3. Determine TM, the transformation matrix.
This is the most important step in solving recurrence relation. In this step, we have to make matrix of dimension k*k.
Such that
T(i)=TM*(initial value vector)
Here **initial value vector** is vector that contains intial value.we name this vector as **initial**.



[illegible]

So, general term will be First row of Tn. T = [illegible]

[illegible]

O(logn).

Below is C++ program to implement above approach

```
 // CPP program to find n-th term of a recursive
// function using matrix exponentiation.
#include <bits/stdc++.h>
using namespace std;
#define MOD 1000000009

#define ll long long int


ll power(ll n)
```

```
{
    if (n <= 1)
      return 1;

    // This power function returns first row of
    // {Transformation Matrix}^n-1*Initial Vector
    n--;

    // This is an identity matrix.
    ll res[2][2] = { 1, 0, 0, 1 };

    // this is Transformation matrix.
    ll tMat[2][2] = { 2, 3, 1, 0 };

    // Matrix exponentiation to calculate power of {tMat}^n-1
    // store res in "res" matrix.
    while (n) {

        if (n & 1) {
            ll tmp[2][2];
            tmp[0][0] = (res[0][0] * tMat[0][0] +
                         res[0][1] * tMat[1][0]) % MOD;
            tmp[0][1] = (res[0][0] * tMat[0][1] +
                         res[0][1] * tMat[1][1]) % MOD;
            tmp[1][0] = (res[1][0] * tMat[0][0] +
                         res[1][1] * tMat[1][0]) % MOD;
            tmp[1][1] = (res[1][0] * tMat[0][1] +
                         res[1][1] * tMat[1][1]) % MOD;
            res[0][0] = tmp[0][0];
            res[0][1] = tmp[0][1];
            res[1][0] = tmp[1][0];
            res[1][1] = tmp[1][1];
        }
        n = n / 2;
        ll tmp[2][2];
        tmp[0][0] = (tMat[0][0] * tMat[0][0] +
                     tMat[0][1] * tMat[1][0]) % MOD;
        tmp[0][1] = (tMat[0][0] * tMat[0][1] +
                     tMat[0][1] * tMat[1][1]) % MOD;
        tmp[1][0] = (tMat[1][0] * tMat[0][0] +
                     tMat[1][1] * tMat[1][0]) % MOD;
        tmp[1][1] = (tMat[1][0] * tMat[0][1] +
                     tMat[1][1] * tMat[1][1]) % MOD;
        tMat[0][0] = tmp[0][0];
        tMat[0][1] = tmp[0][1];
        tMat[1][0] = tmp[1][0];
        tMat[1][1] = tmp[1][1];
    }
    // res store {Transformation matrix}^n-1
    // hence will be first row of res*Initial Vector.
    return (res[0][0] * 1 + res[0][1] * 1) % MOD;
}

// Driver code
int main()
{
    ll n = 3;
    cout << power(n);
    return 0;
}
```

Output:

```
13
```

Time Complexity : O(Log n)

The same idea is used to find n-th Fibonacci number in O(Log n)

Source

https://www.geeksforgeeks.org/find-nth-term-a-matrix-exponentiation-example/

Chapter 24

Find a Fixed Point (Value equal to index) in a given array

Find a Fixed Point (Value equal to index) in a given array - GeeksforGeeks

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
```

Method 1 (Linear Search)
Linearly search for an index i such that arr[i] == i. Return the first such index found. Thanks to pm for suggesting this solution.

C/C++

```
 // C/C++ program to check fixed point
// in an array using linear search
#include<stdio.h>
```

```
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;
}
```

Java

```
 // Java program to check fixed point
// in an array using linear search

class Main
{
    static 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;
    }
    //main function
    public static void main(String args[])
    {
        int arr[] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
        int n = arr.length;
        System.out.println("Fixed Point is "
                     + linearSearch(arr, n));
    }
}
```

Python

```
 # Python program to check fixed point
# in an array using linear search
def linearSearch(arr, n):
    for i in range(n):
        if arr[i] is i:
            return i
    # If no fixed point present then return -1
    return -1

# Driver program to check above functions
arr = [-10, -1, 0, 3, 10, 11, 30, 50, 100]
n = len(arr)
print("Fixed Point is " + str(linearSearch(arr,n)))

# This code is contributed by Pratik Chhajer
```

C#

```
 // C# program to check fixed point
// in an array using linear search
using System;

class GFG
{
    static 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 code
    public static void Main()
    {
        int []arr = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
        int n = arr.Length;
        Console.Write("Fixed Point is "+ linearSearch(arr, n));
    }
}

// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP program to check fixed point
// in an array using linear search

function linearSearch($arr, $n)
{
    for($i = 0; $i < $n; $i++)
    {
        if($arr[$i] == $i)
            return $i;
    }

    // If no fixed point present then
    // return -1
    return -1;
}

    // Driver Code
    $arr = array(-10, -1, 0, 3, 10,
                  11, 30, 50, 100);
    $n = count($arr);
    echo "Fixed Point is ".
            linearSearch($arr,$n);

// This code is contributed by Sam007
?>
```

Output:

```
Fixed Point is 3
```

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.

C/C++

```
 // C/C++ program to check fixed point
// in an array using binary search
#include<stdio.h>

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;
}
```

Java

```
 // Java program to check fixed point
// in an array using binary search

class Main
{
    static int binarySearch(int arr[], int low, int high)
    {
        if(high >= low)
        {
            /* low + (high - low)/2; */
            int mid = (low + high)/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;
    }

    //main function
    public static void main(String args[])
    {
        int arr[] = {-10, -1, 0, 3 , 10, 11, 30, 50, 100};
        int n = arr.length;
        System.out.println("Fixed Point is "
                    + binarySearch(arr,0, n-1));
    }
}
```

Python

```
 # Python program to check fixed point
# in an array using binary search
def binarySearch(arr, low, high):
    if high >= low:
        mid = (low + high)//2

    if mid is 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 */
```

```
arr = [-10, -1, 0, 3, 10, 11, 30, 50, 100]
n = len(arr)
print("Fixed Point is " + str(binarySearch(arr, 0, n-1)))


# This code is contributed by Pratik Chhajer
```

C#

```
 // C# program to check fixed point
// in an array using binary search
using System;

class GFG
{
    static int binarySearch(int []arr, int low, int high)
    {
        if(high >= low)
        {
            // low + (high - low)/2;
            int mid = (low + high)/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 code
    public static void Main()
    {
        int []arr = {-10, -1, 0, 3 , 10, 11, 30, 50, 100};
        int n = arr.Length;
        Console.Write("Fixed Point is "+ binarySearch(arr,0, n-1));
    }
}
// This code is contributed by Sam007
```

PHP

```
 <?php
```

```
// PHP program to check fixed po
// in an array using binary search

function binarySearch($arr, $low, $high)
{
    if($high >= $low)
    {

         /*low + (high - low)/2;*/
        $mid = (int)(($low + $high) / 2);
        if($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 Po*/
    return -1;
}

    // Driver Code
    $arr = array(-10, -1, 0, 3, 10,
                  11, 30, 50, 100);
    $n = count($arr);
    echo "Fixed Point is: "
        . binarySearch($arr, 0, $n - 1);

// This code is contributed by Anuj_67
?>
```

Output:

```
Fixed Point is 3
```

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

Find a Fixed Point (Value equal to index) in a given array | Duplicates Allowed

Improved By : Sam007, vt_m

Source

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

Chapter 25

Find a peak element

Find a peak element - GeeksforGeeks

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 Conquerto find a peak in O(Logn) time. The idea is Binary Search based, we compare middle element with its neighbors. If middle element is not smaller than any of its neighbors, then we return it. If the middle element is smaller than the its left neighbor, then there is always a peak in left half (Why? take few examples). If the middle element is smaller than the its right neighbor, then there is always a peak in right half (due to same reason as left half). Following are C and Java implementations of this approach.

C/C++

```
 // A C++ program to find a peak element element using divide and conquer
#include <stdio.h>

// A binary search based function that returns index of a peak element
```

```
int findPeakUtil(int arr[], int low, int high, int n)
{
    // Find index of middle element
    int mid = low + (high - low)/2;  /* (low + high)/2 */

    // Compare middle element with its neighbours (if neighbours exist)
    if ((mid == 0 || arr[mid-1] <= arr[mid]) &&
            (mid == n-1 || arr[mid+1] <= arr[mid]))
        return mid;

    // If middle element is not peak and its left neighbour is greater
    // than it, then left half must have a peak element
    else if (mid > 0 && arr[mid-1] > arr[mid])
        return findPeakUtil(arr, low, (mid -1), n);

    // If middle element is not peak and its right neighbour is greater
    // than it, then right half must have a peak element
    else return findPeakUtil(arr, (mid + 1), high, n);
}

// A wrapper over recursive function findPeakUtil()
int findPeak(int arr[], int n)
{
    return findPeakUtil(arr, 0, n-1, n);
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 3, 20, 4, 1, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Index of a peak point is %d", findPeak(arr, n));
    return 0;
}
```

Java

```
 // A Java program to find a peak element element using divide and conquer
import java.util.*;
import java.lang.*;
import java.io.*;

class PeakElement
{
    // A binary search based function that returns index of a peak
    // element
    static int findPeakUtil(int arr[], int low, int high, int n)
    {
        // Find index of middle element
        int mid = low + (high - low)/2;  /* (low + high)/2 */

        // Compare middle element with its neighbours (if neighbours
        // exist)
        if ((mid == 0 || arr[mid-1] <= arr[mid]) && (mid == n-1 ||
             arr[mid+1] <= arr[mid]))
            return mid;

        // If middle element is not peak and its left neighbor is
        // greater than it,then left half must have a peak element
        else if (mid > 0 && arr[mid-1] > arr[mid])
            return findPeakUtil(arr, low, (mid -1), n);

        // If middle element is not peak and its right neighbor
        // is greater than it, then right half must have a peak
        // element
        else return findPeakUtil(arr, (mid + 1), high, n);
    }

    // A wrapper over recursive function findPeakUtil()
    static int findPeak(int arr[], int n)
    {
        return findPeakUtil(arr, 0, n-1, n);
    }

    // Driver method
    public static void main (String[] args)
    {
        int arr[] = {1, 3, 20, 4, 1, 0};
        int n = arr.length;
        System.out.println("Index of a peak point is " +
                            findPeak(arr, n));
    }
}
```

Python3

```
 # A python 3 program to find a peak
# element element using divide and conquer

# A binary search based function
# that returns index of a peak element
def findPeakUtil(arr, low, high, n):

     # Find index of middle element
     # (low + high)/2
     mid = low + (high - low)/2
     mid = int(mid)

    # Compare middle element with its
    # neighbours (if neighbours exist)
    if ((mid == 0 or arr[mid - 1] <= arr[mid]) and
       (mid == n - 1 or arr[mid + 1] <= arr[mid])):
        return mid


    # If middle element is not peak and
    # its left neighbour is greater
    # than it, then left half must
    # have a peak element
    elif (mid > 0 and arr[mid - 1] > arr[mid]):
        return findPeakUtil(arr, low, (mid - 1), n)

    # If middle element is not peak and
    # its right neighbour is greater
    # than it, then right half must
    # have a peak element
    else:
        return findPeakUtil(arr, (mid + 1), high, n)


# A wrapper over recursive
# function findPeakUtil()
def findPeak(arr, n):

    return findPeakUtil(arr, 0, n - 1, n)


# Driver code
arr = [1, 3, 20, 4, 1, 0]
n = len(arr)
print("Index of a peak point is", findPeak(arr, n))

# This code is contributed by
# Smitha Dinesh Semwal
```

C#

```
 // A C# program to find
// a peak element element
// using divide and conquer
using System;

class GFG
{
```

```
// A binary search based
// function that returns
// index of a peak element
static int findPeakUtil(int []arr, int low,
                        int high, int n)
{
    // Find index of
    // middle element
    int mid = low + (high - low) / 2;

    // Compare middle element with
    // its neighbours (if neighbours
    // exist)
    if ((mid == 0 ||
         arr[mid - 1] <= arr[mid]) &&
        (mid == n - 1 ||
         arr[mid + 1] <= arr[mid]))
        return mid;

    // If middle element is not
    // peak and its left neighbor
    // is greater than it,then
    // left half must have a
    // peak element
    else if (mid > 0 &&
             arr[mid - 1] > arr[mid])
        return findPeakUtil(arr, low,
                            (mid - 1), n);

    // If middle element is not
    // peak and its right neighbor
    // is greater than it, then
    // right half must have a peak
    // element
    else return findPeakUtil(arr, (mid + 1),
                             high, n);
}

// A wrapper over recursive
// function findPeakUtil()
static int findPeak(int []arr,
                    int n)
{
    return findPeakUtil(arr, 0,
                        n - 1, n);
}
```

```
    // Driver Code
    static public void Main ()
    {
        int []arr = {1, 3, 20,
                     4, 1, 0};
        int n = arr.Length;
        Console.WriteLine("Index of a peak " +
                                   "point is " +
                         findPeak(arr, n));
    }
}

// This code is contributed by ajit
```

PHP

```
 <?php
// A PHP program to find a
// peak element element using
// divide and conquer

// A binary search based function
// that returns index of a peak
// element
function findPeakUtil($arr, $low,
                      $high, $n)
{
    // Find index of middle element
    $mid = $low + ($high - $low) / 2; // (low + high)/2

    // Compare middle element with
    // its neighbours (if neighbours exist)
    if (($mid == 0 ||
         $arr[$mid - 1] <= $arr[$mid]) &&
        ($mid == $n - 1 ||
         $arr[$mid + 1] <= $arr[$mid]))
        return $mid;

    // If middle element is not peak
    // and its left neighbour is greater
    // than it, then left half must
    // have a peak element
    else if ($mid > 0 &&
             $arr[$mid - 1] > $arr[$mid])
        return findPeakUtil($arr, $low,
                           ($mid - 1), $n);

    // If middle element is not peak
    // and its right neighbour is
    // greater than it, then right
    // half must have a peak element
    else return(findPeakUtil($arr, ($mid + 1),
                             $high, $n));
}

// A wrapper over recursive
// function findPeakUtil()
function findPeak($arr, $n)
{
    return floor(findPeakUtil($arr, 0,
                              $n - 1, $n));
}

// Driver Code
$arr = array(1, 3, 20, 4, 1, 0);
$n = sizeof($arr);
echo "Index of a peak point is ",
              findPeak($arr, $n);

// This code is contributed by ajit
?>
```

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

Related Problem:
Find local minima in an array

Improved By : jit_t

Source

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

Chapter 26

Find a peak element in a 2D array

Find a peak element in a 2D array - GeeksforGeeks

An element is a peak element if it is greater than or equal to its four neighbors, left, right, top and bottom. For example neighbors for A[i][j] are A[i-1][j], A[i+1][j], A[i][j-1] and A[i][j+1]. For corner elements, missing neighbors are considered of negative infinite value.

Examples:

```
Input : 10 20 15
        21 30 14
        7  16 32
Output : 30
30 is a peak element because all its
neighbors are smaller or equal to it.
32 can also be picked as a peak.

Input : 10 7
        11 17
Output : 17
```

Below are some facts about this problem:
1: A Diagonal adjacent is not considered as neighbor.
2: A peak element is not necessarily the maximal element.
3: More than one such elements can exist.
4: There is always a peak element. We can see this property by creating some matrices using pen and paper.

Method 1: (Brute Force)
Iterate through all the elements of Matrix and check if it is greater/equal to all its neighbors. If yes, return the element.

Time Complexity: O(rows * columns)
Auxiliary Space: O(1)

Method 2 : (Efficient)
This problem is mainly an extension of Find a peak element in 1D array. We apply similar Binary Search based solution here.

1. Consider mid column and find maximum element in it.
2. Let index of mid column be 'mid', value of maximum element in mid column be 'max' and maximum element be at 'mat[max_index][mid]'.
3. If max >= A[index][mid-1] & max >= A[index][pick+1], max is a peak, return max.
4. If max < mat[max_index][mid-1], recur for left half of matrix.
5. If max < mat[max_index][mid+1], recur for right half of matrix.

Below is the C++ implementation of above algorithm:

```
 // Finding peak element in a 2D Array.
#include<bits/stdc++.h>
using namespace std;

const int MAX = 100;

// Function to find the maximum in column 'mid'
// 'rows' is number of rows.
int findMax(int arr[][MAX], int rows, int mid, int &max)
{
    int max_index = 0;
    for (int i = 0; i < rows; i++)
    {
        if (max < arr[i][mid])
        {
            // Saving global maximum and its index
            // to check its neighbours
            max = arr[i][mid];
            max_index = i;
        }
    }
    return max_index;
}

// Function to find a peak element
int findPeakRec(int arr[][MAX], int rows, int columns,
                                              int mid)
{
    // Evaluating maximum of mid column. Note max is
    // passed by reference.
    int max = 0;
    int max_index = findMax(arr, rows, mid, max);
    // If we are on the first or last column,
    // max is a peak
    if (mid == 0 || mid == columns-1)
        return max;

    // If mid column maximum is also peak
    if (max >= arr[max_index][mid-1] &&
            max >= arr[max_index][mid+1])
        return max;

    // If max is less than its left
    if (max < arr[max_index][mid-1])
        return findPeakRec(arr, rows, columns, mid - mid/2);

    // If max is less than its left
    // if (max < arr[max_index][mid+1])
    return findPeakRec(arr, rows, columns, mid+mid/2);
}

// A wrapper over findPeakRec()
int findPeak(int arr[][MAX], int rows, int columns)
{
    return findPeakRec(arr, rows, columns, columns/2);
}

// Driver Code
int main()
{
    int arr[][MAX] = {{ 10, 8, 10, 10 },
                      { 14, 13, 12, 11 },
                      { 15, 9, 11, 21 },
                      { 16, 17, 19, 20 } };

    // Number of Columns
    int rows = 4, columns = 4;
    cout << findPeak(arr, rows, columns);
    return 0;
}
```

```
Output: 21
```

Time Complexity : O(rows * log(columns)). We recur for half number of columns. In every recursive call we linearly search for maximum in current mid column.

Auxiliary Space : O(columns/2) for Recursion Call Stack

Source:

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-201
lecture-videos/MIT6_006F11_lec01.pdf

Source

https://www.geeksforgeeks.org/find-peak-element-2d-array/

Chapter 27

Find bitonic point in given bitonic sequence

Find bitonic point in given bitonic sequence - GeeksforGeeks

You are given a bitonic sequence, the task is to find **Bitonic Point** in it. A Bitonic Sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing.

A Bitonic Point is a point in bitonic sequence before which elements are strictly increasing and after which elements are strictly decreasing. A Bitonic point doesn't exist if array is only decreasing or only increasing.

Examples :

```
Input : arr[] = {6, 7, 8, 11, 9, 5, 2, 1}
Output: 11
All elements before 11 are smaller and all
elements after 11 are greater.

Input : arr[] = {-3, -2, 4, 6, 10, 8, 7, 1}
Output: 10
```

A **simple solution** for this problem is to use linear search. Element **arr[i]** is bitonic point if both i-1'th and i+1'th both elements are less than i'th element. Time complexity for this approach is O(n).

An **efficient solution** for this problem is to use **modified** binary search.

- If **arr[mid-1]** < **arr[mid]** and **arr[mid]** > **arr[mid+1]** then we are done with bitonic point.
- If **arr[mid]** < **arr[mid+1]** then search in right sub-array, else search in left sub-array.

C++

```
 // C++ program to find bitonic point in a bitonic array.
#include<bits/stdc++.h>
using namespace std;

// Function to find bitonic point using binary search
int binarySearch(int arr[], int left, int right)
{
    if (left <= right)
    {
        int mid = (left+right)/2;

        // base condition to check if arr[mid] is
        // bitonic point or not
        if (arr[mid-1]<arr[mid] && arr[mid]>arr[mid+1])
            return mid;

        // We assume that sequence is bitonic. We go to
        // right subarray if middle point is part of
        // increasing subsequence. Else we go to left
        // subarray.
        if (arr[mid] < arr[mid+1])
            return binarySearch(arr, mid+1,right);
        else
            return binarySearch(arr, left, mid-1);
    }

    return -1;
}

// Driver program to run the case
int main()
{
    int arr[] = {6, 7, 8, 11, 9, 5, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    int index = binarySearch(arr, 1, n-2);
    if (index != -1)
       cout << arr[index];
    return 0;
}
```

Java

```
 // Java program to find bitonic
// point in a bitonic array.
```

```
import java.io.*;

class GFG
{
    // Function to find bitonic point
    // using binary search
    static int binarySearch(int arr[], int left,
                                       int right)
    {
        if (left <= right)
        {
            int mid = (left + right) / 2;

            // base condition to check if arr[mid]
            // is bitonic point or not
            if (arr[mid - 1] < arr[mid] &&
                 arr[mid] > arr[mid + 1])
                 return mid;

            // We assume that sequence is bitonic. We go to
            // right subarray if middle point is part of
            // increasing subsequence. Else we go to left
            // subarray.
            if (arr[mid] < arr[mid + 1])
                return binarySearch(arr, mid + 1, right);
            else
                return binarySearch(arr, left, mid - 1);
        }

        return -1;
    }

    // Driver program
    public static void main (String[] args)
    {
        int arr[] = {6, 7, 8, 11, 9, 5, 2, 1};
        int n = arr.length;
        int index = binarySearch(arr, 1, n - 2);
        if (index != -1)
        System.out.println ( arr[index]);

    }
}

// This code is contributed by vt_m
```

C#

```
 // C# program to find bitonic
// point in a bitonic array.
using System;

class GFG
{
    // Function to find bitonic point
    // using binary search
    static int binarySearch(int []arr, int left,
                                  int right)
    {
        if (left <= right)
        {
            int mid = (left + right) / 2;

            // base condition to check if arr[mid]
            // is bitonic point or not
            if (arr[mid - 1] < arr[mid] &&
                arr[mid] > arr[mid + 1])
                return mid;

            // We assume that sequence is bitonic. We go
            // to right subarray if middle point is part of
            // increasing subsequence. Else we go to left subarray.
            if (arr[mid] < arr[mid + 1])
                return binarySearch(arr, mid + 1, right);
            else
                return binarySearch(arr, left, mid - 1);
        }

        return -1;
    }

    // Driver program
    public static void Main ()
    {
        int []arr = {6, 7, 8, 11, 9, 5, 2, 1};
        int n = arr.Length;
        int index = binarySearch(arr, 1, n - 2);
        if (index != -1)
        Console.Write ( arr[index]);
    }
}

// This code is contributed by nitin mittal
```

PHP

```
 <?php
// PHP program to find bitonic
// point in a bitonic array.

// Function to find bitonic point
// using binary search
function binarySearch($arr, $left, $right)
{
    if ($left <= $right)
    {
        $mid = ($left + $right) / 2;

        // base condition to check if
        // arr[mid] is bitonic point
        // or not
        if ($arr[$mid - 1] < $arr[$mid] &&
            $arr[$mid] > $arr[$mid + 1])
            return $mid;

        // We assume that sequence
        // is bitonic. We go to right
        // subarray if middle point
        // is part of increasing
        // subsequence. Else we go
        // to left subarray.
        if ($arr[$mid] < $arr[$mid + 1])
            return binarySearch($arr, $mid + 1,$right);
        else
            return binarySearch($arr, $left, $mid - 1);
    }

    return -1;
}

    // Driver Code
    $arr = array(6, 7, 8, 11, 9, 5, 2, 1);
    $n = sizeof($arr);
    $index = binarySearch($arr, 1, $n-2);
    if ($index != -1)
        echo $arr[$index];

// This code is contributed by nitin mittal
?>
```

Output:

```
11
```

Time complexity : O(Log n)

Improved By : nitin mittal, sirjan13

Source

https://www.geeksforgeeks.org/find-bitonic-point-given-bitonic-sequence/

Chapter 28

Find closest number in array

Find closest number in array - GeeksforGeeks

Given an array of sorted integers. We need to find the closest value to the given number. Array may contain duplicate values and negative numbers.

Examples:

```
Input : arr[] = {1, 2, 4, 5, 6, 6, 8, 9}
             Target number = 11
Output : 9
9 is closest to 11 in given array

Input :arr[] = {2, 5, 6, 7, 8, 8, 9};
       Target number = 4
Output : 5
```

A **simple solution** is to traverse through the given array and keep track of absolute difference of current element with every element. Finally return the element that has minimum absolution difference.

An **efficient solution** is to use Binary Search.

C++

```
 // CPP program to find element
// closet to given target.
#include <iostream>
using namespace std;

int getClosest(int, int, int);

// Returns element closest to target in arr[]
```

```
int findClosest(int arr[], int n, int target)
{
    // Corner cases
    if (target <= arr[0])
        return arr[0];
    if (target >= arr[n - 1])
        return arr[n - 1];

    // Doing binary search
    int i = 0, j = n, mid = 0;
    while (i < j) {
        mid = (i + j) / 2;

        if (arr[mid] == target)
            return arr[mid];

        /* If target is less than array element,
            then search in left */
        if (target < arr[mid]) {

            // If target is greater than previous
            // to mid, return closest of two
            if (mid > 0 && target > arr[mid - 1])
                return getClosest(arr[mid - 1],
                                  arr[mid], target);

            /* Repeat for left half */
            j = mid;
        }

        // If target is greater than mid
        else {
            if (mid < n - 1 && target < arr[mid + 1])
                return getClosest(arr[mid],
                                  arr[mid + 1], target);
            // update i
            i = mid + 1;
        }
    }

    // Only single element left after search
    return arr[mid];
}

// Method to compare which one is the more close.
// We find the closest by taking the difference
// between the target and both values. It assumes
// that val2 is greater than val1 and target lies
// between these two.
int getClosest(int val1, int val2,
               int target)
{
    if (target - val1 >= val2 - target)
        return val2;
    else
        return val1;
}

// Driver code
int main()
{
    int arr[] = { 1, 2, 4, 5, 6, 6, 8, 9 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 11;
    cout << (findClosest(arr, n, target));
}

// This code is contributed bu Smitha Dinesh Semwal
```

Java

```
 // Java program to find element closet to given target.
import java.util.*;
import java.lang.*;
import java.io.*;

class FindClosestNumber {

    // Returns element closest to target in arr[]
    public static int findClosest(int arr[], int target)
    {
        int n = arr.length;

        // Corner cases
        if (target <= arr[0])
            return arr[0];
        if (target >= arr[n - 1])
            return arr[n - 1];

        // Doing binary search
        int i = 0, j = n, mid = 0;
        while (i < j) {
            mid = (i + j) / 2;

            if (arr[mid] == target)
                return arr[mid];
            /* If target is less than array element,
               then search in left */
            if (target < arr[mid]) {

                // If target is greater than previous
                // to mid, return closest of two
                if (mid > 0 && target > arr[mid - 1])
                    return getClosest(arr[mid - 1],
                                  arr[mid], target);

                /* Repeat for left half */
                j = mid;
            }

            // If target is greater than mid
            else {
                if (mid < n-1 && target < arr[mid + 1])
                    return getClosest(arr[mid],
                          arr[mid + 1], target);
                i = mid + 1; // update i
            }
        }

        // Only single element left after search
        return arr[mid];
    }

    // Method to compare which one is the more close
    // We find the closest by taking the difference
    //  between the target and both values. It assumes
    // that val2 is greater than val1 and target lies
    // between these two.
    public static int getClosest(int val1, int val2,
                                         int target)
    {
        if (target - val1 >= val2 - target)
            return val2;
        else
            return val1;
    }

    // Driver code
    public static void main(String[] args)
    {
        int arr[] = { 1, 2, 4, 5, 6, 6, 8, 9 };
        int target = 11;
        System.out.println(findClosest(arr, target));
    }
}
```

Python 3

```
 # Python3 program to find element
# closet to given target.

# Returns element closest to target in arr[]
def findClosest(arr, n, target):

    # Corner cases
    if (target <= arr[0]):
        return arr[0]
    if (target >= arr[n - 1]):
        return arr[n - 1]

    # Doing binary search
    i = 0; j = n; mid = 0
    while (i < j):
        mid = (i + j) / 2

        if (arr[mid] == target):
            return arr[mid]

        # If target is less than array
        # element, then search in left
        if (target < arr[mid]) :

            # If target is greater than previous
            # to mid, return closest of two
            if (mid > 0 and target > arr[mid - 1]):
                return getClosest(arr[mid - 1], arr[mid], target)

            # Repeat for left half
            j = mid

        # If target is greater than mid
        else :
            if (mid < n - 1 and target < arr[mid + 1]):
                return getClosest(arr[mid], arr[mid + 1], target)

            # update i
            i = mid + 1

    # Only single element left after search
    return arr[mid]
```

```
# Method to compare which one is the more close.
# We find the closest by taking the difference
# between the target and both values. It assumes
# that val2 is greater than val1 and target lies
# between these two.
def getClosest(val1, val2, target):

    if (target - val1 >= val2 - target):
        return val2
    else:
        return val1

# Driver code
arr = [1, 2, 4, 5, 6, 6, 8, 9]
n = len(arr)
target = 11
print(findClosest(arr, n, target))

# This code is contributed by Smitha Dinesh Semwal
```

C#

```
 // C# program to find element
// closet to given target.
using System;

class GFG
{

    // Returns element closest
    // to target in arr[]
    public static int findClosest(int []arr,
                                  int target)
    {
        int n = arr.Length;

        // Corner cases
        if (target <= arr[0])
            return arr[0];
        if (target >= arr[n - 1])
            return arr[n - 1];

        // Doing binary search
        int i = 0, j = n, mid = 0;
        while (i < j)
        {
            mid = (i + j) / 2;
            if (arr[mid] == target)
                return arr[mid];

            /* If target is less
            than array element,
            then search in left */
            if (target < arr[mid])
            {

                // If target is greater
                // than previous to mid,
                // return closest of two
                if (mid > 0 && target > arr[mid - 1])
                    return getClosest(arr[mid - 1],
                                 arr[mid], target);

                /* Repeat for left half */
                j = mid;
            }

            // If target is
            // greater than mid
            else
            {
                if (mid < n-1 && target < arr[mid + 1])
                    return getClosest(arr[mid],
                         arr[mid + 1], target);
                i = mid + 1; // update i
            }
        }

        // Only single element
        // left after search
        return arr[mid];
    }

    // Method to compare which one
    // is the more close We find the
    // closest by taking the difference
    // between the target and both
    // values. It assumes that val2 is
    // greater than val1 and target
    // lies between these two.
    public static int getClosest(int val1, int val2,
                                 int target)
    {
        if (target - val1 >= val2 - target)
```

```
            return val2;
        else
            return val1;
    }

    // Driver code
    public static void Main()
    {
        int []arr = {1, 2, 4, 5,
                     6, 6, 8, 9};
        int target = 11;
        Console.WriteLine(findClosest(arr, target));
    }
}

// This code is contributed by anuj_67.
```

Output:

```
9
```

Improved By : harishkumar88, vt_m

Source

https://www.geeksforgeeks.org/find-closest-number-array/

Chapter 29

Find cubic root of a number

Find cubic root of a number - GeeksforGeeks

Given a number n, find the cube root of n.

Examples:

```
Input:  n = 3
Output: Cubic Root is 1.442250

Input: n = 8
Output: Cubic Root is 2.000000
```

We can use binary search. First we define error e. Let us say 0.0000001 in our case. The main steps of our algorithm for calculating the cubic root of a number n are:

1. Initialize start = 0 and end = n
2. Calculate mid = (start + end)/2
3. Check if the absolute value of (n – mid*mid*mid) < e. If this condition holds true then mid is our answer so return mid.
4. If (mid*mid*mid)>n then set end=mid
5. If (mid*mid*mid)<n set start=mid.

Below is the implementation of above idea.

C++

```
 // C++ program to find cubic root of a number
// using Binary Search
#include <bits/stdc++.h>
using namespace std;
```

```
// Returns the absolute value of n-mid*mid*mid
double diff(double n,double mid)
{
    if (n > (mid*mid*mid))
        return (n-(mid*mid*mid));
    else
        return ((mid*mid*mid) - n);
}

// Returns cube root of a no n
double cubicRoot(double n)
{
    // Set start and end for binary search
    double start = 0, end = n;

    // Set precision
    double e = 0.0000001;

    while (true)
    {
        double mid = (start + end)/2;
        double error = diff(n, mid);

        // If error is less than e then mid is
        // our answer so return mid
        if (error <= e)
            return mid;

        // If mid*mid*mid is greater than n set
        // end = mid
        if ((mid*mid*mid) > n)
            end = mid;

        // If mid*mid*mid is less than n set
        // start = mid
        else
            start = mid;
    }
}

// Driver code
int main()
{
    double n = 3;
    printf("Cubic root of %lf is %lf\n",
           n, cubicRoot(n));
    return 0;
}
```

Java

```
 // Java program to find cubic root of a number
// using Binary Search
import java.io.*;

class GFG
{
    // Returns the absolute value of n-mid*mid*mid
    static double diff(double n,double mid)
    {
        if (n > (mid*mid*mid))
            return (n-(mid*mid*mid));
        else
            return ((mid*mid*mid) - n);
    }

    // Returns cube root of a no n
    static double cubicRoot(double n)
    {
        // Set start and end for binary search
        double start = 0, end = n;

        // Set precision
        double e = 0.0000001;

        while (true)
        {
            double mid = (start + end)/2;
            double error = diff(n, mid);

            // If error is less than e then mid is
            // our answer so return mid
            if (error <= e)
                return mid;

            // If mid*mid*mid is greater than n set
            // end = mid
            if ((mid*mid*mid) > n)
                end = mid;

            // If mid*mid*mid is less than n set
            // start = mid
            else
                start = mid;
        }
    }

    // Driver program to test above function
    public static void main (String[] args)
    {
        double n = 3;
        System.out.println("Cube root of "+n+" is "+cubicRoot(n));
    }
}

// This code is contributed by Pramod Kumar
```

Python3

```
 # Python 3 program to find cubic root
# of a number using Binary Search

# Returns the absolute value of
# n-mid*mid*mid
def diff(n, mid) :
    if (n > (mid * mid * mid)) :
        return (n - (mid * mid * mid))
    else :
        return ((mid * mid * mid) - n)

# Returns cube root of a no n
def cubicRoot(n) :

    # Set start and end for binary
    # search
    start = 0
    end = n

    # Set precision
    e = 0.0000001
    while (True) :

        mid = (start + end) / 2
        error = diff(n, mid)

        # If error is less than e
        # then mid is our answer
        # so return mid
        if (error <= e) :
            return mid

        # If mid*mid*mid is greater
        # than n set end = mid
        if ((mid * mid * mid) > n) :
            end = mid

        # If mid*mid*mid is less
        # than n set start = mid
        else :
            start = mid

# Driver code
n = 3
print("Cubic root of", n, "is",
      round(cubicRoot(n),6))


# This code is contributed by Nikita Tiwari.
```

C#

```
 // C# program to find cubic root
// of a number using Binary Search
using System;

class GFG {

    // Returns the absolute value
    // of n - mid * mid * mid
    static double diff(double n, double mid)
    {
        if (n > (mid * mid * mid))
            return (n-(mid * mid * mid));
        else
            return ((mid * mid * mid) - n);
    }

    // Returns cube root of a no. n
    static double cubicRoot(double n)
    {

        // Set start and end for
        // binary search
        double start = 0, end = n;

        // Set precision
        double e = 0.0000001;

        while (true)
        {
            double mid = (start + end) / 2;
            double error = diff(n, mid);

            // If error is less than e then
            // mid is our answer so return mid
            if (error <= e)
                return mid;

            // If mid * mid * mid is greater
            // than n set end = mid
            if ((mid * mid * mid) > n)
                end = mid;

            // If mid*mid*mid is less than
            // n set start = mid
            else
                start = mid;
        }
    }

    // Driver Code
    public static void Main ()
    {
        double n = 3;
        Console.Write("Cube root of "+ n
                       + " is "+cubicRoot(n));
    }
}

// This code is contributed by nitin mittal.
```

PHP

```
 <?php
// PHP program to find cubic root
// of a number using Binary Search

// Returns the absolute value
// of n - mid * mid * mid
function diff($n,$mid)
{
    if ($n > ($mid * $mid * $mid))
        return ($n - ($mid *
                $mid * $mid));
    else
        return (($mid * $mid *
                 $mid) - $n);
}
```

```
// Returns cube root of a no n
function cubicRoot($n)
{

    // Set start and end
    // for binary search
    $start = 0;
    $end = $n;

    // Set precision
    $e = 0.0000001;

    while (true)
    {
        $mid = (($start + $end)/2);
        $error = diff($n, $mid);

        // If error is less
        // than e then mid is
        // our answer so return mid
        if ($error <= $e)
            return $mid;

        // If mid*mid*mid is
        // greater than n set
        // end = mid
        if (($mid * $mid * $mid) > $n)
            $end = $mid;

        // If mid*mid*mid is
        // less than n set
        // start = mid
        else
            $start = $mid;
    }
}

    // Driver Code
    $n = 3;
    echo("Cubic root of $n is ");
    echo(cubicRoot($n));

// This code is contributed by nitin mittal.
?>
```

Output:

```
Cubic root of 3.000000 is 1.442250
```

Time Complexity : O(Log n)

Improved By : nitin mittal

Source

https://www.geeksforgeeks.org/find-cubic-root-of-a-number/

Chapter 30

Find frequency of each element in a limited range array in less than O(n) time

Find frequency of each element in a limited range array in less than O(n) time - GeeksforGeeks

Given an sorted array of positive integers, count number of occurrences for each element in the array. Assume all elements in the array are less than some constant M.

Do this without traversing the complete array. i.e. expected time complexity is less than O(n).

Examples:

```
Input: arr[] = [1, 1, 1, 2, 3, 3, 5,
               5, 8, 8, 8, 9, 9, 10]
Output:
Element 1 occurs 3 times
Element 2 occurs 1 times
Element 3 occurs 2 times
Element 5 occurs 2 times
Element 8 occurs 3 times
Element 9 occurs 2 times
Element 10 occurs 1 times
```

Method 1 (Linear Search)
The idea is traverse the input array and for each distinct element of array, store its frequency in a map and finally print the map. This approach takes O(n) time.

Method 2 (Use Binary Search)
This problem can be solved in less than O(n) using a modified binary search. The idea is

to recursively divide the array into two equal subarrays if its end elements are different. If both its end elements are same, that means that all elements in the subarray is also same as the array is already sorted. We then simply increment the count of the element by size of the subarray.

The time complexity of above approach is O(m log n), where m is number of distinct elements in the array of size n. Since m <= M (a constant), the time complexity of this solution is O(log n).

Below is C++ implementation of above idea –

```
 // C++ program to count number of occurrences of
// each element in the array in less than O(n) time
#include <iostream>
#include <vector>
using namespace std;

// A recursive function to count number of occurrences
// for each element in the array without traversing
// the whole array
void findFrequencyUtil(int arr[], int low, int high,
                        vector<int>& freq)
{
    // If element at index low is equal to element
    // at index high in the array
    if (arr[low] == arr[high])
    {
        // increment the frequency of the element
        // by count of elements between high and low
        freq[arr[low]] += high - low + 1;
    }
    else
    {
        // Find mid and recurse for left and right
        // subarray
        int mid = (low + high) / 2;
        findFrequencyUtil(arr, low, mid, freq);
        findFrequencyUtil(arr, mid + 1, high, freq);
    }
}

// A wrapper over recursive function
// findFrequencyUtil(). It print number of
// occurrences of each element in the array.
void findFrequency(int arr[], int n)
{
    // create a empty vector to store frequencies
    // and initialize it by 0. Size of vector is
    // maximum value (which is last value in sorted
    // array) plus 1.
    vector<int> freq(arr[n - 1] + 1, 0);

    // Fill the vector with frequency
    findFrequencyUtil(arr, 0, n - 1, freq);

    // Print the frequencies
    for (int i = 0; i <= arr[n - 1]; i++)
        if (freq[i] != 0)
            cout << "Element " << i << " occurs "
                 << freq[i] << " times" << endl;
}

// Driver function
int main()
{
    int arr[] = { 1, 1, 1, 2, 3, 3, 5, 5,
                  8, 8, 8, 9, 9, 10 };
    int n = sizeof(arr) / sizeof(arr[0]);

    findFrequency(arr, n);

    return 0;
}
```

Output:

```
Element 1 occurs 3 times
Element 2 occurs 1 times
Element 3 occurs 2 times
Element 5 occurs 2 times
Element 8 occurs 3 times
Element 9 occurs 2 times
Element 10 occurs 1 times
```

Source

https://www.geeksforgeeks.org/find-frequency-of-each-element-in-a-limited-range-array-in-less-than-on-time/

Chapter 31

Find index of an extra element present in one sorted array

Find index of an extra element present in one sorted array - GeeksforGeeks

Given two sorted arrays. There is only 1 difference between the arrays. First array has one element extra added in between. Find the index of the extra element.

Examples :

```
Input : {2, 4, 6, 8, 9, 10, 12};
        {2, 4, 6, 8, 10, 12};
Output : 4
The first array has an extra element 9.
The extra element is present at index 4.

Input :  {3, 5, 7, 9, 11, 13}
         {3, 5, 7, 11, 13}
Output :  3
```

Method 1 (Basic)

The basic method is to iterate through the whole second array and check element by element if they are different.

C++

```
 // C++ program to find an extra
// element present in arr1[]
#include <iostream>
using namespace std;
```

```
// Returns index of extra element
// in arr1[]. n is size of arr2[].
// Size of arr1[] is n-1.
int findExtra(int arr1[],
              int arr2[], int n)
{
for (int i = 0; i < n; i++)
    if (arr1[i] != arr2[i])
        return i;

return n;
}

// Driver code
int main()
{
    int arr1[] = {2, 4, 6, 8,
                  10, 12, 13};
    int arr2[] = {2, 4, 6,
                  8, 10, 12};
    int n = sizeof(arr2) / sizeof(arr2[0]);

    // Solve is passed both arrays
    cout << findExtra(arr1, arr2, n);
    return 0;
}
```

Java

```
 // Java program to find an extra
// element present in arr1[]
class GFG
{

    // Returns index of extra element
    // in arr1[]. n is size of arr2[].
    // Size of arr1[] is n-1.
    static int findExtra(int arr1[],
                         int arr2[], int n)
    {
    for (int i = 0; i < n; i++)
        if (arr1[i] != arr2[i])
            return i;

    return n;
    }

    // Driver Code
    public static void main (String[] args)
    {
        int arr1[] = {2, 4, 6, 8,
                      10, 12, 13};
        int arr2[] = {2, 4, 6,
                      8, 10, 12};
        int n = arr2.length;

        // Solve is passed both arrays
        System.out.println(findExtra(arr1,
                                     arr2, n));
    }
}

// This code is contributed by Harsh Agarwal
```

Python3

```
 # Python 3 program to find an
# extra element present in arr1[]


# Returns index of extra .
# element in arr1[] n is
# size of arr2[]. Size of
# arr1[] is n-1.
def findExtra(arr1, arr2, n) :
    for i in range(0, n) :
        if (arr1[i] != arr2[i]) :
            return i

    return n


# Driver code
arr1 = [2, 4, 6, 8,  10, 12, 13]
arr2 = [2, 4, 6, 8, 10, 12]
n = len(arr2)

# Solve is passed both arrays
print(findExtra(arr1, arr2, n))

# This code is contributed
# by Nikita Tiwari.
```

C#

```
 // C# program to find an extra
```

```
// element present in arr1[]
using System;

class GfG
{

    // Returns index of extra
    // element in arr1[]. n is
    // size of arr2[]. Size of
    // arr1[] is n-1.
    static int findExtra(int []arr1,
                         int []arr2, int n)
    {
        for (int i = 0; i < n; i++)
            if (arr1[i] != arr2[i])
                return i;

        return n;
    }

    // Driver code
    public static void Main ()
    {
        int []arr1 = {2, 4, 6, 8,
                      10, 12, 13};
        int []arr2 = {2, 4, 6,
                      8, 10, 12};
        int n = arr2.Length;

        // Solve is passed both arrays
        Console.Write(findExtra(arr1, arr2, n));
    }
}

// This code is contributed by parashar.
```

PHP

```
 <?php
// PHP program to find an extra
// element present in arr1[]

// Returns index of extra element
// in arr1[]. n is size of arr2[].
// Size of arr1[] is n-1.
function findExtra($arr1,
                   $arr2, $n)
{
```

```
for ($i = 0; $i < $n; $i++)
    if ($arr1[$i] != $arr2[$i])
        return $i;

return $n;
}

// Driver code
$arr1 = array (2, 4, 6, 8,
               10, 12, 13);
$arr2 = array(2, 4, 6,
              8, 10, 12);
$n = sizeof($arr2);

// Solve is passed
// both arrays
echo findExtra($arr1, $arr2, $n);

// This code is contributed by ajit
?>
```

Output :

```
 6
```

Time complexity : O(n)

Method 2 (Using Binary search)

We use binary search to check whether the same indices elements are different & reduce our search by a factor of 2 in each step.

C++

```
 // C++ program to find an extra
// element present in arr1[]
#include <iostream>
using namespace std;

// Returns index of extra element
// in arr1[]. n is size of arr2[].
// Size of arr1[] is n-1.
int findExtra(int arr1[],
              int arr2[], int n)
{
    // Initialize result
    int index = n;

    // left and right are end
    // points denoting the current range.
    int left = 0, right = n - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;

        // If middle element is same
        // of both arrays, it means
        // that extra element is after
        // mid so we update left to mid+1
        if (arr2[mid] == arr1[mid])
            left = mid + 1;

        // If middle element is different
        // of the arrays, it means that
        // the index we are searching for
        // is either mid, or before mid.
        // Hence we update right to mid-1.
        else
        {
            index = mid;
            right = mid - 1;
        }
    }

    // when right is greater than
    // left our search is complete.
    return index;
}

// Driver code
int main()
{
    int arr1[] = {2, 4, 6, 8, 10, 12, 13};
    int arr2[] = {2, 4, 6, 8, 10, 12};
    int n = sizeof(arr2) / sizeof(arr2[0]);

    // Solve is passed both arrays
    cout << findExtra(arr1, arr2, n);
    return 0;
}
```

Java

```
 // Java program to find an extra
// element present in arr1[]
class GFG
{
    // Returns index of extra element
    // in arr1[]. n is size of arr2[].
    // Size of arr1[] is n-1.
    static int findExtra(int arr1[],
                         int arr2[], int n)
    {
        // Initialize result
        int index = n;

        // left and right are end
        // points denoting the current range.
        int left = 0, right = n - 1;
        while (left <= right)
        {
            int mid = (left+right) / 2;

            // If middle element is same
            // of both arrays, it means
            // that extra element is after
            // mid so we update left to mid+1
            if (arr2[mid] == arr1[mid])
                left = mid + 1;

            // If middle element is different
            // of the arrays, it means that
            // the index we are searching for
            // is either mid, or before mid.
            // Hence we update right to mid-1.
            else
            {
                index = mid;
                right = mid - 1;
            }
        }

        // when right is greater than
        // left, our search is complete.
        return index;
    }

    // Driver Code
    public static void main (String[] args)
    {
        int arr1[] = {2, 4, 6, 8, 10, 12,13};
        int arr2[] = {2, 4, 6, 8, 10, 12};
        int n = arr2.length;

        // Solve is passed both arrays
        System.out.println(findExtra(arr1, arr2, n));
    }
}

// This code is contributed by Harsh Agarwal
```

Python3

```
 # Python3 program to find an extra
# element present in arr1[]

# Returns index of extra element
# in arr1[]. n is size of arr2[].
# Size of arr1[] is n-1.
def findExtra(arr1, arr2, n) :

    index = n # Initialize result

    # left and right are end points
    # denoting the current range.
    left = 0
    right = n - 1
    while (left <= right) :
        mid = (int)((left + right) / 2)

        # If middle element is same
        # of both arrays, it means
        # that extra element is after
        # mid so we update left to
        # mid + 1
        if (arr2[mid] == arr1[mid]) :
            left = mid + 1

        # If middle element is different
        # of the arrays, it means that
        # the index we are searching for
        # is either mid, or before mid.
        # Hence we update right to mid-1.
        else :
            index = mid
            right = mid - 1

    # when right is greater than left our
    # search is complete.
    return index

# Driver code
arr1 = [2, 4, 6, 8, 10, 12, 13]
arr2 = [2, 4, 6, 8, 10, 12]
n = len(arr2)

# Solve is passed both arrays
print(findExtra(arr1, arr2, n))

# This code is contributed by Nikita Tiwari.
```

C#

```
 // C# program to find an extra
// element present in arr1[]
using System;

class GFG {

    // Returns index of extra
    // element in arr1[]. n is
    // size of arr2[].
    // Size of arr1[] is
    // n - 1.
    static int findExtra(int []arr1,
                         int []arr2,
                         int n)
    {

        // Initialize result
        int index = n;

        // left and right are
        // end points denoting
        // the current range.
        int left = 0, right = n - 1;
        while (left <= right)
        {
            int mid = (left+right) / 2;

            // If middle element is
            // same of both arrays,
            // it means that extra
            // element is after mid
            // so we update left
            // to mid + 1
            if (arr2[mid] == arr1[mid])
```

```
                left = mid + 1;

            // If middle element is
            // different of the arrays,
            // it means that the index
            // we are searching for is
            // either mid, or before mid.
            // Hence we update right to mid-1.
            else
            {
                index = mid;
                right = mid - 1;
            }
        }

        // when right is greater
        // than left our
        // search is complete.
        return index;
    }

    // Driver Code
    public static void Main ()
    {
        int []arr1 = {2, 4, 6, 8, 10, 12,13};
        int []arr2 = {2, 4, 6, 8, 10, 12};
        int n = arr2.Length;

        // Solve is passed
        // both arrays
        Console.Write(findExtra(arr1, arr2, n));
    }
}

// This code is contributed by nitin mittal.
```

PHP

```
 <?php
// PHP program to find an extra
// element present in arr1[]

// Returns index of extra element
// in arr1[]. n is size of arr2[].
// Size of arr1[] is n-1.
function findExtra($arr1, $arr2, $n)
{
    // Initialize result
    $index = $n;

    // left and right are
    // end points denoting
    // the current range.
    $left = 0; $right = $n - 1;
    while ($left <= $right)
    {
        $mid = ($left+$right) / 2;

        // If middle element is same
        // of both arrays, it means
        // that extra element is after
        // mid so we update left to mid+1
        if ($arr2[$mid] == $arr1[$mid])
            $left = $mid + 1;

        // If middle element is different
        // of the arrays, it means that the
        // index we are searching for is either
        // mid, or before mid. Hence we update
        // right to mid-1.
        else
        {
            $index = $mid;
            $right = $mid - 1;
        }
    }

    // when right is greater than
    // left, our search is complete.
    return $index;
}

// Driver code
{
    $arr1 = array(2, 4, 6, 8,
                  10, 12, 13);
    $arr2 = array(2, 4, 6,
                  8, 10, 12);
    $n = sizeof($arr2) / sizeof($arr2[0]);

    // Solve is passed both arrays
    echo findExtra($arr1, $arr2, $n);
    return 0;
}

// This code is contributed by nitin mittal
```

```
?>
```

Output :

```
6
```

Time complexity : O(log n)

Improved By : parashar, nitin mittal, jit_t

Source

https://www.geeksforgeeks.org/find-index-of-an-extra-element-present-in-one-sorted-array/

Chapter 32

Find the Missing Number in a sorted array

Find the Missing Number in a sorted array - GeeksforGeeks

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.

Examples:

```
Input : arr[] = [1, 2, 3, 4, 6, 7, 8]
Output : 5

Input : arr[] = [1, 2, 3, 4, 5, 6, 8, 9]
Output : 7
```

One **Simple solution** is to apply methods discussed for finding the missing element in an unsorted array. Time complexity of this solution is O(n).

An **efficient solution** is based on the divide and conquer algorithm that we have seen in binary search, the concept behind this solution is that the elements appearing before the missing element will have ar[i] – i = 1 and those appearing after the missing element will have ar[i] – i = 2.

This solution has a time complexity of O(log n)

C++

```
 // A binary search based program to find the
// only missing number in a sorted array of
// distinct elements within limited range.
```

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

int search(int ar[], int size)
{
    int a = 0, b = size - 1;
    int mid;
    while ((b - a) > 1) {
        mid = (a + b) / 2;
        if ((ar[a] - a) != (ar[mid] - mid))
            b = mid;
        else if ((ar[b] - b) != (ar[mid] - mid))
            a = mid;
    }
    return (ar[mid] + 1);
}

int main()
{
    int ar[] = { 1, 2, 3, 4, 5, 6, 8 };
    int size = sizeof(ar) / sizeof(ar[0]);
    cout << "Missing number:" << search(ar, size);
}
```

Java

```
 // A binary search based program
// to find the only missing number
// in a sorted array of distinct
// elements within limited range.
import java.io.*;

class GFG
{
static int search(int ar[],
                  int size)
{
    int a = 0, b = size - 1;
    int mid = 0;
    while ((b - a) > 1)
    {
        mid = (a + b) / 2;
        if ((ar[a] - a) != (ar[mid] - mid))
            b = mid;
        else if ((ar[b] - b) != (ar[mid] - mid))
            a = mid;
    }
    return (ar[mid] + 1);
}

// Driver Code
public static void main (String[] args)
{
    int ar[] = { 1, 2, 3, 4, 5, 6, 8 };
    int size = ar.length;
    System.out.println("Missing number: " +
                        search(ar, size));
}
}

// This code is contributed
// by inder_verma.
```

C#

```
 // A binary search based program
// to find the only missing number
// in a sorted array of distinct
// elements within limited range.
using System;

class GFG
{
static int search(int []ar,
                  int size)
{
    int a = 0, b = size - 1;
    int mid = 0;
    while ((b - a) > 1)
    {
        mid = (a + b) / 2;
        if ((ar[a] - a) != (ar[mid] - mid))
            b = mid;
        else if ((ar[b] - b) != (ar[mid] - mid))
            a = mid;
    }
    return (ar[mid] + 1);
}

// Driver Code
static public void Main (String []args)
{
    int []ar = { 1, 2, 3, 4, 5, 6, 8 };
    int size = ar.Length;
    Console.WriteLine("Missing number: " +
                        search(ar, size));
}
}

// This code is contributed
// by Arnab Kundu
```

Output:

```
Missing number: 7
```

Improved By : inderDuMCA, andrew1234

Source

https://www.geeksforgeeks.org/find-the-missing-number-in-a-sorted-array/

Chapter 33

Find the Rotation Count in Rotated Sorted array

Find the Rotation Count in Rotated Sorted array - GeeksforGeeks

Consider an array of distinct numbers sorted in increasing order. The array has been rotated (clockwise) k number of times. Given such an array, find the value of k.

Examples:

```
Input : arr[] = {15, 18, 2, 3, 6, 12}
Output: 2
Explanation : Initial array must be {2, 3,
6, 12, 15, 18}. We get the given array after
rotating the initial array twice.

Input : arr[] = {7, 9, 11, 12, 5}
Output: 4

Input: arr[] = {7, 9, 11, 12, 15};
Output: 0
```

Method 1 (Using linear search)

If we take closer look at examples, we can notice that the number of rotations is equal to index of minimum element. A simple linear solution is to find minimum element and returns its index. Below is C++ implementation of the idea.

C++

```
 // C++ program to find number of rotations
// in a sorted and rotated array.
```

```
#include<bits/stdc++.h>
using namespace std;

// Returns count of rotations for an array which
// is first sorted in ascending order, then rotated
int countRotations(int arr[], int n)
{
    // We basically find index of minimum
    // element
    int min = arr[0], min_index;
    for (int i=0; i<n; i++)
    {
        if (min > arr[i])
        {
            min = arr[i];
            min_index = i;
        }
    }
    return min_index;
}

// Driver code
int main()
{
    int arr[] = {15, 18, 2, 3, 6, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << countRotations(arr, n);
    return 0;
}
```

Java

```
 // Java program to find number of
// rotations in a sorted and rotated
// array.
import java.util.*;
import java.lang.*;
import java.io.*;

class LinearSearch
{
    // Returns count of rotations for an
    // array which is first sorted in
    // ascending order, then rotated
    static int countRotations(int arr[], int n)
    {
        // We basically find index of minimum
        // element
        int min = arr[0], min_index = -1;
        for (int i = 0; i < n; i++)
        {
            if (min > arr[i])
            {
                min = arr[i];
                min_index = i;
            }
        }
        return min_index;
    }

    // Driver program to test above functions
    public static void main (String[] args)
    {
        int arr[] = {15, 18, 2, 3, 6, 12};
        int n = arr.length;

        System.out.println(countRotations(arr, n));
    }
}
// This code is contributed by Chhavi
```

Python3

```
 # Python3 program to find number
# of rotations in a sorted and
# rotated array.

# Returns count of rotations for
# an array which is first sorted
# in ascending order, then rotated
def countRotations(arr, n):

    # We basically find index
    # of minimum element
    min = arr[0]
    for i in range(0, n):

        if (min > arr[i]):

            min = arr[i]
            min_index = i

    return min_index;


# Driver code
```

```
arr = [15, 18, 2, 3, 6, 12]
n = len(arr)
print(countRotations(arr, n))

# This code is contributed by Smitha Dinesh Semwal
```

C#

```
 // c# program to find number of
// rotations in a sorted and rotated
// array.
using System;

class LinearSearch
{
    // Returns count of rotations for an
    // array which is first sorted in
    // ascending order, then rotated
    static int countRotations(int []arr, int n)
    {
        // We basically find index of minimum
        // element
        int min = arr[0], min_index = -1;
        for (int i = 0; i < n; i++)
        {
            if (min > arr[i])
            {
                min = arr[i];
                min_index = i;
            }
        }
        return min_index;
    }

    // Driver program to test above functions
    public static void Main ()
    {
        int []arr = {15, 18, 2, 3, 6, 12};
        int n = arr.Length;

    Console.WriteLine(countRotations(arr, n));
    }
}
// This code is contributed by vt_m.
```

PHP

```
 <?php
```

```
// PHP program to find number
// of rotations in a sorted
// and rotated array.

// Returns count of rotations
// for an array which is first
// sorted in ascending order,
// then rotated
function countRotations($arr, $n)
{
    // We basically find index
    // of minimum element
    $min = $arr[0];
    $min_index;
    for ($i = 0; $i < $n; $i++)
    {
        if ($min > $arr[$i])
        {
            $min = $arr[$i];
            $min_index = $i;
        }
    }
    return $min_index;
}

// Driver code
$arr = array(15, 18, 2,
             3, 6, 12);
$n = sizeof($arr);
echo countRotations($arr, $n);

// This code is contributed
// by ajit
?>
```

Output:

```
2
```

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

Method 2 (Efficient Using Binary Search)
Here are also we find index of minimum element, but using Binary Search. The idea is based on below facts :

- The minimum element is the only element whose previous is greater than it. If there is no previous element element, then there is no rotation (first element is minimum).

  We check this condition for middle element by comparing it with (mid-1)'th and (mid+1)'th elements.
- If minimum element is not at middle (neither mid nor mid + 1), then minimum element lies in either left half or right half.
    1. If middle element is smaller than last element, then the minimum element lies in left half
    2. Else minimum element lies in right half.

Below is the implementation taken from here.

C++

```
 // Binary Search based C++ program to find number
// of rotations in a sorted and rotated array.
#include<bits/stdc++.h>
using namespace std;

// Returns count of rotations for an array which
// is first sorted in ascending order, then rotated
int countRotations(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 0;

    // If there is only one element left
    if (high == low)
        return low;

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element.
    // Consider the cases like {3, 4, 5, 1, 2}
    if (mid < high && arr[mid+1] < arr[mid])
       return (mid+1);

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
       return mid;

    // Decide whether we need to go to left half or
    // right half
    if (arr[high] > arr[mid])
       return countRotations(arr, low, mid-1);
    return countRotations(arr, mid+1, high);
}

// Driver code
int main()
{
    int arr[] = {15, 18, 2, 3, 6, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << countRotations(arr, 0, n-1);
    return 0;
}
```

Java

```
 // Java program to find number of
// rotations in a sorted and rotated
// array.
import java.util.*;
import java.lang.*;
import java.io.*;

class BinarySearch
{
    // Returns count of rotations for an array
    // which is first sorted in ascending order,
    // then rotated
    static int countRotations(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 0;

        // If there is only one element left
        if (high == low)
            return low;

        // Find mid
        // /*(low + high)/2;*/
        int mid = low + (high - low)/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 (mid + 1);
        // Check if mid itself is minimum element
        if (mid > low && arr[mid] < arr[mid - 1])
            return mid;

        // Decide whether we need to go to left
        // half or right half
        if (arr[high] > arr[mid])
            return countRotations(arr, low, mid - 1);

        return countRotations(arr, mid + 1, high);
    }

    // Driver program to test above functions
    public static void main (String[] args)
    {
        int arr[] = {15, 18, 2, 3, 6, 12};
        int n = arr.length;

        System.out.println(countRotations(arr, 0, n-1));
    }
}
// This code is contributed by Chhavi
```

Python3

```
 # Binary Search based Python3
# program to find number of
# rotations in a sorted and
# rotated array.

# Returns count of rotations for
# an array which is first sorted
# in ascending order, then rotated
def countRotations(arr,low, high):

    # This condition is needed to
    # handle the case when array
    # is not rotated at all
    if (high < low):
        return 0

    # If there is only one
    # element left
    if (high == low):
        return low

    # Find mid
    # (low + high)/2
    mid = low + (high - low)/2;
    mid = int(mid)

    # Check if element (mid+1) is
    # minimum element. Consider
    # the cases like {3, 4, 5, 1, 2}
    if (mid < high and arr[mid+1] < arr[mid]):
        return (mid+1)

    # Check if mid itself is
    # minimum element
    if (mid > low and arr[mid] < arr[mid - 1]):
        return mid

    # Decide whether we need to go
    # to left half or right half
    if (arr[high] > arr[mid]):
        return countRotations(arr, low, mid-1);

    return countRotations(arr, mid+1, high)

# Driver code
arr = [15, 18, 2, 3, 6, 12]
n = len(arr)
print(countRotations(arr, 0, n-1))

# This code is contributed by Smitha Dinesh Semwal
```

C#

```
 // C# program to find number of
// rotations in a sorted and rotated
// array.
using System;

class BinarySearch
{
    // Returns count of rotations for an array
    // which is first sorted in ascending order,
    // then rotated
    static int countRotations(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 0;
        // If there is only one element left
        if (high == low)
            return low;

        // Find mid
        // /*(low + high)/2;*/
        int mid = low + (high - low)/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 (mid + 1);

        // Check if mid itself is minimum element
        if (mid > low && arr[mid] < arr[mid - 1])
            return mid;

        // Decide whether we need to go to left
        // half or right half
        if (arr[high] > arr[mid])
            return countRotations(arr, low, mid - 1);

        return countRotations(arr, mid + 1, high);
    }

    // Driver program to test above functions
    public static void Main ()
    {
        int []arr = {15, 18, 2, 3, 6, 12};
        int n = arr.Length;

    Console.WriteLine(countRotations(arr, 0, n-1));
    }
}
// This code is contributed by vt_m.
```

PHP

```
 <?php
// Binary Search based PHP
// program to find number
// of rotations in a sorted
// and rotated array.

// Returns count of rotations
// for an array which is
// first sorted in ascending
// order, then rotated
function countRotations($arr,
                        $low, $high)
{
    // This condition is needed
    // to handle the case when
    // array is not rotated at all
    if ($high < $low)
        return 0;

    // If there is only
    // one element left
    if ($high == $low)
        return $low;

    // Find mid
    $mid = $low + ($high -
                   $low) / 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 (int)($mid + 1);

    // Check if mid itself
    // is minimum element
    if ($mid > $low &&
        $arr[$mid] < $arr[$mid - 1])
    return (int)($mid);

    // Decide whether we need
    // to go to left half or
    // right half
    if ($arr[$high] > $arr[$mid])
    return countRotations($arr, $low,
                          $mid - 1);

    return countRotations($arr,
                          $mid + 1,
                          $high);
}

// Driver code
$arr = array(15, 18, 2, 3, 6, 12);
$n = sizeof($arr);
```

```
echo countRotations($arr, 0, $n - 1);

// This code is contributed bu ajit
?>
```

Output:

```
2
```

Time Complexity : O(Log n)
Auxiliary Space : O(1) if we use iterative Binary Search is used (Readers can refer Binary Search article for iterative Binary Search)

Improved By : komal27, jit_t

Source

https://www.geeksforgeeks.org/find-rotation-count-rotated-sorted-array/

Chapter 34

Find the element that appears once in a sorted array

Find the element that appears once in a sorted array - GeeksforGeeks

Given a sorted array in which all elements appear twice (one after one) and one element appears only once. Find that element in O(log n) complexity.

Example:

```
Input:   arr[] = {1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 8}
Output:  4

Input:   arr[] = {1, 1, 3, 3, 4, 4, 5, 5, 7, 7, 8}
Output:  8
```

A **Simple Solution** is to traverse the array from left to right. Since the array is sorted, we can easily figure out the required element.

An **Efficient Solution** can find the required element in O(Log n) time. The idea is to use Binary Search. Below is an observation in input array.
All elements before the required have first occurrence at even index (0, 2, ..) and next occurrence at odd index (1, 3, …). And all elements after the required element have first occurrence at odd index and next occurrence at even index.

1) Find the middle index, say 'mid'.

2) If 'mid' is even, then compare arr[mid] and arr[mid + 1]. If both are same, then the required element after 'mid' else before mid.

3) If 'mid' is odd, then compare arr[mid] and arr[mid – 1]. If both are same, then the required element after 'mid' else before mid.

Below is the implementation based on above idea.

C/C++

```
 // C program to find the element that appears only once
#include<stdio.h>

// A Binary Search based function to find the element
// that appears only once
void search(int *arr, int low, int high)
{
     // Base cases
    if (low > high)
       return;

    if (low==high)
    {
        printf("The required element is %d ", arr[low]);
        return;
    }

    // Find the middle point
    int mid = (low + high) / 2;

    // If mid is even and element next to mid is
    // same as mid, then output element lies on
    // right side, else on left side
    if (mid%2 == 0)
    {
        if (arr[mid] == arr[mid+1])
            search(arr, mid+2, high);
        else
            search(arr, low, mid);
    }
    else  // If mid is odd
    {
        if (arr[mid] == arr[mid-1])
            search(arr, mid+1, high);
        else
            search(arr, low, mid-1);
    }
}

// Driver program
int main()
{
    int arr[] = {1, 1, 2, 4, 4, 5, 5, 6, 6};
    int len = sizeof(arr)/sizeof(arr[0]);
    search(arr, 0, len-1);
    return 0;
}
```

Java

```
 // Java program to find the element that appears only once

public class Main
{
    // A Binary Search based method to find the element
    // that appears only once
    public static void search(int[] arr, int low, int high)
    {
        if(low > high)
            return;
        if(low == high)
        {
            System.out.println("The required element is "+arr[low]);
            return;
        }

        // Find the middle point
        int mid = (low + high)/2;

        // If mid is even and element next to mid is
        // same as mid, then output element lies on
        // right side, else on left side
        if(mid % 2 == 0)
        {
            if(arr[mid] == arr[mid+1])
                search(arr, mid+2, high);
            else
                search(arr, low, mid);
        }
        // If mid is odd
        else if(mid % 2 == 1)
        {
            if(arr[mid] == arr[mid-1])
                search(arr, mid+1, high);
            else
                search(arr, low, mid-1);
        }
    }

    public static void main(String[] args)
    {
        int[] arr = {1, 1, 2, 4, 4, 5, 5, 6, 6};
        search(arr, 0, arr.length-1);
    }
}
// This code is contributed by Tanisha Mittal
```

Python

```
 # A Binary search based function to find
# the element that appears only once
def search(arr, low, high):

    # Base cases
    if low > high:
        return None

    if low == high:
        return arr[low]

    # Find the middle point
    mid = low + (high - low)/2

    # If mid is even and element next to mid is
    # same as mid, then output element lies on
    # right side, else on left side
    if mid%2 == 0:

        if arr[mid] == arr[mid+1]:
            return search(arr, mid+2, high)
        else:
            return search(arr, low, mid)

    else:
        # if mid is odd
        if arr[mid] == arr[mid-1]:
            return search(arr, mid+1, high)
        else:
            return search(arr, low, mid-1)


# Test Array
arr = [ 1, 1, 2, 4, 4, 5, 5, 6, 6 ]

# Function call
result = search(arr, 0, len(arr)-1)

if result is not None:
    print "The required element is %d" % result
else:
    print "Invalid Array"
```

C#

```
 // C# program to find the element
// that appears only once
using System;

class GFG {

    // A Binary Search based
    // method to find the element
    // that appears only once
    public static void search(int[] arr,
                              int low,
                              int high)
    {

        if(low > high)
            return;
        if(low == high)
        {
            Console.WriteLine("The required element is "
                                             +arr[low]);
            return;
        }

        // Find the middle point
        int mid = (low + high)/2;

        // If mid is even and element
        // next to mid is same as mid
        // then output element lies on
        // right side, else on left side
        if(mid % 2 == 0)
        {
            if(arr[mid] == arr[mid + 1])
                search(arr, mid + 2, high);
            else
                search(arr, low, mid);
        }

        // If mid is odd
        else if(mid % 2 == 1)
        {
            if(arr[mid] == arr[mid - 1])
                search(arr, mid + 1, high);
            else
                search(arr, low, mid - 1);
        }
    }

    // Driver Code
    public static void Main(String[] args)
    {
        int[] arr = {1, 1, 2, 4, 4, 5, 5, 6, 6};
        search(arr, 0, arr.Length - 1);
    }
}

// This code is contributed by Nitin Mittal.
```

PHP

```
 <?php
// PHP program to find the element
// that appears only once

// A Binary Search based function
// to find the element that
// appears only once
function search($arr, $low, $high)
{

    // Base cases
    if ($low > $high)
        return;

    if ($low==$high)
    {
        echo("The required element is " );
        echo $arr[$low] ;
        return;
    }

    // Find the middle point
    $mid = ($low + $high) / 2;

    // If mid is even and element
    // next to mid is same as mid,
    // then output element lies on
    // right side, else on left side
    if ($mid % 2 == 0)
    {
        if ($arr[$mid] == $arr[$mid + 1])
            search($arr, $mid + 2, $high);
        else
            search($arr, $low, $mid);
    }

    // If mid is odd
    else
    {
        if ($arr[$mid] == $arr[$mid - 1])
            search($arr, $mid + 1, $high);
        else
            search($arr, $low, $mid - 1);
    }
}

    // Driver Code
    $arr = array(1, 1, 2, 4, 4, 5, 5, 6, 6);
    $len = sizeof($arr);
    search($arr, 0, $len - 1);

// This code is contributed by nitin mittal
?>
```

Output:

```
The required element is 2
```

Time Complexity: O(Log n)

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

Improved By : Aalaa Abdel Mawla, nitin mittal, AntonyCherepanov

Source

https://www.geeksforgeeks.org/find-the-element-that-appears-once-in-a-sorted-array/

Chapter 35

Find the maximum element in an array which is first increasing and then decreasing

Find the maximum element in an array which is first increasing and then decreasing - GeeksforGeeks

Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.
Examples :

```
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.

C

```
 #include <stdio.h>

int findMaximum(int arr[], int low, int high)
{
   int max = arr[low];
   int i;
   for (i = low; i <= high; i++)
   {
       if (arr[i] > max)
          max = arr[i];
   }
   return max;
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 30, 40, 50, 60, 70, 23, 20};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The maximum element is %d", findMaximum(arr, 0, n-1));
   getchar();
   return 0;
}
```

Java

```
 // java program to find maximum
// element

class Main
{
    // function to find the
    // maximum element
    static int findMaximum(int arr[], int low, int high)
    {
       int max = arr[low];
       int i;
       for (i = low; i <= high; i++)
       {
           if (arr[i] > max)
              max = arr[i];
       }
       return max;
    }

    // main function
    public static void main (String[] args)
    {
        int arr[] = {1, 30, 40, 50, 60, 70, 23, 20};
        int n = arr.length;
        System.out.println("The maximum element is "+
                            findMaximum(arr, 0, n-1));
    }
}
```

Python3

```
 # Python3 program to find
# maximum element

def findMaximum(arr, low, high):
    max = arr[low]
    i = low
    for i in range(high+1):
        if arr[i] > max:
            max = arr[i]
    return max

# Driver program to check above functions */
arr = [1, 30, 40, 50, 60, 70, 23, 20]
n = len(arr)
print ("The maximum element is %d"%
        findMaximum(arr, 0, n-1))

# This code is contributed by Shreyanshi Arun.
```

C#

```
 // C# program to find maximum
// element
using System;

class GFG
{
    // function to find the
    // maximum element
    static int findMaximum(int []arr, int low, int high)
    {
        int max = arr[low];
        int i;
        for (i = low; i <= high; i++)
        {
            if (arr[i] > max)
                max = arr[i];
        }
        return max;
    }

    // Driver code
    public static void Main ()
    {
        int []arr = {1, 30, 40, 50, 60, 70, 23, 20};
        int n = arr.Length;
        Console.Write("The maximum element is "+
                        findMaximum(arr, 0, n-1));
    }
}

// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP program to Find the maximum
// element in an array which is first
// increasing and then decreasing

function findMaximum($arr, $low, $high)
{
$max = $arr[$low];
$i;
for ($i = $low; $i <= $high; $i++)
{
    if ($arr[$i] > $max)
        $max = $arr[$i];
}
return $max;
}

// Driver Code
$arr = array(1, 30, 40, 50,
             60, 70, 23, 20);
$n = count($arr);
echo "The maximum element is ",
      findMaximum($arr, 0, $n-1);

// This code is contributed by anuj_67.
?>
```

Output :

```
The maximum element is 70
```

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;
}
```

Java

```
 // java program to find maximum
// element

class Main
{
    // function to find the
    // maximum element
    static 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];

       /*low + (high - low)/2;*/
       int mid = (low + high)/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
         return findMaximum(arr, mid + 1, high);
    }

    // main function
    public static void main (String[] args)
    {
        int arr[] = {1, 3, 50, 10, 9, 7, 6};
        int n = arr.length;
        System.out.println("The maximum element is "+
                            findMaximum(arr, 0, n-1));
    }
}
```

Python3

```
 def findMaximum(arr, low, 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 and 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 and arr[low] < arr[high]:
        return arr[high]

    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] and 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] and 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 */
arr = [1, 3, 50, 10, 9, 7, 6]
n = len(arr)
print ("The maximum element is %d"% findMaximum(arr, 0, n-1))

# This code is contributed by Shreyanshi Arun.
```

C#

```
 // C# program to find maximum
// element
using System;

class GFG
{
    // function to find the
    // maximum element
    static 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];

    /*low + (high - low)/2;*/
    int mid = (low + high)/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
        return findMaximum(arr, mid + 1, high);
    }

    // main function
    public static void Main()
    {
        int []arr = {1, 3, 50, 10, 9, 7, 6};
        int n = arr.Length;
        Console.Write("The maximum element is "+
                            findMaximum(arr, 0, n-1));
    }
}
// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP program to Find the maximum
// element in an array which is
// first increasing and then decreasing

function findMaximum($arr, $low, $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];

    /*low + (high - low)/2;*/
    $mid = ($low + $high) / 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);

    // when arr[mid] is greater than
    // arr[mid-1] and smaller than
    // arr[mid+1]
    else
        return findMaximum($arr,
                           $mid + 1, $high);
}

// Driver Code
$arr = array(1, 3, 50, 10, 9, 7, 6);
$n = sizeof($arr);
echo("The maximum element is ");
echo(findMaximum($arr, 0, $n-1));

// This code is contributed by nitin mittal.
?>
```

Output :

```
The maximum element is 50
```

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}.

Improved By : vt_m, nitin mittal

Source

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

Chapter 36

Find the minimum element in a sorted and rotated array

Find the minimum element in a sorted and rotated array - GeeksforGeeks

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 ?(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 is greater than it. If there is no previous element element, then there is no rotation (first element is minimum). We check this condition for middle element by comparing it with (mid-1)'th and (mid+1)'th elements.
- If minimum element is not at middle (neither mid nor mid + 1), then minimum element lies in either left half or right half.

1. If middle element is smaller than last element, then the minimum element lies in left half
2. Else minimum element lies in right half.

We strongly recommend you to try it yourself before seeing the following implementation.

C/C++

```
 // C program to find minimum element in a sorted and rotated array
#include <stdio.h>

int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)  return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {3, 4, 5, 1, 2}
    if (mid < high && arr[mid+1] < arr[mid])
       return arr[mid+1];

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
       return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
       return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] =  {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));

    int arr2[] =  {1, 2, 3, 4};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));
    int arr3[] =  {1};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

    int arr4[] =  {1, 2};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

    int arr5[] =  {2, 1};
    int n5 = sizeof(arr5)/sizeof(arr5[0]);
    printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));

    int arr6[] =  {5, 6, 7, 1, 2, 3, 4};
    int n6 = sizeof(arr6)/sizeof(arr6[0]);
    printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

    int arr7[] =  {1, 2, 3, 4, 5, 6, 7};
    int n7 = sizeof(arr7)/sizeof(arr7[0]);
    printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

    int arr8[] =  {2, 3, 4, 5, 6, 7, 8, 1};
    int n8 = sizeof(arr8)/sizeof(arr8[0]);
    printf("The minimum element is %d\n", findMin(arr8, 0, n8-1));

    int arr9[] =  {3, 4, 5, 1, 2};
    int n9 = sizeof(arr9)/sizeof(arr9[0]);
    printf("The minimum element is %d\n", findMin(arr9, 0, n9-1));

    return 0;
}
```

Java

```
 // Java program to find minimum element in a sorted and rotated array
import java.util.*;
import java.lang.*;
import java.io.*;

class Minimum
{
    static int findMin(int arr[], int low, int high)
    {
        // This condition is needed to handle the case when array
        // is not rotated at all
        if (high < low)  return arr[0];

        // If there is only one element left
        if (high == low) return arr[low];

        // Find mid
        int mid = low + (high - low)/2; /*(low + high)/2;*/

        // Check if element (mid+1) is minimum element. Consider
        // the cases like {3, 4, 5, 1, 2}
        if (mid < high && arr[mid+1] < arr[mid])
            return arr[mid+1];

        // Check if mid itself is minimum element
        if (mid > low && arr[mid] < arr[mid - 1])
            return arr[mid];

        // Decide whether we need to go to left half or right half
        if (arr[high] > arr[mid])
            return findMin(arr, low, mid-1);
        return findMin(arr, mid+1, high);
    }

    // Driver Program
    public static void main (String[] args)
    {
        int arr1[] =  {5, 6, 1, 2, 3, 4};
        int n1 = arr1.length;
        System.out.println("The minimum element is "+ findMin(arr1, 0, n1-1));

        int arr2[] =  {1, 2, 3, 4};
        int n2 = arr2.length;
        System.out.println("The minimum element is "+ findMin(arr2, 0, n2-1));

        int arr3[] =  {1};
        int n3 = arr3.length;
        System.out.println("The minimum element is "+ findMin(arr3, 0, n3-1));

        int arr4[] =  {1, 2};
        int n4 = arr4.length;
        System.out.println("The minimum element is "+ findMin(arr4, 0, n4-1));

        int arr5[] =  {2, 1};
        int n5 = arr5.length;
        System.out.println("The minimum element is "+ findMin(arr5, 0, n5-1));

        int arr6[] =  {5, 6, 7, 1, 2, 3, 4};
        int n6 = arr6.length;
        System.out.println("The minimum element is "+ findMin(arr6, 0, n1-1));

        int arr7[] =  {1, 2, 3, 4, 5, 6, 7};
        int n7 = arr7.length;
        System.out.println("The minimum element is "+ findMin(arr7, 0, n7-1));

        int arr8[] =  {2, 3, 4, 5, 6, 7, 8, 1};
        int n8 = arr8.length;
        System.out.println("The minimum element is "+ findMin(arr8, 0, n8-1));

        int arr9[] =  {3, 4, 5, 1, 2};
        int n9 = arr9.length;
        System.out.println("The minimum element is "+ findMin(arr9, 0, n9-1));
    }
}
```

Python

```
 # Python program to find minimum element
# in a sorted and rotated array

def findMin(arr, low, 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
    mid = int((low + high)/2)

    # Check if element (mid+1) is minimum element. Consider
    # the cases like [3, 4, 5, 1, 2]
    if mid < high and arr[mid+1] < arr[mid]:
        return arr[mid+1]

    # Check if mid itself is minimum element
    if mid > low and 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
arr1 = [5, 6, 1, 2, 3, 4]
n1 = len(arr1)
```

```
print("The minimum element is " + str(findMin(arr1, 0, n1-1)))

arr2 = [1, 2, 3, 4]
n2 = len(arr2)
print("The minimum element is " + str(findMin(arr2, 0, n2-1)))

arr3 = [1]
n3 = len(arr3)
print("The minimum element is " + str(findMin(arr3, 0, n3-1)))

arr4 = [1, 2]
n4 = len(arr4)
print("The minimum element is " + str(findMin(arr4, 0, n4-1)))

arr5 = [2, 1]
n5 = len(arr5)
print("The minimum element is " + str(findMin(arr5, 0, n5-1)))

arr6 = [5, 6, 7, 1, 2, 3, 4]
n6 = len(arr6)
print("The minimum element is " + str(findMin(arr6, 0, n6-1)))

arr7 = [1, 2, 3, 4, 5, 6, 7]
n7 = len(arr7)
print("The minimum element is " + str(findMin(arr7, 0, n7-1)))

arr8 = [2, 3, 4, 5, 6, 7, 8, 1]
n8 = len(arr8)
print("The minimum element is " + str(findMin(arr8, 0, n8-1)))

arr9 = [3, 4, 5, 1, 2]
n9 = len(arr9)
print("The minimum element is " + str(findMin(arr9, 0, n9-1)))

# This code is contributed by Pratik Chhajer
```

C#

```
 // C# program to find minimum element
// in a sorted and rotated array
using System;

class Minimum {

    static int findMin(int[] arr, int low, int high)
    {
        // This condition is needed to handle
        // the case when array
        // is not rotated at all
        if (high < low)
            return arr[0];

        // If there is only one element left
        if (high == low)
            return arr[low];

        // Find mid
        // (low + high)/2
        int mid = low + (high - low) / 2;

        // Check if element (mid+1) is minimum element. Consider
        // the cases like {3, 4, 5, 1, 2}
        if (mid < high && arr[mid + 1] < arr[mid])
            return arr[mid + 1];

        // Check if mid itself is minimum element
        if (mid > low && arr[mid] < arr[mid - 1])
            return arr[mid];

        // Decide whether we need to go to
        // left half or right half
        if (arr[high] > arr[mid])
            return findMin(arr, low, mid - 1);
        return findMin(arr, mid + 1, high);
    }

    // Driver Program
    public static void Main()
    {
        int[] arr1 = { 5, 6, 1, 2, 3, 4 };
        int n1 = arr1.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr1, 0, n1 - 1));

        int[] arr2 = { 1, 2, 3, 4 };
        int n2 = arr2.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr2, 0, n2 - 1));

        int[] arr3 = { 1 };
        int n3 = arr3.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr3, 0, n3 - 1));

        int[] arr4 = { 1, 2 };
        int n4 = arr4.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr4, 0, n4 - 1));

        int[] arr5 = { 2, 1 };
        int n5 = arr5.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr5, 0, n5 - 1));

        int[] arr6 = { 5, 6, 7, 1, 2, 3, 4 };
        int n6 = arr6.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr6, 0, n1 - 1));

        int[] arr7 = { 1, 2, 3, 4, 5, 6, 7 };
        int n7 = arr7.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr7, 0, n7 - 1));

        int[] arr8 = { 2, 3, 4, 5, 6, 7, 8, 1 };
        int n8 = arr8.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr8, 0, n8 - 1));

        int[] arr9 = { 3, 4, 5, 1, 2 };
        int n9 = arr9.Length;
        Console.WriteLine("The minimum element is " +
                          findMin(arr9, 0, n9 - 1));
    }
}

// This code is contributed by vt_m.
```

PHP

```
 <?php
// PHP program to find minimum
// element in a sorted and
// rotated array

function findMin($arr, $low,
                 $high)
{
    // This condition is needed
    // to handle the case when
    // array is not rotated at all
    if ($high < $low) return $arr[0];

    // If there is only
    // one element left
    if ($high == $low) return $arr[$low];

    // Find mid
    $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 Code
$arr1 = array(5, 6, 1, 2, 3, 4);
$n1 = sizeof($arr1);
echo "The minimum element is " .
    findMin($arr1, 0, $n1 - 1) . "\n";

$arr2 = array(1, 2, 3, 4);
$n2 = sizeof($arr2);
echo "The minimum element is " .
    findMin($arr2, 0, $n2 - 1) . "\n";

$arr3 = array(1);
$n3 = sizeof($arr3);
echo "The minimum element is " .
    findMin($arr3, 0, $n3 - 1) . "\n";

$arr4 = array(1, 2);
$n4 = sizeof($arr4);
```

```
echo "The minimum element is " .
    findMin($arr4, 0, $n4 - 1) . "\n";

$arr5 = array(2, 1);
$n5 = sizeof($arr5);
echo "The minimum element is " .
    findMin($arr5, 0, $n5 - 1) . "\n";

$arr6 = array(5, 6, 7, 1, 2, 3, 4);
$n6 = sizeof($arr6);
echo "The minimum element is " .
    findMin($arr6, 0, $n6 - 1) . "\n";

$arr7 = array(1, 2, 3, 4, 5, 6, 7);
$n7 = sizeof($arr7);
echo "The minimum element is " .
    findMin($arr7, 0, $n7 - 1) . "\n";

$arr8 = array(2, 3, 4, 5, 6, 7, 8, 1);
$n8 = sizeof($arr8);
echo "The minimum element is " .
    findMin($arr8, 0, $n8 - 1) . "\n";

$arr9 = array(3, 4, 5, 1, 2);
$n9 = sizeof($arr9);
echo "The minimum element is " .
    findMin($arr9, 0, $n9 - 1) . "\n";

// This code is contributed by ChitraNayal
?>
```

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. So the problem with repetition can be solved in O(n) worst case.

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.

Improved By : ChitraNayal

Source

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

Chapter 37

Find the missing number in Arithmetic Progression

Find the missing number in Arithmetic Progression - GeeksforGeeks

Given an array that represents elements of arithmetic progression in order. One element is missing in the progression, find the missing number.

Examples:

```
Input: arr[]  = {2, 4, 8, 10, 12, 14}
Output: 6

Input: arr[]  = {1, 6, 11, 16, 21, 31};
Output: 26
```

A **Simple Solution** is to linearly traverse the array and find the missing number. Time complexity of this solution is O(n).

We can solve this problem **in O(Logn) time** using Binary Search. The idea is to go to the middle element. Check if the difference between middle and next to middle is equal to diff or not, if not then the missing element lies between mid and mid+1. If the middle element is equal to $n/2^{th}$ term in Arithmetic Series (Let n be the number of elements in input array), then missing element lies in right half. Else element lies in left half.

Following is implementation of above idea.

C

```
 // A C program to find the missing number in a given
// arithmetic progression
#include <stdio.h>
```

```
#include <limits.h>

// A binary search based recursive function that returns
// the missing element in arithmetic progression
int findMissingUtil(int arr[], int low, int high, int diff)
{
    // There must be two elements to find the missing
    if (high <= low)
        return INT_MAX;

    // Find index of middle element
    int mid = low + (high - low)/2;

    // The element just after the middle element is missing.
    // The arr[mid+1] must exist, because we return when
    // (low == high) and take floor of (high-low)/2
    if (arr[mid+1] - arr[mid] != diff)
        return (arr[mid] + diff);

    // The element just before mid is missing
    if (mid > 0 && arr[mid] - arr[mid-1] != diff)
        return (arr[mid-1] + diff);

    // If the elements till mid follow AP, then recur
    // for right half
    if (arr[mid] == arr[0] + mid*diff)
        return findMissingUtil(arr, mid+1, high, diff);

    // Else recur for left half
    return findMissingUtil(arr, low, mid-1, diff);
}

// The function uses findMissingUtil() to find the missing
// element in AP.  It assumes that there is exactly one missing
// element and may give incorrect result when there is no missing
// element or more than one missing elements.
// This function also assumes that the difference in AP is an
// integer.
int findMissing(int arr[], int n)
{
    // If exactly one element is missing, then we can find
    // difference of arithmetic progression using following
    // formula.  Example, 2, 4, 6, 10, diff = (10-2)/4 = 2.
    // The assumption in formula is that the difference is
    // an integer.
    int diff = (arr[n-1] - arr[0])/n;

    // Binary search for the missing number using above
    // calculated diff
    return findMissingUtil(arr, 0, n-1, diff);
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {2, 4, 8, 10, 12, 14};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("The missing element is %d", findMissing(arr, n));
    return 0;
}
```

Java

```
 // A Java program to find
// the missing number in
// a given arithmetic
// progression
import java.io.*;

class GFG
{

// A binary search based
// recursive function that
// returns the missing
// element in arithmetic
// progression
static int findMissingUtil(int arr[], int low,
                           int high, int diff)
{
    // There must be two elements
    // to find the missing
    if (high <= low)
        return Integer.MAX_VALUE;

    // Find index of
    // middle element
    int mid = low + (high - low) / 2;

    // The element just after the
    // middle element is missing.
    // The arr[mid+1] must exist,
    // because we return when
    // (low == high) and take
    // floor of (high-low)/2
    if (arr[mid + 1] - arr[mid] != diff)
```

```
        return (arr[mid] + diff);

    // The element just
    // before mid is missing
    if (mid > 0 && arr[mid] -
                  arr[mid - 1] != diff)
        return (arr[mid - 1] + diff);

    // If the elements till mid follow
    // AP, then recur for right half
    if (arr[mid] == arr[0] + mid * diff)
        return findMissingUtil(arr, mid + 1,
                               high, diff);

    // Else recur for left half
    return findMissingUtil(arr, low, mid - 1, diff);
}

// The function uses findMissingUtil()
// to find the missing element in AP.
// It assumes that there is exactly
// one missing element and may give
// incorrect result when there is no
// missing element or more than one
// missing elements. This function also
// assumes that the difference in AP is
// an integer.
static int findMissing(int arr[], int n)
{
    // If exactly one element is missing,
    // then we can find difference of
    // arithmetic progression using
    // following formula. Example, 2, 4,
    // 6, 10, diff = (10-2)/4 = 2.
    // The assumption in formula is that
    // the difference is an integer.
    int diff = (arr[n - 1] - arr[0]) / n;

    // Binary search for the missing
    // number using above calculated diff
    return findMissingUtil(arr, 0, n - 1, diff);
}

// Driver Code
public static void main (String[] args)
{
    int arr[] = {2, 4, 8, 10, 12, 14};
    int n = arr.length;
    System.out.println("The missing element is "+
                            findMissing(arr, n));
}
}

// This code is contributed by anuj_67.
```

C#

```
 // A C# program to find
// the missing number in
// a given arithmetic
// progression
using System;

class GFG
{

// A binary search based
// recursive function that
// returns the missing
// element in arithmetic
// progression
static int findMissingUtil(int []arr, int low,
                           int high, int diff)
{
    // There must be two elements
    // to find the missing
    if (high <= low)
        return int.MaxValue;

    // Find index of
    // middle element
    int mid = low + (high -
                     low) / 2;

    // The element just after the
    // middle element is missing.
    // The arr[mid+1] must exist,
    // because we return when
    // (low == high) and take
    // floor of (high-low)/2
    if (arr[mid + 1] -
        arr[mid] != diff)
        return (arr[mid] + diff);

    // The element just
    // before mid is missing
    if (mid > 0 && arr[mid] -
                  arr[mid - 1] != diff)
        return (arr[mid - 1] + diff);

    // If the elements till mid follow
    // AP, then recur for right half
    if (arr[mid] == arr[0] +
              mid * diff)
        return findMissingUtil(arr, mid + 1,
                                high, diff);

    // Else recur for left half
    return findMissingUtil(arr, low,
                           mid - 1, diff);
}

// The function uses findMissingUtil()
// to find the missing element
// in AP. It assumes that there
// is exactly one missing element
// and may give incorrect result
// when there is no missing element
// or more than one missing elements.
// This function also assumes that
// the difference in AP is an integer.
static int findMissing(int []arr, int n)
{
    // If exactly one element
    // is missing, then we can
    // find difference of arithmetic
    // progression using following
    // formula. Example, 2, 4, 6, 10,
    // diff = (10-2)/4 = 2.The assumption
    // in formula is that the difference
    // is an integer.
    int diff = (arr[n - 1] -
                arr[0]) / n;

    // Binary search for the
    // missing number using
    // above calculated diff
    return findMissingUtil(arr, 0,
                           n - 1, diff);
}

// Driver Code
public static void Main ()
{
    int []arr = {2, 4, 8,
                 10, 12, 14};
    int n = arr.Length;
    Console.WriteLine("The missing element is "+
                           findMissing(arr, n));
}
}

// This code is contributed by anuj_67.
```

Output:

```
The missing element is 6
```

Exercise:
Solve the same problem for Geometrical Series. What is the time complexity of your solution? What about Fibonacci Series?

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

Improved By : vt_m

Source

https://www.geeksforgeeks.org/find-missing-number-arithmetic-progression/

Chapter 38

Find the number of zeroes

Find the number of zeroes - GeeksforGeeks

Given an array of 1s and 0s which has all 1s first followed by all 0s. Find the number of 0s. Count the number of zeroes in the given array.

Examples :

```
Input: arr[] = {1, 1, 1, 1, 0, 0}
Output: 2

Input: arr[] = {1, 0, 0, 0, 0}
Output: 4

Input: arr[] = {0, 0, 0}
Output: 3

Input: arr[] = {1, 1, 1, 1}
Output: 0
```

A **simple solution** is to traverse the input array. As soon as we find a 0, we return n – index of first 0. Here n is number of elements in input array. Time complexity of this solution would be O(n).

Since the input array is sorted, we can use **Binary Search** to find the first occurrence of 0. Once we have index of first element, we can return count as n – index of first zero.

C

```
 // A divide and conquer solution to find count of zeroes in an array
// where all 1s are present before all 0s
#include <stdio.h>
```

```
/* if 0 is present in arr[] then returns the index of FIRST occurrence
   of 0 in arr[low..high], otherwise returns -1 */
int firstZero(int arr[], int low, int high)
{
    if (high >= low)
    {
        // Check if mid element is first 0
        int mid = low + (high - low)/2;
        if (( mid == 0 || arr[mid-1] == 1) && arr[mid] == 0)
            return mid;

        if (arr[mid] == 1)  // If mid element is not 0
            return firstZero(arr, (mid + 1), high);
        else  // If mid element is 0, but not first 0
            return firstZero(arr, low, (mid -1));
    }
    return -1;
}

// A wrapper over recursive function firstZero()
int countZeroes(int arr[], int n)
{
    // Find index of first zero in given array
    int first = firstZero(arr, 0, n-1);

    // If 0 is not present at all, return 0
    if (first == -1)
        return 0;

    return (n - first);
}

/* Driver program to check above functions */
int main()
{
    int arr[] =   {1, 1, 1, 0, 0, 0, 0, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Count of zeroes is %d", countZeroes(arr, n));
    return 0;
}
```

Java

```
 // A divide and conquer solution to find count of zeroes in an array
// where all 1s are present before all 0s

class CountZeros
{
    /* if 0 is present in arr[] then returns the index of FIRST occurrence
       of 0 in arr[low..high], otherwise returns -1 */
    int firstZero(int arr[], int low, int high)
    {
        if (high >= low)
        {
            // Check if mid element is first 0
            int mid = low + (high - low) / 2;
            if ((mid == 0 || arr[mid - 1] == 1) && arr[mid] == 0)
                return mid;

            if (arr[mid] == 1) // If mid element is not 0
                return firstZero(arr, (mid + 1), high);
            else // If mid element is 0, but not first 0
                return firstZero(arr, low, (mid - 1));
        }
        return -1;
    }

    // A wrapper over recursive function firstZero()
    int countZeroes(int arr[], int n)
    {
        // Find index of first zero in given array
        int first = firstZero(arr, 0, n - 1);

        // If 0 is not present at all, return 0
        if (first == -1)
            return 0;

        return (n - first);
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        CountZeros count = new CountZeros();
        int arr[] = {1, 1, 1, 0, 0, 0, 0, 0};
        int n = arr.length;
        System.out.println("Count of zeroes is " + count.countZeroes(arr, n));
    }
}
```

Python3

```
 # A divide and conquer solution to
# find count of zeroes in an array
# where all 1s are present before all 0s
# if 0 is present in arr[] then returns
# the index of FIRST occurrence of 0 in
# arr[low..high], otherwise returns -1
def firstZero(arr, low, high):

    if (high >= low):

        # Check if mid element is first 0
        mid = low + int((high - low) / 2)
        if (( mid == 0 or arr[mid-1] == 1)
                      and arr[mid] == 0):
            return mid

        # If mid element is not 0
        if (arr[mid] == 1):
            return firstZero(arr, (mid + 1), high)

        # If mid element is 0, but not first 0
        else:
            return firstZero(arr, low, (mid - 1))

    return -1

# A wrapper over recursive
# function firstZero()
def countZeroes(arr, n):

    # Find index of first zero in given array
    first = firstZero(arr, 0, n - 1)

    # If 0 is not present at all, return 0
    if (first == -1):
        return 0

    return (n - first)

# Driver Code
arr = [1, 1, 1, 0, 0, 0, 0, 0]
n = len(arr)
print("Count of zeroes is",
        countZeroes(arr, n))

# This code is contributed by Smitha Dinesh Semwal
```

C#

```
 // A divide and conquer solution to find
// count of zeroes in an array where all
// 1s are present before all 0s
using System;

class CountZeros
{
    /* if 0 is present in arr[] then returns
       the index of FIRST occurrence of 0 in
       arr[low..high], otherwise returns -1 */
    int firstZero(int []arr, int low, int high)
    {
        if (high >= low)
        {
            // Check if mid element is first 0
            int mid = low + (high - low) / 2;
            if ((mid == 0 || arr[mid - 1] == 1) &&
                                 arr[mid] == 0)
                return mid;

            if (arr[mid] == 1) // If mid element is not 0
                return firstZero(arr, (mid + 1), high);

            else // If mid element is 0, but not first 0
                return firstZero(arr, low, (mid - 1));
        }
        return -1;
    }

    // A wrapper over recursive function firstZero()
    int countZeroes(int []arr, int n)
    {
        // Find index of first zero in given array
        int first = firstZero(arr, 0, n - 1);

        // If 0 is not present at all, return 0
        if (first == -1)
            return 0;

        return (n - first);
    }

    // Driver program to test above functions
    public static void Main()
    {
        CountZeros count = new CountZeros();
        int []arr = {1, 1, 1, 0, 0, 0, 0, 0};
        int n = arr.Length;
        Console.Write("Count of zeroes is " +
                       count.countZeroes(arr, n));
    }
}

// This code is contributed by nitin mittal.
```

PHP

```
 <?php
// A divide and conquer solution to
// find count of zeroes in an array
// where all 1s are present before all 0s

/* if 0 is present in arr[] then
   returns the index of FIRST
   occurrence of 0 in arr[low..high],
   otherwise returns -1 */
function firstZero($arr, $low, $high)
{
    if ($high >= $low)
    {

        // Check if mid element is first 0
        $mid = $low + floor(($high - $low)/2);

        if (( $mid == 0 || $arr[$mid-1] == 1) &&
                                 $arr[$mid] == 0)
            return $mid;

        // If mid element is not 0
        if ($arr[$mid] == 1)
            return firstZero($arr, ($mid + 1), $high);

        // If mid element is 0,
        // but not first 0
        else
            return firstZero($arr, $low,
                            ($mid - 1));
    }
    return -1;
}

// A wrapper over recursive
// function firstZero()
function countZeroes($arr, $n)
{

    // Find index of first
    // zero in given array
    $first = firstZero($arr, 0, $n - 1);

    // If 0 is not present
    // at all, return 0
    if ($first == -1)
        return 0;

    return ($n - $first);
}

    // Driver Code
    $arr = array(1, 1, 1, 0, 0, 0, 0, 0);
    $n = sizeof($arr);
    echo("Count of zeroes is ");
    echo(countZeroes($arr, $n));

// This code is contributed by nitin mittal
?>
```

Output:

```
Count of zeroes is 5
```

Time Complexity: O(Logn) where n is number of elements in arr[].

Improved By : nitin mittal

Source

https://www.geeksforgeeks.org/find-number-zeroes/

Chapter 39

Find the only repeating element in a sorted array of size n

Find the only repeating element in a sorted array of size n - GeeksforGeeks

Given a sorted array of n elements containing elements in range from 1 to n-1 i.e. one element occurs twice, the task is to find the repeating element in an array.

Examples :

```
Input :  arr[] = { 1, 2 , 3 , 4 , 4}
Output :  4

Input :  arr[] = { 1 , 1 , 2 , 3 , 4}
Output :  1
```

A **naive approach** is to scan the whole array and check if an element occurs twice, then return. This approach takes O(n) time.

An **efficient** method is to use Binary Search .
1- Check if the middle element is the repeating one.
2- If not then check if the middle element is at proper position if yes then start searching repeating element in right.
3- Otherwise the repeating one will be in left.

C/C++

```
 // C++ program to find the only repeating element in an
// array of size n and elements from range 1 to n-1.
#include <bits/stdc++.h>
using namespace std;

// Returns index of second appearance of a repeating element
// The function assumes that array elements are in range from
// 1 to n-1.
int findRepeatingElement(int arr[], int low, int high)
{
    // low = 0 , high = n-1;
    if (low > high)
        return -1;

    int mid = (low + high) / 2;

    // Check if the mid element is the repeating one
    if (arr[mid] != mid + 1)
    {
        if (mid > 0 && arr[mid]==arr[mid-1])
            return mid;

        // If mid element is not at its position that means
        // the repeated element is in left
        return  findRepeatingElement(arr, low, mid-1);
    }

    // If mid is at proper position then repeated one is in
    // right.
    return findRepeatingElement(arr, mid+1, high);
}

// Driver code
int main()
{
    int  arr[] = {1, 2, 3, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int index = findRepeatingElement(arr, 0, n-1);
    if (index != -1)
        cout << arr[index];
    return 0;
}
```

Java

```
 // Java program to find the only repeating element in an
// array of size n and elements from range 1 to n-1.

class Test
{
    // Returns index of second appearance of a repeating element
    // The function assumes that array elements are in range from
    // 1 to n-1.
    static int findRepeatingElement(int arr[], int low, int high)
    {
        // low = 0 , high = n-1;
        if (low > high)
            return -1;

        int mid = (low + high) / 2;

        // Check if the mid element is the repeating one
        if (arr[mid] != mid + 1)
        {
            if (mid > 0 && arr[mid]==arr[mid-1])
                return mid;

            // If mid element is not at its position that means
            // the repeated element is in left
            return  findRepeatingElement(arr, low, mid-1);
        }

        // If mid is at proper position then repeated one is in
        // right.
        return findRepeatingElement(arr, mid+1, high);
    }

    // Driver method
    public static void main(String[] args)
    {
        int  arr[] = {1, 2, 3, 3, 4, 5};
        int index = findRepeatingElement(arr, 0, arr.length-1);
        if (index != -1)
            System.out.println(arr[index]);
    }
}
```

Python

```
 # Python program to find the only repeating element in an
# array of size n and elements from range 1 to n-1

# Returns index of second appearance of a repeating element
# The function assumes that array elements are in range from
# 1 to n-1.
def findRepeatingElement(arr, low, high):

    # low = 0 , high = n-1
    if low > high:
        return -1

    mid = (low + high) / 2
    # Check if the mid element is the repeating one
    if (arr[mid] != mid + 1):

        if (mid > 0 and arr[mid]==arr[mid-1]):
            return mid

        # If mid element is not at its position that means
        # the repeated element is in left
        return  findRepeatingElement(arr, low, mid-1)

    # If mid is at proper position then repeated one is in
    # right.
    return findRepeatingElement(arr, mid+1, high)

# Driver code
arr = [1, 2, 3, 3, 4, 5]
n = len(arr)
index = findRepeatingElement(arr, 0, n-1)
if (index is not -1):
    print arr[index]

#This code is contributed by Afzal Ansari
```

C#

```
 // C# program to find the only repeating
// element in an array of size n and
// elements from range 1 to n-1.
using System;

class Test
{
    // Returns index of second appearance of a
    // repeating element. The function assumes that
    // array elements are in range from 1 to n-1.
    static int findRepeatingElement(int []arr, int low,
                                              int high)
    {
        // low = 0 , high = n-1;
        if (low > high)
            return -1;

        int mid = (low + high) / 2;

        // Check if the mid element
        // is the repeating one
        if (arr[mid] != mid + 1)
        {
            if (mid > 0 && arr[mid]==arr[mid-1])
                return mid;

            // If mid element is not at its position
            // that means the repeated element is in left
            return findRepeatingElement(arr, low, mid-1);
        }

        // If mid is at proper position
        // then repeated one is in right.
        return findRepeatingElement(arr, mid+1, high);
    }

    // Driver method
    public static void Main()
    {
        int []arr = {1, 2, 3, 3, 4, 5};
        int index = findRepeatingElement(arr, 0, arr.Length-1);
        if (index != -1)
        Console.Write(arr[index]);
    }
}

// This code is contributed by Nitin Mittal.
```

PHP

```
 <?php
// PHP program to find the only
// repeating element in an array
// of size n and elements from
// range 1 to n-1.

// Returns index of second
// appearance of a repeating
// element. The function assumes
// that array elements are in
// range from 1 to n-1.
function findRepeatingElement($arr,
                              $low,
                              $high)
{
    // low = 0 , high = n-1;
    if ($low > $high)
        return -1;

    $mid = floor(($low + $high) / 2);
    // Check if the mid element
    // is the repeating one
    if ($arr[$mid] != $mid + 1)
    {
        if ($mid > 0 && $arr[$mid] ==
                        $arr[$mid - 1])
            return $mid;

        // If mid element is not at
        // its position that means
        // the repeated element is in left
        return findRepeatingElement($arr, $low,
                                    $mid - 1);
    }

    // If mid is at proper position
    // then repeated one is in right.
    return findRepeatingElement($arr, $mid + 1,
                                        $high);
}

// Driver code
$arr = array(1, 2, 3, 3, 4, 5);
$n = sizeof($arr);
$index = findRepeatingElement($arr, 0,
                              $n - 1);
if ($index != -1)
echo $arr[$index];

// This code is contributed
// by nitin mittal.
?>
```

Output :

```
3
```

Time Complexity : O(log n)

Improved By : nitin mittal

Source

https://www.geeksforgeeks.org/find-repeating-element-sorted-array-size-n/

Chapter 40

First strictly greater element in a sorted array in Java

First strictly greater element in a sorted array in Java - GeeksforGeeks

Given a sorted array and a target value, find the first element that is strictly greater than given element.

Examples:

```
Input : arr[] = {1, 2, 3, 5, 8, 12}
        Target = 5
Output : 4 (Index of 8)

Input : {1, 2, 3, 5, 8, 12}
        Target = 8
Output : 5 (Index of 12)

Input : {1, 2, 3, 5, 8, 12}
        Target = 15
Output : -1
```

A **simple solution** is to linearly traverse given array and find first element that is strictly greater. If no such element exists, then return -1.

An **efficient solution** is to use Binary Search. In a general binary search, we are looking for a value which appears in the array. Sometimes, however, we need to find the first element which is either greater than a target.

To see that this algorithm is correct, consider each comparison being made. If we find an element that's no greater than the target element, then it and everything below it can't possibly match, so there's no need to search that region. We can thus search the right half. If we find an element that is larger than the element in question, then anything after it must also be larger, so they can't be the first element that's bigger and so we don't need to search them. The middle element is thus the last possible place it could be.

Note that on each iteration we drop off at least half the remaining elements from consideration. If the top branch executes, then the elements in the range [low, (low + high) / 2] are all discarded, causing us to lose floor((low + high) / 2) – low + 1 >= (low + high) / 2 – low = (high – low) / 2 elements.

If the bottom branch executes, then the elements in the range [(low + high) / 2 + 1, high] are all discarded. This loses us high – floor(low + high) / 2 + 1 >= high – (low + high) / 2 = (high – low) / 2 elements.

Consequently, we'll end up finding the first element greater than the target in O(lg n) iterations of this process.

```
 // Java program to find first element that
// is strictly greater than given target.

class GfG {
    private static int next(int[] arr, int target)
    {
        int start = 0, end = arr.length - 1;

        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;

            // Move to right side if target is
            // greater.
            if (arr[mid] <= target) {
                start = mid + 1;
            }

            // Move left side.
            else {
                ans = mid;
                end = mid - 1;
            }
        }
        return ans;
    }

    // Driver code
    public static void main(String[] args)
    {
        int[] arr = { 1, 2, 3, 5, 8, 12 };
        System.out.println(next(arr, 8));
    }
}
```

Output:

```
5
```

Source

https://www.geeksforgeeks.org/first-strictly-greater-element-in-a-sorted-array-in-java/

Chapter 41

First strictly smaller element in a sorted array in Java

First strictly smaller element in a sorted array in Java - GeeksforGeeks

Given a sorted array and a target value, find the first element that is strictly smaller than given element.

Examples:

```
Input : arr[] = {1, 2, 3, 5, 8, 12}
        Target = 5
Output : 2 (Index of 3)

Input : {1, 2, 3, 5, 8, 12}
        Target = 8
Output : 3 (Index of 5)

Input : {1, 2, 3, 5, 8, 12}
        Target = 15
Output : -1
```

A **simple solution** is to linearly traverse given array and find first element that is strictly greater. If no such element exists, then return -1.

An **efficient solution** is to useBinary Search. In a general binary search, we are looking for a value which appears in the array. Sometimes, however, we need to find the first element which is either greater than a target.

To see that this algorithm is correct, consider each comparison being made. If we find an element that's greater than the target element, then it and everything above it can't possibly match, so there's no need to search that region. We can thus search the left half. If we find an element that is smaller than the element in question, then anything before it must also be larger, so they can't be the first element that's smaller and so we don't need to search them. The middle element is thus the last possible place it could be.

Note that on each iteration we drop off at least half the remaining elements from consideration. If the top branch executes, then the elements in the range [low, (low + high) / 2] are all discarded, causing us to lose floor((low + high) / 2) – low + 1 >= (low + high) / 2 – low = (high – low) / 2 elements.

If the bottom branch executes, then the elements in the range [(low + high) / 2 + 1, high] are all discarded. This loses us high – floor(low + high) / 2 + 1 >= high – (low + high) / 2 = (high – low) / 2 elements.

Consequently, we'll end up finding the first element smaller than the target in O(lg n) iterations of this process.

```
 // Java program to find first element that
// is strictly greater than given target.

class GfG {

    private static int next(int[] arr, int target)
    {
        int start = 0, end = arr.length-1;

        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;

            // Move to the left side if the target is smaller
            if (arr[mid] >= target) {
                end = mid - 1;
            }

            // Move right side
            else {
                ans = mid;
                start = mid + 1;
            }
        }
        return ans;
    }

    // Driver code
    public static void main(String[] args)
    {
        int[] arr = { 1, 2, 3, 5, 8, 12 };
        System.out.println(next(arr, 5));
    }
}
```

Output:

```
2
```

Source

https://www.geeksforgeeks.org/first-strictly-smaller-element-in-a-sorted-array-in-java/

Chapter 42

Floor in a Sorted Array

Floor in a Sorted Array - GeeksforGeeks

Given a sorted array and a value x, the floor of x is the largest element in array smaller than or equal to x. Write efficient functions to find floor of x.

Examples:

```
Input : arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 5
Output : 2
2 is the largest element in arr[] smaller than 5.

Input : arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 20
Output : 19
19 is the largest element in arr[] smaller than 20.

Input : arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 0
Output : -1
Since floor doesn't exist, output is -1.
```

Method 1 (Simple)
A simple solution is linearly traverse input sorted array and search for the first element greater than x. The element just before the found element is floor of x.

C++

```
 // C/C++ program to find floor of a given number
// in a sorted array
#include<stdio.h>

/* An inefficient function to get index of floor
of x in arr[0..n-1] */
```

```
int floorSearch(int arr[], int n, int x)
{
    // If last element is smaller than x
    if (x >= arr[n-1])
        return n-1;

    // If first element is greater than x
    if (x < arr[0])
        return -1;

    // Linearly search for the first element
    // greater than x
    for (int i=1; i<n; i++)
    if (arr[i] > x)
        return (i-1);

    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 2, 4, 6, 10, 12, 14};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 7;
    int index = floorSearch(arr, n-1, x);
    if (index == -1)
        printf("Floor of %d doesn't exist in array ", x);
    else
        printf("Floor of %d is %d", x, arr[index]);
    return 0;
}
```

Python3

```
 # Python3 program to find floor of a
# given number in a sorted array

# Function to get index of floor
# of x in arr[low..high]
def floorSearch(arr, low, high, x):

    # If low and high cross each other
    if (low > high):
        return -1

    # If last element is smaller than x
    if (x >= arr[high]):
        return high

    # Find the middle point
    mid = int((low + high) / 2)

    # If middle point is floor.
    if (arr[mid] == x):
        return mid

    # If x lies between mid-1 and mid
    if (mid > 0 and arr[mid-1] <= x
                and x < arr[mid]):
        return mid - 1

    # If x is smaller than mid,
    # floor must be in left half.
    if (x < arr[mid]):
        return floorSearch(arr, low, mid-1, x)

    # If mid-1 is not floor and x is greater than
    # arr[mid],
    return floorSearch(arr, mid+1, high, x)


# Driver Code
arr = [1, 2, 4, 6, 10, 12, 14]
n = len(arr)
x = 7
index = floorSearch(arr, 0, n-1, x)

if (index == -1):
    print("Floor of", x, "doesn't exist \
                    in array ", end = "")
else:
    print("Floor of", x, "is", arr[index])

# This code is contributed by Smitha Dinesh Semwal.
```

Output:

```
Floor of 7 is 6.
```

Time Complexity : O(n)

Method 2 (Efficient)

The idea is to use Binary Search.

C++

```
 // A C/C++ program to find floor of a given number
// in a sorted array
#include<stdio.h>

/* Function to get index of floor of x in
   arr[low..high] */
int floorSearch(int arr[], int low, int high, int x)
{
    // If low and high cross each other
    if (low > high)
        return -1;

    // If last element is smaller than x
    if (x >= arr[high])
        return high;

    // Find the middle point
    int mid = (low+high)/2;

    // If middle point is floor.
    if (arr[mid] == x)
        return mid;

    // If x lies between mid-1 and mid
    if (mid > 0 && arr[mid-1] <= x && x < arr[mid])
        return mid-1;

    // If x is smaller than mid, floor must be in
    // left half.
    if (x < arr[mid])
        return floorSearch(arr, low, mid-1, x);

    // If mid-1 is not floor and x is greater than
    // arr[mid],
    return floorSearch(arr, mid+1, high, x);
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 2, 4, 6, 10, 12, 14};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 7;
    int index = floorSearch(arr, 0, n-1, x);
    if (index == -1)
        printf("Floor of %d doesn't exist in array ", x);
    else
        printf("Floor of %d is %d", x, arr[index]);
    return 0;
}
```

Java

```
 // Java program to find floor of
// a given number in a sorted array
import java.io.*;

class GFG {

    /* Function to get index of floor of x in
    arr[low..high] */
    static int floorSearch(int arr[], int low,
                             int high, int x)
    {
        // If low and high cross each other
        if (low > high)
            return -1;

        // If last element is smaller than x
        if (x >= arr[high])
            return high;

        // Find the middle point
        int mid = (low+high)/2;

        // If middle point is floor.
        if (arr[mid] == x)
            return mid;

        // If x lies between mid-1 and mid
        if (mid > 0 && arr[mid-1] <= x && x <
                                    arr[mid])
            return mid-1;

        // If x is smaller than mid, floor
        // must be in left half.
        if (x < arr[mid])
            return floorSearch(arr, low,
                               mid - 1, x);

        // If mid-1 is not floor and x is
        // greater than arr[mid],
        return floorSearch(arr, mid + 1, high,
                                         x);
    }

    /* Driver program to check above functions */
    public static void main(String[] args)
    {
        int arr[] = {1, 2, 4, 6, 10, 12, 14};
        int n = arr.length;
        int x = 7;
        int index = floorSearch(arr, 0, n - 1,
                                          x);
        if (index == -1)
            System.out.println("Floor of " + x +
                     " dosen't exist in array ");
        else
            System.out.println("Floor of " + x +
                            " is " + arr[index]);
    }
}
// This code is contributed by Prerna Saini
```

Python3

```
 # Python3 program to find floor of a
# given number in a sorted array

# Function to get index of floor
# of x in arr[low..high]
def floorSearch(arr, low, high, x):

    # If low and high cross each other
    if (low > high):
        return -1

    # If last element is smaller than x
    if (x >= arr[high]):
        return high

    # Find the middle point
    mid = int((low + high) / 2)

    # If middle point is floor.
    if (arr[mid] == x):
        return mid

    # If x lies between mid-1 and mid
    if (mid > 0 and arr[mid-1] <= x
                and x < arr[mid]):
        return mid - 1

    # If x is smaller than mid,
    # floor must be in left half.
    if (x < arr[mid]):
        return floorSearch(arr, low, mid-1, x)

    # If mid-1 is not floor and x is greater than
    # arr[mid],
    return floorSearch(arr, mid+1, high, x)


# Driver Code
arr = [1, 2, 4, 6, 10, 12, 14]
n = len(arr)
x = 7
index = floorSearch(arr, 0, n-1, x)

if (index == -1):
    print("Floor of", x, "doesn't exist\
                    in array ", end = "")
else:
    print("Floor of", x, "is", arr[index])

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 // C# program to find floor of
// a given number in a sorted array
using System;

class GFG {

    /* Function to get index of floor of x in
    arr[low..high] */
    static int floorSearch(int []arr, int low,
                          int high, int x)
    {

        // If low and high cross each other
        if (low > high)
            return -1;

        // If last element is smaller than x
        if (x >= arr[high])
            return high;
        // Find the middle point
        int mid = (low + high) / 2;

        // If middle point is floor.
        if (arr[mid] == x)
            return mid;

        // If x lies between mid-1 and mid
        if (mid > 0 && arr[mid-1] <= x && x <
                                    arr[mid])
            return mid - 1;

        // If x is smaller than mid, floor
        // must be in left half.
        if (x < arr[mid])
            return floorSearch(arr, low,
                               mid - 1, x);

        // If mid-1 is not floor and x is
        // greater than arr[mid],
        return floorSearch(arr, mid + 1, high,
                                          x);
    }

    /* Driver program to check above functions */
    public static void Main()
    {
        int []arr = {1, 2, 4, 6, 10, 12, 14};
        int n = arr.Length;
        int x = 7;
        int index = floorSearch(arr, 0, n - 1,
                                           x);
        if (index == -1)
            Console.Write("Floor of " + x +
                     " dosen't exist in array ");
        else
            Console.Write("Floor of " + x +
                            " is " + arr[index]);
    }
}

// This code is contributed by nitin mittal.
```

Output:

```
Floor of 7 is 6.
```

Time Complexity : O(Log n)

Improved By : nitin mittal, jit_t

Source

https://www.geeksforgeeks.org/floor-in-a-sorted-array/

Chapter 43

Iterative Fast Fourier Transformation for polynomial multiplication

Iterative Fast Fourier Transformation for polynomial multiplication - GeeksforGeeks

Given two polynomials, A(x) and B(x), find the product C(x) = A(x)*B(x). In the previous postwe discussed the recursive approach to solve this problem which has O(nlogn) complexity.
Examples:

```
Input :
 A[] = {9, -10, 7, 6}
 B[] = {-5, 4, 0, -2}
Output :
 C(x) =
```

In real life applications such as signal processing, speed matters a lot, this article examines an efficient FFT implementation. This article focuses on iterative version of the FFT algorithm that runs in O(nlogn) time but can have a lower constant hidden than the recursive version plus it saves the recursion stack space.
Pre-requisite: recursive algorithm of FFT.

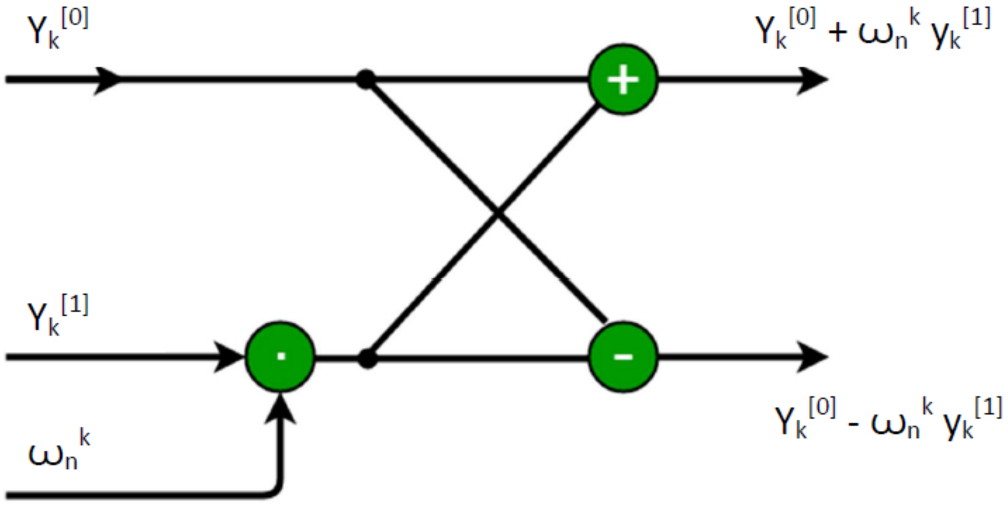
Recall the recursive-FFT pseudo code from previous post, in the for loop evaluation of y_k, [illegible] is calculated twice. We can change the loop to compute it only once, storing it in a temporary variable t. So, it becomes,

for k $\longleftarrow$ 0 to n/2 – 1

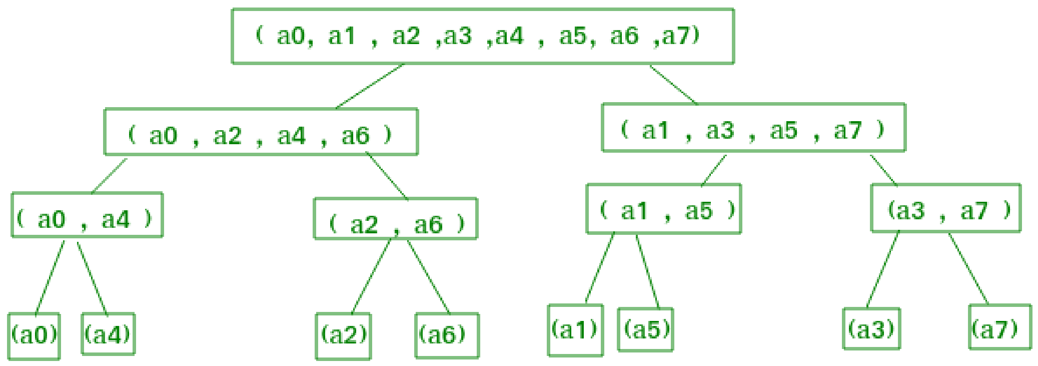
do t [illegible]

```
$y_k \leftarrow y_k^{[0]} + t$
$y_{k+(n/2)} \leftarrow y_k^{[0]} - t$
$\omega \leftarrow \omega\omega_n$
```

The operation in this loop, multiplying the twiddle factor w = ω_n^k by $y_k^{[1]}$, storing the product into t, and adding and subtracting t from $y_k^{[0]}$, is known as a butterfly operation.
Pictorially, this is what butterfly operation looks like:


$Y_k^{[0]}$
$Y_k^{[0]} + \omega_n^k y_k^{[1]}$
+
$Y_k^{[1]}$
·
-
ω_n^k
$Y_k^{[0]} - \omega_n^k y_k^{[1]}$

Let us take for n=8 and procede for formation of iterative fft algorithm. Looking at the recursion tree above, we find that if we arrange the elements of the initial coefficient vector a into the order in which they appear in the leaves, we could trace the execution of the Recusive-FFT procedure, but bottom up instead of top down. First, we take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds n/2 2-element DFTs. Next, we take these n/2 DFTs in pairs and compute the DFT of the four vector elements they come from by executing two butterfy operations, replacing two 2-element DFTs with one 4-element DFT. The vector then holds n/4 4-element DFTs. We continue in this manner until the vector holds two (n/2) element DFTs, which we combine using n/2 butterfly operations into the final n-element DFT.



To turn this bottom-up approach into code, we use an array A[0…n] that initially holds the elements of the input vector a in the order in which they appear in the leaves of the tree. Because we have to combine DFT so n each level of the tree, we introduce avariable s to count the levels, ranging from 1 (at the bottom, when we are combining pairs to form 2-element DFTs) to lgn (at the top, when weare combining two n/2 element DFTs to produce the final result). The algorithm therefore is:

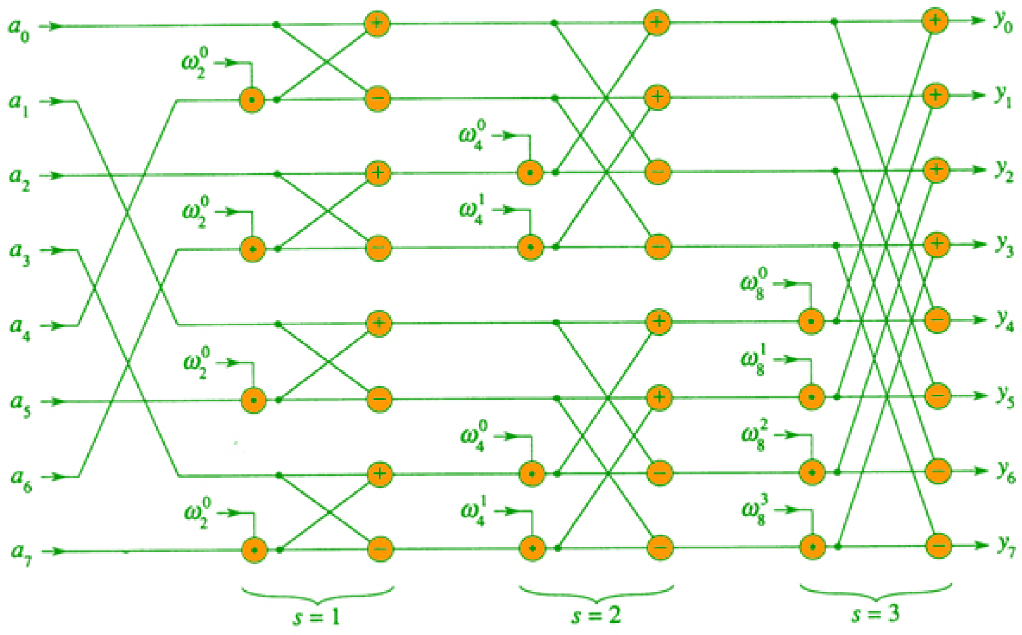
```
1. for s=1 to lgn
2.     do for k=0 to n-1 by
3.           do combine the two  -element DFTs in
                A[k...k+-1] and A[k+...k+-1]
                into one 2s-element DFT in A[k...k+-1]
```

Now for generating the code, we arrange the coefficient vector elements in the order of leaves. Example- The order in leaves 0, 4, 2, 6, 1, 5, 3, 7 is a bit reversal of the indices. Start with 000, 001, 010, 011, 100, 101, 110, 111 and reverse the bits of each binary number to obtain 000, 100, 010, 110, 001, 101, 011, 111.Pseudo code for iterative FFT :

```
BIT-REVERSE-COPY(a, A)
n = length [a]
for k = 0 to n-1
        do A[rev(k)] = a[k]

ITERATIVE-FFT
BIT-REVERSE-COPY(a, A)
n = length(a)
for s = 1 to log n
        do m=
       =
           for j = 0 to m/2-1
               do for k = j to n-1 by m
```

```
                      do t = A[k+m/2]
                         u = A[k]
                         A[k] = u+t
                         A[k+m/2] = u-t

return A
```

It will be more clear from the below parallel FFT circuit :



```
 // CPP program to implement iterative
// fast Fourier transform.
#include <bits/stdc++.h>
using namespace std;

typedef complex<double> cd;
const double PI = 3.1415926536;

// Utility function for reversing the bits
// of given index x
unsigned int bitReverse(unsigned int x, int log2n)
{
    int n = 0;
    for (int i = 0; i < log2n; i++)
    {
        n <<= 1;
        n |= (x & 1);
        x >>= 1;
    }
    return n;
}

// Iterative FFT function to compute the DFT
// of given coefficient vector
void fft(vector<cd>& a, vector<cd>& A, int log2n)
{
    int n = 4;

    // bit reversal of the given array
    for (unsigned int i = 0; i < n; ++i) {
        int rev = bitReverse(i, log2n);
        A[i] = a[rev];
    }

    // j is iota
    const complex<double> J(0, 1);
    for (int s = 1; s <= log2n; ++s) {
        int m = 1 << s; // 2 power s
        int m2 = m >> 1; // m2 = m/2 -1
        cd w(1, 0);

        // principle root of nth complex
        // root of unity.
        cd wm = exp(J * (PI / m2));
        for (int j = 0; j < m2; ++j) {
            for (int k = j; k < n; k += m) {

                // t = twiddle factor
                cd t = w * A[k + m2];
                cd u = A[k];

                // similar calculating y[k]
                A[k] = u + t;

                // similar calculating y[k+n/2]
                A[k + m2] = u - t;
            }
            w *= wm;
        }
    }
}

int main()
{
    vector<cd> a{ 1, 2, 3, 4 };
    vector<cd> A(4);
    fft(a, A, 2);
    for (int i = 0; i < 4; ++i)
        cout << A[i] << "\n";
}
```

```
Input:  1 2 3 4
Output:
(10, 0)
(-2, -2)
(-2, 0)
(-2, 2)
```

Time complexity Analysis:
The complexity is O(nlgn). To show this we show the innermost loop runs in O(nlgn) time as :

$$
\begin{aligned}
T(n) &= \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2} \\
&= \Theta(n \lg n)
\end{aligned}
$$

Source

https://www.geeksforgeeks.org/iterative-fast-fourier-transformation-polynomial-multiplication/

Chapter 44

Iterative Tower of Hanoi

Iterative Tower of Hanoi - GeeksforGeeks

Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say 'source pole') to another pole (say 'destination pole') with the help of third pole (say auxiliary pole).

The puzzle has the following two rules:

1. You can't place a larger disk onto smaller disk
2. Only one disk can be moved at a time

We've already discussed recursive solution for Tower of Hanoi. We have also seen that, for n disks, total $2^n - 1$ moves are required.

Iterative Algorithm:

```
1. Calculate the total number of moves required i.e. "pow(2, n)
   - 1" here n is number of disks.
2. If number of disks (i.e. n) is even then interchange destination
   pole and auxiliary pole.
3. for i = 1 to total number of moves:
     if i%3 == 1:
    legal movement of top disk between source pole and
        destination pole
     if i%3 == 2:
    legal movement top disk between source pole and
        auxiliary pole
     if i%3 == 0:
        legal movement top disk between auxiliary pole
        and destination pole
```

Example:

```
Let us understand with a simple example with 3 disks:
So, total number of moves required = 7

        S                       A                   D

When i= 1, (i % 3 == 1) legal movement between‘S’ and ‘D’


When i = 2,  (i % 3 == 2) legal movement between ‘S’ and ‘A’


When i = 3, (i % 3 == 0) legal movement between ‘A’ and ‘D’ ’


When i = 4, (i % 4 == 1) legal movement between ‘S’ and ‘D’


When i = 5, (i % 5 == 2) legal movement between ‘S’ and ‘A’


When i = 6, (i % 6 == 0) legal movement between ‘A’ and ‘D’


When i = 7, (i % 7 == 1) legal movement between ‘S’ and ‘D’
```

So, after all these destination pole contains all the in order of size.
After observing above iterations, we can think that after a disk other than the smallest disk is moved, the next disk to be moved must be the smallest disk because it is the top disk resting on the spare pole and there are no other choices to move a disk.

C

```
 // C Program for Iterative Tower of Hanoi
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
   unsigned capacity;
   int top;
   int *array;
};

// function to create a stack of given capacity.
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack -> capacity = capacity;
    stack -> top = -1;
    stack -> array =
        (int*) malloc(stack -> capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
   return (stack->top == stack->capacity - 1);
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
   return (stack->top == -1);
}

// Function to add an item to stack.  It increases
// top by 1
void push(struct Stack *stack, int item)
{
    if (isFull(stack))
        return;
    stack -> array[++stack -> top] = item;
}

// Function to remove an item from stack.  It
// decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack -> array[stack -> top--];
}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(struct Stack *src,
            struct Stack *dest, char s, char d)
{
    int pole1TopDisk = pop(src);
    int pole2TopDisk = pop(dest);

    // When pole 1 is empty
    if (pole1TopDisk == INT_MIN)
    {
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }

    // When pole2 pole is empty
    else if (pole2TopDisk == INT_MIN)
    {
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }

    // When top disk of pole1 > top disk of pole2
    else if (pole1TopDisk > pole2TopDisk)
    {
        push(src, pole1TopDisk);
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }

    // When top disk of pole1 < top disk of pole2
    else
    {
        push(dest, pole2TopDisk);
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
}

//Function to show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
    printf("Move the disk %d from \'%c\' to \'%c\'\n",
           disk, fromPeg, toPeg);
}

//Function to implement TOH puzzle
void tohIterative(int num_of_disks, struct Stack
             *src, struct Stack *aux,
             struct Stack *dest)
{
    int i, total_num_of_moves;
    char s = 'S', d = 'D', a = 'A';

    //If number of disks is even, then interchange
    //destination pole and auxiliary pole
    if (num_of_disks % 2 == 0)
    {
        char temp = d;
        d = a;
        a  = temp;
    }
    total_num_of_moves = pow(2, num_of_disks) - 1;

    //Larger disks will be pushed first
    for (i = num_of_disks; i >= 1; i--)
        push(src, i);

    for (i = 1; i <= total_num_of_moves; i++)
    {
        if (i % 3 == 1)
          moveDisksBetweenTwoPoles(src, dest, s, d);

        else if (i % 3 == 2)
          moveDisksBetweenTwoPoles(src, aux, s, a);

        else if (i % 3 == 0)
          moveDisksBetweenTwoPoles(aux, dest, a, d);
    }
}

// Driver Program
int main()
{
    // Input: number of disks
    unsigned num_of_disks = 3;

    struct Stack *src, *dest, *aux;

    // Create three stacks of size 'num_of_disks'
    // to hold the disks
    src = createStack(num_of_disks);
    aux = createStack(num_of_disks);
    dest = createStack(num_of_disks);

    tohIterative(num_of_disks, src, aux, dest);
    return 0;
}
```

Java

```
 // Java program for iterative
// Tower of Hanoi

public class TOH
{
    // A structure to represent a stack
    class Stack
    {
        int capacity;
        int top;
        int array[];
    }

    // function to create a stack of given capacity.
    Stack createStack(int capacity)
    {
        Stack stack=new Stack();
        stack.capacity = capacity;
        stack.top = -1;
        stack.array = new int[capacity];
        return stack;
    }

    // Stack is full when top is equal to the last index
    boolean isFull(Stack stack)
    {
        return (stack.top == stack.capacity - 1);
    }

    // Stack is empty when top is equal to -1
    boolean isEmpty(Stack stack)
    {
        return (stack.top == -1);
    }

    // Function to add an item to stack.  It increases
    // top by 1
    void push(Stack stack,int item)
    {
        if(isFull(stack))
            return;
        stack.array[++stack.top] = item;
    }

    // Function to remove an item from stack.  It
    // decreases top by 1
```

```
int pop(Stack stack)
{
    if(isEmpty(stack))
        return Integer.MIN_VALUE;
    return stack.array[stack.top--];
}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(Stack src, Stack dest,
                                        char s, char d)
{
    int pole1TopDisk = pop(src);
    int pole2TopDisk = pop(dest);

    // When pole 1 is empty
    if (pole1TopDisk == Integer.MIN_VALUE)
    {
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }
    // When pole2 pole is empty
    else if (pole2TopDisk == Integer.MIN_VALUE)
    {
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
    // When top disk of pole1 > top disk of pole2
    else if (pole1TopDisk > pole2TopDisk)
    {
        push(src, pole1TopDisk);
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }
    // When top disk of pole1 < top disk of pole2
    else
    {
        push(dest, pole2TopDisk);
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
}

// Function to show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
    System.out.println("Move the disk "+disk +
                    " from "+fromPeg+" to "+toPeg);
```

```
}

// Function to implement TOH puzzle
void tohIterative(int num_of_disks, Stack
             src, Stack aux, Stack dest)
{
    int i, total_num_of_moves;
    char s = 'S', d = 'D', a = 'A';

    // If number of disks is even, then interchange
    // destination pole and auxiliary pole
    if (num_of_disks % 2 == 0)
    {
        char temp = d;
        d = a;
        a  = temp;
    }
    total_num_of_moves = (int) (Math.pow(2, num_of_disks) - 1);

    // Larger disks will be pushed first
    for (i = num_of_disks; i >= 1; i--)
        push(src, i);

    for (i = 1; i <= total_num_of_moves; i++)
    {
        if (i % 3 == 1)
          moveDisksBetweenTwoPoles(src, dest, s, d);

        else if (i % 3 == 2)
          moveDisksBetweenTwoPoles(src, aux, s, a);

        else if (i % 3 == 0)
          moveDisksBetweenTwoPoles(aux, dest, a, d);
    }
}

// Driver Program to test above functions
public static void main(String[] args)
{

    // Input: number of disks
    int num_of_disks = 3;

    TOH ob = new TOH();
    Stack src, dest, aux;

    // Create three stacks of size 'num_of_disks'
    // to hold the disks
```

```
        src = ob.createStack(num_of_disks);
        dest = ob.createStack(num_of_disks);
        aux = ob.createStack(num_of_disks);

        ob.tohIterative(num_of_disks, src, aux, dest);
    }
}

// This code is Contibuted by Sumit Ghosh
```

Output:

```
Move the disk 1 from 'S' to 'D'
Move the disk 2 from 'S' to 'A'
Move the disk 1 from 'D' to 'A'
Move the disk 3 from 'S' to 'D'
Move the disk 1 from 'A' to 'S'
Move the disk 2 from 'A' to 'D'
Move the disk 1 from 'S' to 'D'
```

Related Articles

- Recursive Functions
- Tail recursion
- Quiz on Recursion

References:
http://en.wikipedia.org/wiki/Tower_of_Hanoi#Iterative_solution

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

Improved By : VIKASGUPTA1127

Source

https://www.geeksforgeeks.org/iterative-tower-of-hanoi/

Chapter 45

Java 8 | Arrays parallelSort() method with Examples

Java 8 | Arrays parallelSort() method with Examples - GeeksforGeeks

Java 8 introduced a new method called as **parallelSort()** in **java.util.Arrays** Class. It uses Parallel Sorting of array elements

Algorithm of parallelSort()

```
1. The array is divided into sub-arrays and that
   sub-arrays is again divided into their sub-arrays,
   until the minimum level of detail in a set of array.
2. Arrays are sorted individually by multiple thread.
3. The parallel sort uses Fork/Join Concept for sorting.
4. Sorted sub-arrays are then merged.
```

Syntax :

1. **For sorting data in ascending order :**

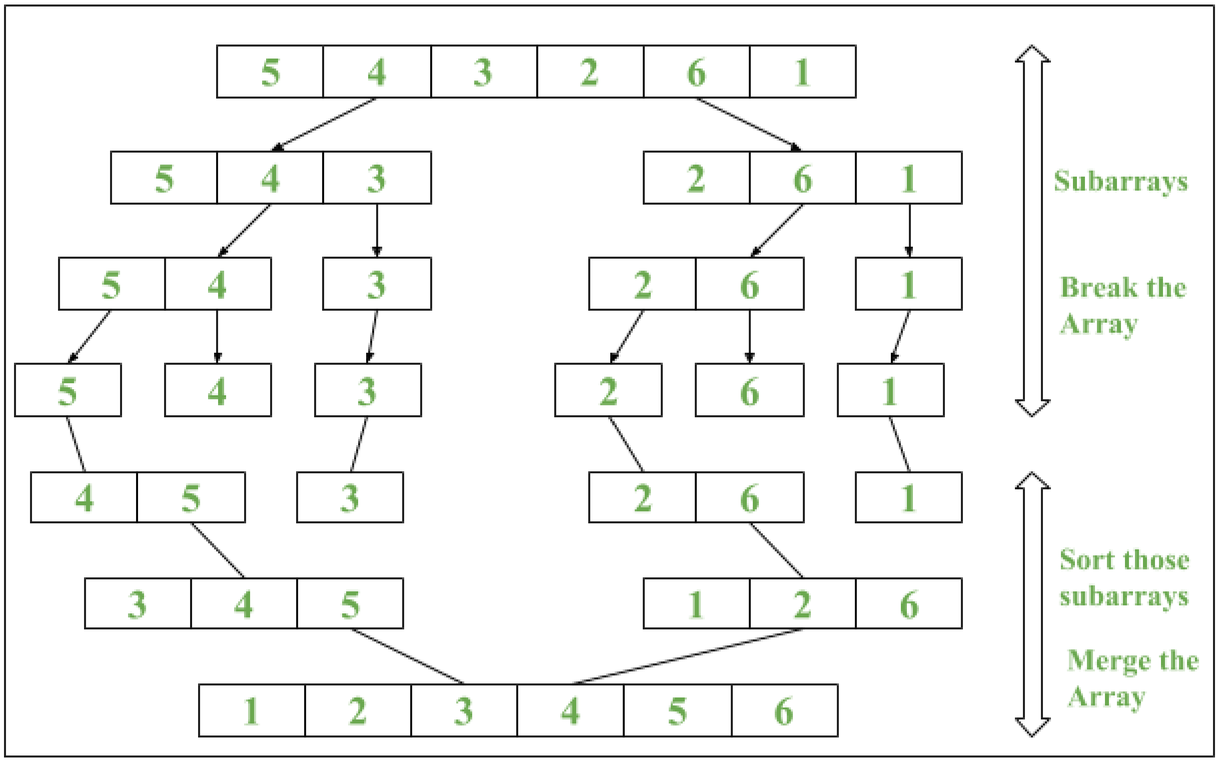
   ```
   public static void parallelSort(Object obj[])
   ```

2. **For sorting data in specified range in ascending order :**

   ```
   public static void parallelSort(Object obj[], int from, int to)
   ```

Advantage :
parallelSort() method uses concept of **MultiThreading** which makes the sorting **faster** as compared to normal sorting method.

Example



Below are the program that will illustrate the use of Arrays.parallelSort():

Program 1: To demonstrate use of Parallel Sort

```
 // Java program to demonstrate
// Arrays.parallelSort() method

import java.util.Arrays;

public class ParallelSort {
    public static void main(String[] args)
    {
        // Creating an array
        int numbers[] = { 9, 8, 7, 6, 3, 1 };

        // Printing unsorted Array
        System.out.print("Unsorted Array: ");
        // Iterating the Elements using stream
        Arrays.stream(numbers)
            .forEach(n -> System.out.print(n + " "));
        System.out.println();
        // Using Arrays.parallelSort()
        Arrays.parallelSort(numbers);

        // Printing sorted Array
        System.out.print("Sorted Array: ");
        // Iterating the Elements using stream
        Arrays.stream(numbers)
            .forEach(n -> System.out.print(n + " "));
    }
}
```

Output:

```
Unsorted Array: 9 8 7 6 3 1
Sorted Array: 1 3 6 7 8 9
```

Time Complexity is O(nlogn)

Program 2: To demonstrate use of Parallel Sort w.r.t. Series Sort (Normal Sort)

```
 // Java program to demonstrate impact
// of Parallel Sort vs Serial Sort

import java.util.Arrays;
import java.util.Random;

public class ParallelSort {
    public static void main(String[] args)
    {
        // Creating an array
        int numbers[] = new int[100];

        // Iterating Loop till i = 1000
        // with interval of 10
        for (int i = 0; i < 1000; i += 10) {

            System.out.println("\nFor iteration number: "
                               + (i / 10 + 1));

            // Random Int Array Generation
            Random rand = new Random();

            for (int j = 0; j < 100; j++) {
                numbers[j] = rand.nextInt();
            }

            // Start and End Time of Arrays.sort()
            long startTime = System.nanoTime();

            // Performing Serial Sort
            Arrays.sort(numbers);

            long endTime = System.nanoTime();

            // Printing result of Serial Sort
            System.out.println("Start and End Time in Serial (in ns): "
                            + startTime + ":" + endTime);
            System.out.println("Time taken by Serial Sort(in ns): "
                            + (endTime - startTime));

            // Start and End Time of Arrays.parallelSort()
            startTime = System.nanoTime();

            // Performing Parallel Sort
            Arrays.parallelSort(numbers);

            endTime = System.nanoTime();

            // Printing result of Parallel Sort
            System.out.println("Start and End Time in parallel (in ns): "
                            + startTime + ":" + endTime);
            System.out.println("Time taken by Parallel Sort(in ns): "
                            + (endTime - startTime));
            System.out.println();
        }
    }
}
```

Output:

```


For iteration number: 1
Start and End Time in Serial (in ns): 3951000637977:3951000870361
Time taken by Serial Sort(in ns): 232384
Start and End Time in parallel (in ns): 3951000960823:3951000971044
Time taken by Parallel Sort(in ns): 10221


For iteration number: 2
Start and End Time in Serial (in ns): 3951001142284:3951001201757
Time taken by Serial Sort(in ns): 59473
Start and End Time in parallel (in ns): 3951001256643:3951001264039
Time taken by Parallel Sort(in ns): 7396
.
.
```

```
.
For iteration number: 99
Start and End Time in Serial (in ns): 3951050723541:3951050731520
Time taken by Serial Sort(in ns): 7979
Start and End Time in parallel (in ns): 3951050754238:3951050756130
Time taken by Parallel Sort(in ns): 1892


For iteration number: 100
Start and End Time in Serial (in ns): 3951050798392:3951050804741
Time taken by Serial Sort(in ns): 6349
Start and End Time in parallel (in ns): 3951050828544:3951050830582
Time taken by Parallel Sort(in ns): 2038
```

Note : Different time intervals will be printed But parallel sort will be done before normal sort.

Environment: 2.6 GHz Intel Core i7, java version 8

Source

https://www.geeksforgeeks.org/java-8-arrays-parallelsort-method-with-examples/

Chapter 46

K-th Element of Two Sorted Arrays

K-th Element of Two Sorted Arrays - GeeksforGeeks

Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the k'th position of the final sorted array.

Examples:

```
Input : Array 1 - 2 3 6 7 9
        Array 2 - 1 4 8 10
        k = 5
Output : 6
Explanation: The final sorted array would be -
1, 2, 3, 4, 6, 7, 8, 9, 10
The 5th element of this array is 6.
Input : Array 1 - 100 112 256 349 770
        Array 2 - 72 86 113 119 265 445 892
        k = 7
Output : 256
Explanation: Final sorted array is -
72, 86, 100, 112, 113, 119, 256, 265, 349, 445, 770, 892
7th element of this array is 256.
```

Basic Approach
Since we are given two sorted arrays, we can use merging technique to get the final merged array. From this, we simply go to the k'th index.

C++

```
 // Program to find kth element from two sorted arrays
#include <iostream>
using namespace std;

int kth(int arr1[], int arr2[], int m, int n, int k)
{
    int sorted1[m + n];
    int i = 0, j = 0, d = 0;
    while (i < m && j < n)
    {
        if (arr1[i] < arr2[j])
            sorted1[d++] = arr1[i++];
        else
            sorted1[d++] = arr2[j++];
    }
    while (i < m)
        sorted1[d++] = arr1[i++];
    while (j < n)
        sorted1[d++] = arr2[j++];
    return sorted1[k - 1];
}

int main()
{
    int arr1[5] = {2, 3, 6, 7, 9};
    int arr2[4] = {1, 4, 8, 10};
    int k = 5;
    cout << kth(arr1, arr2, 5, 4, k);
    return 0;
}
```

Java

```
 // Java Program to find kth element
// from two sorted arrays

class Main
{
    static int kth(int arr1[], int arr2[], int m, int n, int k)
    {
        int[] sorted1 = new int[m + n];
        int i = 0, j = 0, d = 0;
        while (i < m && j < n)
        {
            if (arr1[i] < arr2[j])
                sorted1[d++] = arr1[i++];
            else
                sorted1[d++] = arr2[j++];
        }
        while (i < m)
            sorted1[d++] = arr1[i++];
        while (j < n)
            sorted1[d++] = arr2[j++];
        return sorted1[k - 1];
    }

    // main function
    public static void main (String[] args)
    {
        int arr1[] = {2, 3, 6, 7, 9};
        int arr2[] = {1, 4, 8, 10};
        int k = 5;
        System.out.print(kth(arr1, arr2, 5, 4, k));
    }
}

/* This code is contributed by Harsh Agarwal */
```

Python3

```
 # Program to find kth element
# from two sorted arrays

def kth(arr1, arr2, m, n, k):

    sorted1 = [0] * (m + n)
    i = 0
    j = 0
    d = 0
    while (i < m and j < n):

        if (arr1[i] < arr2[j]):
            sorted1[d] = arr1[i]
            i += 1
        else:
            sorted1[d] = arr2[j]
            j += 1
        d += 1

    while (i < m):
        sorted1[d] = arr1[i]
        d += 1
        i += 1
    while (j < n):
        sorted1[d] = arr2[j]
        d += 1
        j += 1
    return sorted1[k - 1]

# driver code
arr1 = [2, 3, 6, 7, 9]
arr2 = [1, 4, 8, 10]
k = 5;
print(kth(arr1, arr2, 5, 4, k))

# This code is contributed by Smitha Dinesh Semwal
```

C#

```
 // C# Program to find kth element
// from two sorted arrays
class GFG
{
static int kth(int[] arr1, int[] arr2,
               int m, int n, int k)
{
    int[] sorted1 = new int[m + n];
    int i = 0, j = 0, d = 0;
    while (i < m && j < n)
    {
        if (arr1[i] < arr2[j])
            sorted1[d++] = arr1[i++];
        else
            sorted1[d++] = arr2[j++];
    }
    while (i < m)
        sorted1[d++] = arr1[i++];
    while (j < n)
        sorted1[d++] = arr2[j++];
    return sorted1[k - 1];
}

// Driver Code
static void Main()
{
    int[] arr1 = {2, 3, 6, 7, 9};
    int[] arr2 = {1, 4, 8, 10};
    int k = 5;
    System.Console.WriteLine(kth(arr1, arr2,
                                 5, 4, k));
}
}

// This code is contributed by mits
```

PHP

```
 <?php
// Program to find kth
// element from two
// sorted arrays

function kth($arr1, $arr2,
             $m, $n, $k)
{
    $sorted1[$m + $n] = 0;
    $i = 0;
    $j = 0;
    $d = 0;
    while ($i < $m && $j < $n)
    {
        if ($arr1[$i] < $arr2[$j])
            $sorted1[$d++] = $arr1[$i++];
        else
            $sorted1[$d++] = $arr2[$j++];
    }
    while ($i < $m)
        $sorted1[$d++] = $arr1[$i++];
    while ($j < $n)
        $sorted1[$d++] = $arr2[$j++];
    return $sorted1[$k - 1];
}

// Driver Code
$arr1 = array(2, 3, 6, 7, 9);
$arr2 = array(1, 4, 8, 10);
$k = 5;
echo kth($arr1, $arr2, 5, 4, $k);

// This code is contributed
// by ChitraNayal
?>
```

Output:

```
6
```

Time Complexity: O(n)
Auxiliary Space : O(m + n)

Divide And Conquer Approach 1
While the previous method works, can we make our algorithm more efficient? The answer

is yes. By using a divide and conquer approach, similar to the one used in binary search, we can attempt to find the k'th element in a more efficient way.

```
Explanation:
We compare the middle elements of arrays arr1 and arr2,
let us call these indices mid1 and mid2 respectively.

Let us assume arr1[mid1]  k, then clearly the elements after
mid2 cannot be the required element. We then set the last
element of arr2 to be arr2[mid2].

In this way, we define a new subproblem with half the size
of one of the arrays.
```

C++

```
 // Program to find k-th element from two sorted arrays
#include <iostream>
using namespace std;

int kth(int *arr1, int *arr2, int *end1, int *end2, int k)
{
    if (arr1 == end1)
        return arr2[k];
    if (arr2 == end2)
        return arr1[k];
    int mid1 = (end1 - arr1) / 2;
    int mid2 = (end2 - arr2) / 2;
    if (mid1 + mid2 < k)
    {
        if (arr1[mid1] > arr2[mid2])
            return kth(arr1, arr2 + mid2 + 1, end1, end2,
                k - mid2 - 1);
        else
            return kth(arr1 + mid1 + 1, arr2, end1, end2,
                k - mid1 - 1);
    }
    else
    {
        if (arr1[mid1] > arr2[mid2])
            return kth(arr1, arr2, arr1 + mid1, end2, k);
        else
            return kth(arr1, arr2, end1, arr2 + mid2, k);
    }
}
```

```
int main()
{
    int arr1[5] = {2, 3, 6, 7, 9};
    int arr2[4] = {1, 4, 8, 10};

    int k = 5;
    cout << kth(arr1, arr2, arr1 + 5, arr2 + 4,  k - 1);
    return 0;
}
```

Output:

```
6
```

Note that in the above code, k is 0 indexed, which means if we want a k that's 1 indexed, we have to subtract 1 when passing it to the function.
Time Complexity: O(log n + log m)

Divide And Conquer Approach 2
While the above implementation is very efficient, we can still get away with making it more efficient. Instead of dividing the array into segments of n / 2 and m / 2 then recursing, we can divide them both by k / 2 and recurse. Below implementation displays this.

```
Explanation:
Instead of comparing the middle element of the arrays,
we compare the k / 2th element.
Let arr1 and arr2 be the arrays.
Now, if arr1[k / 2]  arr1[1]

New subproblem:
Array 1 - 6 7 9
Array 2 - 1 4 8 10
k = 5 - 2 = 3

floor(k / 2) = 1
arr1[1] = 6
arr2[1] = 1
arr1[1] > arr2[1]

New subproblem:
Array 1 - 6 7 9
Array 2 - 4 8 10
k = 3 - 1 = 2

floor(k / 2) = 1
arr1[1] = 6
```

```
arr2[1] = 4
arr1[1] > arr2[1]

New subproblem:
Array 1 - 6 7 9
Array 2 - 8 10
k = 2 - 1 = 1

Now, we directly compare first elements,
since k = 1.
arr1[1] < arr2[1]
Hence, arr1[1] = 6 is the answer.
```

C++

```
 // Program to find kth element from two sorted arrays
#include <iostream>
using namespace std;

int kth(int arr1[], int arr2[], int m, int n, int k,
                           int st1 = 0, int st2 = 0)
{
    // In case we have reached end of array 1
    if (st1 == m)
        return arr2[st2 + k - 1];

    // In case we have reached end of array 2
    if (st2 == n)
        return arr1[st1 + k - 1];

    // k should never reach 0 or exceed sizes
    // of arrays
    if (k == 0 || k > (m - st1) + (n - st2))
        return -1;

    // Compare first elements of arrays and return
    if (k == 1)
        return (arr1[st1] < arr2[st2]) ?
            arr1[st1] : arr2[st2];
    int curr = k / 2;

    // Size of array 1 is less than k / 2
    if (curr - 1 >= m - st1)
    {
        // Last element of array 1 is not kth
        // We can directly return the (k - m)th
        // element in array 2
        if (arr1[m - 1] < arr2[st2 + curr - 1])
```

```
            return arr2[st2 + (k - (m - st1) - 1)];
        else
            return kth(arr1, arr2, m, n, k - curr,
                st1, st2 + curr);
    }

    // Size of array 2 is less than k / 2
    if (curr-1 >= n-st2)
    {
        if (arr2[n - 1] < arr1[st1 + curr - 1])
            return arr1[st1 + (k - (n - st2) - 1)];
        else
            return kth(arr1, arr2, m, n, k - curr,
                st1 + curr, st2);
    }
    else
    {
        // Normal comparison, move starting index
        // of one array k / 2 to the right
        if (arr1[curr + st1 - 1] < arr2[curr + st2 - 1])
            return kth(arr1, arr2, m, n, k - curr,
                st1 + curr, st2);
        else
            return kth(arr1, arr2, m, n, k - curr,
                st1, st2 + curr);
    }
}

// Driver code
int main()
{
    int arr1[5] = {2, 3, 6, 7, 9};
    int arr2[4] = {1, 4, 8, 10};

    int k = 5;
    cout << kth(arr1, arr2, 5, 4,  k);
    return 0;
}
```

Output:

```
6
```

Time Complexity: O(log k)

Now, k can take a maximum value of m + n. This means that log k can be in the worst case, log(m + n). Logm + logn = log(mn) by properties of logarithms, and when m, n > 2, log(m

+ n) < log(mn). Thus this algorithm slightly outperforms the previous algorithm.Also see another simple implemented log k approach suggested by **Raj Kumar.**

C++

```
 // C++ Program to find kth element from two sorted arrays
// Time Complexity: O(log k)

#include <iostream>
using namespace std;

int kth(int arr1[], int m, int arr2[], int n, int k)
{

  if (k > (m+n) || k < 1) return -1;

  // let m <= n
  if (m > n) return kth(arr2, n, arr1, m, k);

  // if arr1 is empty returning k-th element of arr2
  if (m == 0) return arr2[k - 1];

  // if k = 1 return minimum of first two elements of both arrays
  if (k == 1) return min(arr1[0], arr2[0]);

  // now the divide and conquer part
  int i = min(m, k / 2), j = min(n, k / 2);

  if (arr1[i - 1] > arr2[j - 1] )
    // Now we need to find only k-j th element since we have found out the lowest j
    return kth(arr1, m, arr2 + j, n - j, k - j);
  else
    // Now we need to find only k-i th element since we have found out the lowest i
    return kth(arr1 + i, m - i, arr2, n, k - i);
}

// Driver code
int main()
{
    int arr1[5] = {2, 3, 6, 7, 9};
    int arr2[4] = {1, 4, 8, 10};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);
    int k = 5;

    int ans = kth(arr1,m,arr2, n, k);

    if(ans == -1) cout<<"Invalid query";
    else cout<<ans;
    return 0;
}
// This code is contributed by Raj Kumar
```

Improved By : ChitraNayal, Mithun Kumar

Source

https://www.geeksforgeeks.org/k-th-element-two-sorted-arrays/

Chapter 47

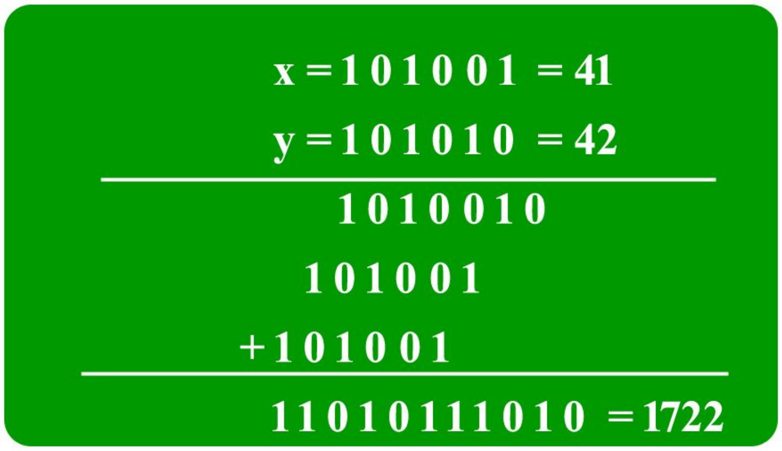
Karatsuba algorithm for fast multiplication using Divide and Conquer algorithm

Karatsuba algorithm for fast multiplication using Divide and Conquer algorithm - GeeksforGeeks

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is "1100" and second bit string is "1010", output should be 120.

For simplicity, let the length of two strings be same and be n.

A **Naive Approach** is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes O(n^2) time.



Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y.

For simplicity let us assume that n is even

```
X =  Xl*2n/2 + Xr    [Xl and Xr contain leftmost and rightmost n/2 bits of X]
Y =  Yl*2n/2 + Yr    [Yl and Yr contain leftmost and rightmost n/2 bits of Y]
```

The product XY can be written as following.

```
XY = (Xl*2n/2 + Xr)(Yl*2n/2 + Yr)
   = 2n XlYl + 2n/2(XlYr + XrYl) + XrYr
```

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into for sub-problems of size n/2. But that doesn't help because solution of recurrence T(n) = 4T(n/2) + O(n) is O(n^2). The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

```
XlYr + XrYl = (Xl + Xr)(Yl + Yr) - XlYl- XrYr
```

So the final value of XY becomes

```
XY = 2n XlYl + 2n/2 * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr
```

With above trick, the recurrence becomes T(n) = 3T(n/2) + O(n) and solution of this recurrence is $O(n^{1.59})$.

What if the lengths of input strings are different and are not even? To handle the different length case, we append 0's in the beginning. To handle odd length, we put *floor(n/2)* bits in left half and *ceil(n/2)* bits in right half. So the expression for XY changes to following.

```
XY = 22ceil(n/2) XlYl + 2ceil(n/2) * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr
```

The above algorithm is called Karatsuba algorithm and it can be used for any base.

Following is C++ implementation of above algorithm.

```
 // C++ implementation of Karatsuba algorithm for bit string multiplication.
#include<iostream>
#include<stdio.h>

using namespace std;

// FOLLOWING TWO FUNCTIONS ARE COPIED FROM http://goo.gl/q0OhZ
// Helper method: given two unequal sized bit strings, converts them to
// same length by adding leading 0s in the smaller string. Returns the
// the new length
int makeEqualLength(string &str1, string &str2)
{
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2)
    {
        for (int i = 0 ; i < len2 - len1 ; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2)
    {
        for (int i = 0 ; i < len1 - len2 ; i++)
            str2 = '0' + str2;
    }
    return len1; // If len1 >= len2
}

// The main function that adds two bit sequences and returns the addition
string addBitStrings( string first, string second )
{
    string result;  // To store the sum bits

    // make the lengths same before adding
    int length = makeEqualLength(first, second);
    int carry = 0;  // Initialize carry

    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--)
    {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (char)sum + result;

        // boolean expression for 3-bit addition
        carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&carry);
    }

    // if overflow, then add a leading 1
    if (carry)  result = '1' + result;

    return result;
}
```

```
// A utility function to multiply single bits of strings a and b
int multiplyiSingleBit(string a, string b)
{  return (a[0] - '0')*(b[0] - '0');  }

// The main function that multiplies two bit strings X and Y and returns
// result as long integer
long int multiply(string X, string Y)
{
    // Find the maximum of lengths of x and Y and make length
    // of smaller string same as that of larger string
    int n = makeEqualLength(X, Y);

    // Base cases
    if (n == 0) return 0;
    if (n == 1) return multiplyiSingleBit(X, Y);

    int fh = n/2;   // First half of string, floor(n/2)
    int sh = (n-fh); // Second half of string, ceil(n/2)

    // Find the first half and second half of first string.
    // Refer http://goo.gl/lLmgn for substr method
    string Xl = X.substr(0, fh);
    string Xr = X.substr(fh, sh);

    // Find the first half and second half of second string
    string Yl = Y.substr(0, fh);
    string Yr = Y.substr(fh, sh);

    // Recursively calculate the three products of inputs of size n/2
    long int P1 = multiply(Xl, Yl);
    long int P2 = multiply(Xr, Yr);
    long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));

    // Combine the three products to get the final result.
    return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}

// Driver program to test aboev functions
int main()
{
    printf ("%ld\n", multiply("1100", "1010"));
    printf ("%ld\n", multiply("110", "1010"));
    printf ("%ld\n", multiply("11", "1010"));
    printf ("%ld\n", multiply("1", "1010"));
    printf ("%ld\n", multiply("0", "1010"));
    printf ("%ld\n", multiply("111", "111"));
    printf ("%ld\n", multiply("11", "11"));
}
```

Output:

```
120
60
30
10
0
49
9
```

Time Complexity: Time complexity of the above solution is O(n$^{1.59}$).

Time complexity of multiplication can be further improved using another Divide and Conquer algorithm, fast Fourier transform. We will soon be discussing fast Fourier transform as a separate post.

Exercise
The above program returns a long int value and will not work for big strings. Extend the above program to return a string instead of a long int value.

Related Article :
Multiply Large Numbers Represented as Strings

References:
Wikipedia page for Karatsuba algorithm
Algorithms 1st Edition by Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani
http://courses.csail.mit.edu/6.006/spring11/exams/notes3-karatsuba
http://www.cc.gatech.edu/~ninamf/Algos11/lectures/lect0131.pdf

Source

https://www.geeksforgeeks.org/karatsuba-algorithm-for-fast-multiplication-using-divide-and-conquer-algorithm/

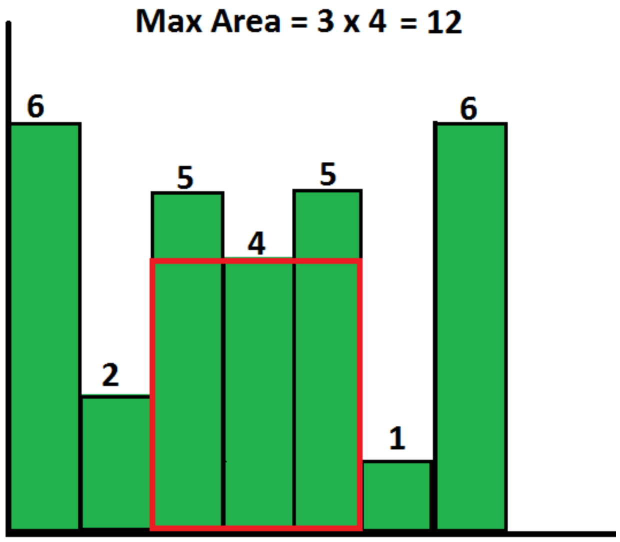
Chapter 48

Largest Rectangular Area in a Histogram | Set 1

Largest Rectangular Area in a Histogram | Set 1 - GeeksforGeeks

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 2, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red)



A **simple solution** is to one by one consider all bars as starting points and calculate area of all rectangles starting with every bar. Finally return maximum of all possible areas. Time complexity of this solution would be O(n^2).

We can use **Divide and Conquer** to solve this in O(nLogn) time. The idea is to find the minimum value in the given array. Once we have index of the minimum value, the max area is maximum of following three values.
a) Maximum area in left side of minimum value (Not including the min value)
b) Maximum area in right side of minimum value (Not including the min value)
c) Number of bars multiplied by minimum value.
The areas in left and right of minimum value bar can be calculated recursively. If we use linear search to find the minimum value, then the worst case time complexity of this algorithm becomes O(n^2). In worst case, we always have (n-1) elements in one side and 0 elements in other side and if the finding minimum takes O(n) time, we get the recurrence similar to worst case of Quick Sort.
How to find the minimum efficiently? Range Minimum Query using Segment Tree can be used for this. We build segment tree of the given histogram heights. Once the segment tree is built, all range minimum queries take O(Logn) time. So over all complexity of the algorithm becomes.

Overall Time = Time to build Segment Tree + Time to recursively find maximum area

Time to build segment tree is O(n). Let the time to recursively find max area be T(n). It can be written as following.
T(n) = O(Logn) + T(n-1)
The solution of above recurrence is O(nLogn). So overall time is O(n) + O(nLogn) which is O(nLogn).

Following is C++ implementation of the above algorithm.

```
 // A Divide and Conquer Program to find maximum rectangular area in a histogram
#include <math.h>
#include <limits.h>
#include <iostream>
using namespace std;

// A utility function to find minimum of three integers
int max(int x, int y, int z)
{  return max(max(x, y), z); }

// A utility function to get minimum of two numbers in hist[]
int minVal(int *hist, int i, int j)
{
    if (i == -1) return j;
    if (j == -1) return i;
    return (hist[i] < hist[j])? i : j;
}

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e)
{   return s + (e -s)/2; }
```

```
/*  A recursive function to get the index of minimum value in a given range of
    indexes. The following are parameters for this function.

    hist   --> Input array for which segment tree is built
    st    --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially 0 is
             passed as root is always at index 0
    ss & se  --> Starting and ending indexes of the segment represented by
                 current node, i.e., st[index]
    qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int *hist, int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return the
    // min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(hist, RMQUtil(hist, st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(hist, st, mid+1, se, qs, qe, 2*index+2));
}

// Return index of minimum element in range from index qs (quey start) to
// qe (query end).  It mainly uses RMQUtil()
int RMQ(int *hist, int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        cout << "Invalid Input";
        return -1;
    }

    return RMQUtil(hist, st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for hist[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int hist[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
       return (st[si] = ss);

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] =  minVal(hist, constructSTUtil(hist, ss, mid, st, si*2+1),
                     constructSTUtil(hist, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int hist[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); //Height of segment tree
    int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(hist, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// A recursive function to find the maximum rectangular area.
// It uses segment tree 'st' to find the minimum value in hist[l..r]
int getMaxAreaRec(int *hist, int *st, int n, int l, int r)
{
    // Base cases
    if (l > r)  return INT_MIN;
    if (l == r)  return hist[l];

    // Find index of the minimum value in given range
    // This takes O(Logn)time
    int m = RMQ(hist, st, n, l, r);

    /* Return maximum of following three possible cases
       a) Maximum area in Left of min value (not including the min)
       a) Maximum area in right of min value (not including the min)
       c) Maximum area including min */
    return max(getMaxAreaRec(hist, st, n, l, m-1),
               getMaxAreaRec(hist, st, n, m+1, r),
               (r-l+1)*(hist[m]) );
}
```

```
// The main function to find max area
int getMaxArea(int hist[], int n)
{
    // Build segment tree from given array. This takes
    // O(n) time
    int *st = constructST(hist, n);

    // Use recursive utility function to find the
    // maximum area
    return getMaxAreaRec(hist, st, n, 0, n-1);
}

// Driver program to test above functions
int main()
{
    int hist[] =  {6, 1, 5, 4, 5, 2, 6};
    int n = sizeof(hist)/sizeof(hist[0]);
    cout << "Maximum area is " << getMaxArea(hist, n);
    return 0;
}
```

Output:

```
Maximum area is 12
```

This problem can be solved in linear time. See below set 2 for linear time solution.
Linear time solution for Largest Rectangular Area in a Histogram

Source

https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/

Chapter 49

Longest Common Prefix using Divide and Conquer Algorithm

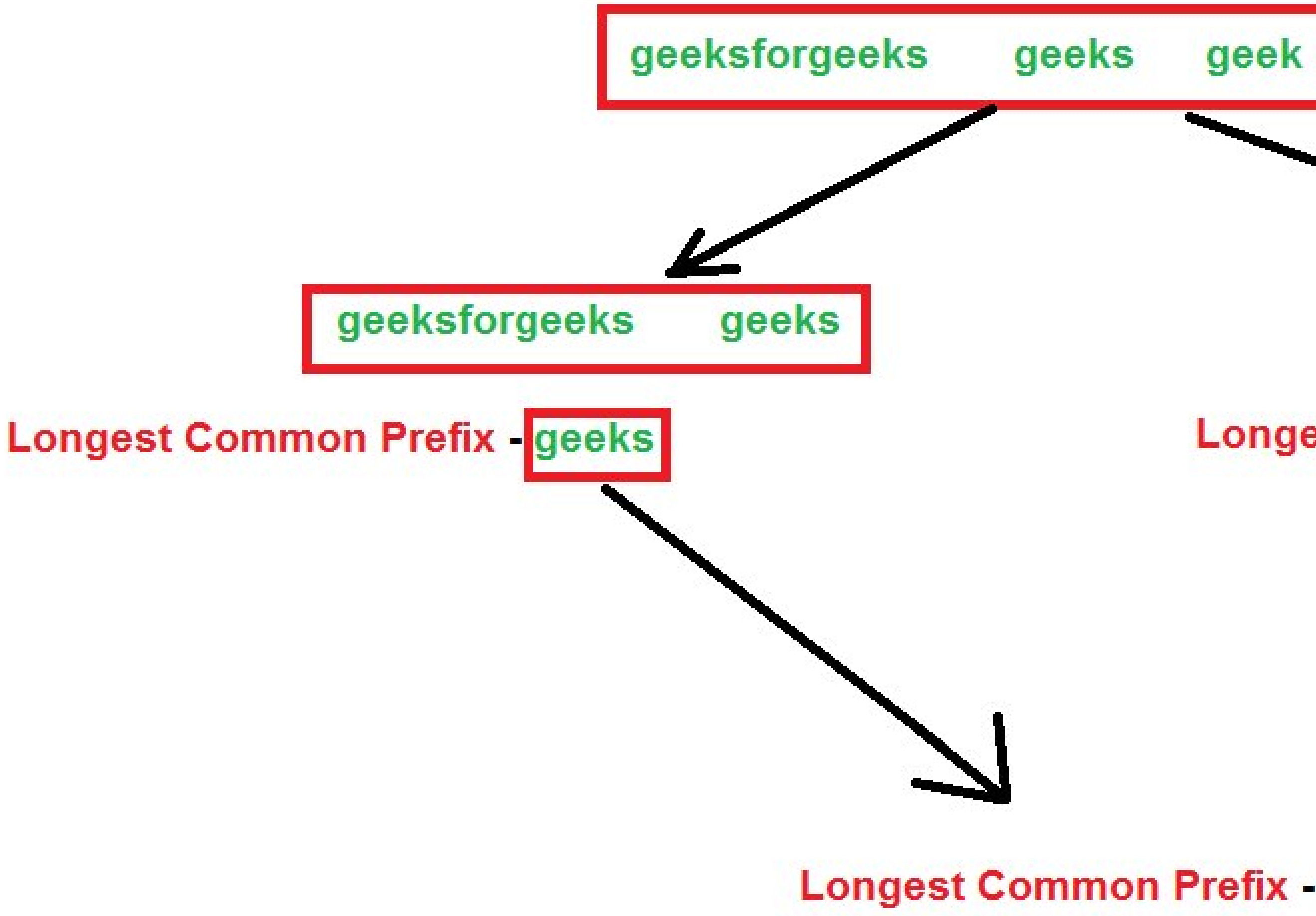
Longest Common Prefix using Divide and Conquer Algorithm - GeeksforGeeks

Given a set of strings, find the longest common prefix.

```
Input  : {"geeksforgeeks", "geeks", "geek", "geezer"}
Output : "gee"

Input  : {"apple", "ape", "april"}
Output : "ap"
```

We have discussed word by word matching and character by character matching algorithms.

In this algorithm, a divide and conquer approach is discussed. We first divide the arrays of string into two parts. Then we do the same for left part and after that for the right part. We will do it until and unless all the strings become of length 1. Now after that, we will start conquering by returning the common prefix of the left and the right strings.
The algorithm will be clear using the below illustration. We consider our strings as – "geeksforgeeks", "geeks", "geek", "geezer"



Below is C++ implementation.

```
 //  A C++ Program to find the longest common prefix
#include<bits/stdc++.h>
using namespace std;

// A Utility Function to find the common prefix between
// strings- str1 and str2
string commonPrefixUtil(string str1, string str2)
{
    string result;
    int n1 = str1.length(), n2 = str2.length();

    for (int i=0, j=0; i<=n1-1&&j<=n2-1; i++,j++)
    {
        if (str1[i] != str2[j])
            break;
        result.push_back(str1[i]);
    }
    return (result);
}

// A Divide and Conquer based function to find the
// longest common prefix. This is similar to the
// merge sort technique
string commonPrefix(string arr[], int low, int high)
{
    if (low == high)
        return (arr[low]);

    if (high > low)
    {
        // Same as (low + high)/2, but avoids overflow for
        // large low and high
        int mid = low + (high - low) / 2;

        string str1 = commonPrefix(arr, low, mid);
        string str2 = commonPrefix(arr, mid+1, high);

        return (commonPrefixUtil(str1, str2));
    }
}

// Driver program to test above function
int main()
{
    string arr[] = {"geeksforgeeks", "geeks",
                    "geek", "geezer"};
    int n = sizeof (arr) / sizeof (arr[0]);

    string ans = commonPrefix(arr, 0, n-1);

    if (ans.length())
        cout << "The longest common prefix is "
             << ans;
    else
        cout << "There is no common prefix";
    return (0);
}
```

Output :

```
The longest common prefix is gee
```

Time Complexity : Since we are iterating through all the characters of all the strings, so we can say that the time complexity is O(N M) where,

```
N = Number of strings
M = Length of the largest string string
```

Auxiliary Space : To store the longest prefix string we are allocating space which is O(M Log N).

Source

https://www.geeksforgeeks.org/longest-common-prefix-using-divide-and-conquer-algorithm/

Chapter 50

Maximum Subarray Sum using Divide and Conquer algorithm

Maximum Subarray Sum using Divide and Conquer algorithm - GeeksforGeeks

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.

C++

```
 // 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)
{
    // 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 %dn", max_sum);
   getchar();
   return 0;
}
```

Java

```
 // A Divide and Conquer based Java
// program for maximum subarray sum
// problem
import java.util.*;

class GFG {

    // Find the maximum possible sum in arr[]
    // such that arr[m] is part of it
    static int maxCrossingSum(int arr[], int l,
                              int m, int h)
    {
        // Include elements on left of mid.
        int sum = 0;
        int left_sum = Integer.MIN_VALUE;
        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 = Integer.MIN_VALUE;
        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]
    static 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 Math.max(Math.max(maxSubArraySum(arr, l, m),
                    maxSubArraySum(arr, m+1, h)),
                    maxCrossingSum(arr, l, m, h));
    }

    /* Driver program to test maxSubArraySum */
    public static void main(String[] args)
    {
    int arr[] = {2, 3, 4, 5, 7};
    int n = arr.length;
    int max_sum = maxSubArraySum(arr, 0, n-1);

    System.out.println("Maximum contiguous sum is "+
                                     max_sum);
    }
}
// This code is contributed by Prerna Saini
```

Python3

```
 # A Divide and Conquer based program
# for maximum subarray sum problem
# Find the maximum possible sum in
# arr[] auch that arr[m] is part of it
def maxCrossingSum(arr, l, m, h) :

    # Include elements on left of mid.
    sm = 0; left_sum = -10000

    for i in range(m, l-1, -1) :
        sm = sm + arr[i]

        if (sm > left_sum) :
            left_sum = sm


    # Include elements on right of mid
    sm = 0; right_sum = -1000
    for i in range(m + 1, h + 1) :
        sm = sm + arr[i]

        if (sm > right_sum) :
            right_sum = sm


    # 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]
def maxSubArraySum(arr, l, h) :

    # Base Case: Only one element
    if (l == h) :
        return arr[l]

    # Find middle point
    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 Code
arr = [2, 3, 4, 5, 7]
n = len(arr)

max_sum = maxSubArraySum(arr, 0, n-1)
print("Maximum contiguous sum is ", max_sum)

# This code is contributed by Nikita Tiwari.
```

C#

```
 // A Divide and Conquer based C#
// program for maximum subarray sum
// problem
using System;

class GFG {

    // Find the maximum possible sum in arr[]
    // such that arr[m] is part of it
    static int maxCrossingSum(int []arr, int l,
                               int m, int h)
    {
        // Include elements on left of mid.
        int sum = 0;
        int left_sum = int.MinValue;
        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.MinValue;;
        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]
    static 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 Math.Max(Math.Max(maxSubArraySum(arr, l, m),
                        maxSubArraySum(arr, m+1, h)),
                       maxCrossingSum(arr, l, m, h));
    }

    /* Driver program to test maxSubArraySum */
    public static void Main()
    {
        int []arr = {2, 3, 4, 5, 7};
        int n = arr.Length;
        int max_sum = maxSubArraySum(arr, 0, n-1);

        Console.Write("Maximum contiguous sum is "+
                                        max_sum);
    }
}

// This code is contributed by vt_m.
```

Output :

```
Maximum contiguous sum is 21
```

Time Complexity: maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \Theta(n)$
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.

Source

https://www.geeksforgeeks.org/maximum-subarray-sum-using-divide-and-conquer-algorithm/

Chapter 51

Maximum and minimum of an array using minimum number of comparisons

Maximum and minimum of an array using minimum number of comparisons - GeeksforGeeks

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;
};

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

```
 /* 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)  = 3n/2 -2
```

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

METHOD 3 (Compare in Pairs)
If n is odd then initialize min and max as first element.
If n is even then initialize min and max as minimum and maximum of the first two elements respectively.
For rest of the elements, pick them in pairs and compare their
maximum and minimum with max and min respectively.

```
 #include<stdio.h>

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

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

```
int i;

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

Improved By : kamleshbhalui

Source

https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/

Chapter 52

Median of two sorted arrays of different sizes

Median of two sorted arrays of different sizes - GeeksforGeeks

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 0: N = 0, M = 2
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 0: There are no elements in first array, return median of second array. If second array is also empty, return -1.

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 12, 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 implementation of the above approach.

C++

```
 // A C++ program to find median of two sorted arrays of
// unequal sizes
#include <bits/stdc++.h>
using namespace std;

// 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;
}

// Utility function to find median of single array
float medianSingle(int arr[], int n)
{
   if (n == 0)
      return -1;
   if (n%2 == 0)
        return (double)(arr[n/2] + arr[n/2-1])/2;
   return arr[n/2];
}

// This function assumes that N is smaller than or equal to M
// This function returns -1 if both arrays are empty
float findMedianUtil( int A[], int N, int B[], int M )
{
    // If smaller array is empty, return median from second array
    if (N == 0)
      return medianSingle(B, 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;
}
```

PHP

```
 <?php
// A PHP program to find median
// of two sorted arrays of
// unequal sizes

// A utility function to
// find median of two integers
function MO2($a, $b)
{
    return ($a + $b) / 2.0;
}
```

```
// A utility function to
// find median of three integers
function MO3($a, $b, $c)
{
    return $a + $b + $c -
       max($a, max($b, $c)) -
       min($a, min($b, $c));
}

// A utility function to find
// median of four integers
function MO4($a, $b, $c, $d)
{
    $Max = max($a, max($b, max($c, $d)));
    $Min = min($a, min($b, min( $c, $d)));
    return ($a + $b + $c + $d - $Max - $Min) / 2.0;
}

// Utility function to
// find median of single array
function medianSingle($arr, $n)
{
if ($n == 0)
    return -1;
if ($n % 2 == 0)
        return ($arr[$n / 2] +
                $arr[$n / 2 - 1]) / 2;
return $arr[$n / 2];
}

// This function assumes that N
// is smaller than or equal to M
// This function returns -1 if
// both arrays are empty
function findMedianUtil(&$A, $N, &$B, $M )
{
    // If smaller array is empty,
    // return median from second array
    if ($N == 0)
    return medianSingle($B, $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 = array(9),
        // $B = array(5, 8, 10, 20, 30)
        // and $A = array(1),
        // $B = array(5, 8, 10, 20, 30)
        if ($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]));
    }

    $idxA = ($N - 1 ) / 2;
    $idxB = ($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
function findMedian(&$A, $N,
                    &$B, $M )
{
    if ($N > $M)
    return findMedianUtil($B, $M,
                          $A, $N );

    return findMedianUtil($A, $N,
                          $B, $M );
}

// Driver Code
$A = array(900);
$B = array(5, 8, 10, 20);

$N = sizeof($A);
$M = sizeof($B);

echo findMedian( $A, $N, $B, $M );

// This code is contributed
// by ChitraNayal
?>
```

Output:

```
 10
```

Time Complexity: O(LogM + LogN)

Improved By : ChitraNayal

Source

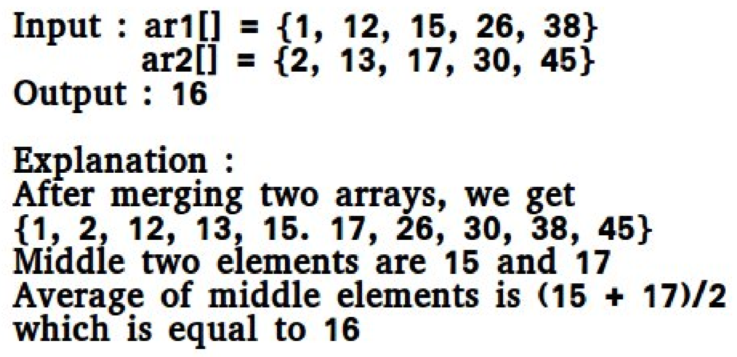
https://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/

Chapter 53

Median of two sorted arrays of same size

Median of two sorted arrays of same size - GeeksforGeeks

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)).



```
Input : ar1[] = {1, 12, 15, 26, 38}
        ar2[] = {2, 13, 17, 30, 45}
Output : 16

Explanation :
After merging two arrays, we get
{1, 2, 12, 13, 15. 17, 26, 30, 38, 45}
Middle two elements are 15 and 17
Average of middle elements is (15 + 17)/2
which is equal to 16
```

Note : Since size of the set for which we are looking for median is even (2n), we need take average of middle two numbers and return floor of the average.

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.

C++

```
// A Simple Merge based O(n)
// solution to find median of
// two sorted arrays
#include <bits/stdc++.h>
using namespace std;

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

    return (m1 + m2)/2;
}

// Driver Code
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)
        cout << "Median is "
             << getMedian(ar1, ar2, n1) ;
    else
        cout << "Doesn't work for arrays"
             << " of unequal size" ;
    getchar();
    return 0;
}

// This code is contributed
// by Shivi_Aggarwal
```

C

```
 // A Simple Merge based O(n) solution to find median of
// two sorted arrays
#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;
}
```

Java

```
 // A Simple Merge based O(n) solution
// to find median of two sorted arrays

class Main
{
    // function to calculate median
    static int getMedian(int ar1[], int ar2[], int n)
    {
        int i = 0;
        int j = 0;
        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])
            {
                /* Store the prev median */
                m1 = m2;
                m2 = ar1[i];
                i++;
            }
            else
            {
                /* Store the prev median */
                m1 = m2;
                m2 = ar2[j];
                j++;
            }
        }

        return (m1 + m2)/2;
    }

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

        int n1 = ar1.length;
        int n2 = ar2.length;
        if (n1 == n2)
            System.out.println("Median is " +
                        getMedian(ar1, ar2, n1));
        else
            System.out.println("arrays are of unequal size");
    }
}
```

Python3

```
 # A Simple Merge based O(n) Python 3 solution
# to find median of two sorted lists

# This function returns median of ar1[] and ar2[].
# Assumptions in this function:
# Both ar1[] and ar2[] are sorted arrays
# Both have n elements
def getMedian( ar1, ar2 , n):
    i = 0 # Current index of i/p list ar1[]

    j = 0 # Current index of i/p list ar2[]

    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
    count = 0
    while count < n + 1:
        count += 1

        # 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[]
        elif j == n:
            m1 = m2
            m2 = ar1[0]
            break
        if ar1[i] < ar2[j]:
            m1 = m2 # Store the prev median
            m2 = ar1[i]
            i += 1
        else:
            m1 = m2 # Store the prev median
            m2 = ar2[j]
            j += 1
    return (m1 + m2)/2
```

```
# Driver code to test above function
ar1 = [1, 12, 15, 26, 38]
ar2 = [2, 13, 17, 30, 45]
n1 = len(ar1)
n2 = len(ar2)
if n1 == n2:
    print("Median is ", getMedian(ar1, ar2, n1))
else:
    print("Doesn't work for arrays of unequal size")

# This code is contributed by "Sharad_Bhardwaj".
```

C#

```
 // A Simple Merge based O(n) solution
// to find median of two sorted arrays
using System;
class GFG
{
    // function to calculate median
    static int getMedian(int []ar1,
                         int []ar2,
                         int n)
    {
        int i = 0;
        int j = 0;
        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])
            {
                // Store the prev median
                m1 = m2;
                m2 = ar1[i];
                i++;
            }
            else
            {
                // Store the prev median
                m1 = m2;
                m2 = ar2[j];
                j++;
            }
        }

        return (m1 + m2)/2;
    }

    // Driver Code
    public static void Main ()
    {
        int []ar1 = {1, 12, 15, 26, 38};
        int []ar2 = {2, 13, 17, 30, 45};

        int n1 = ar1.Length;
        int n2 = ar2.Length;
        if (n1 == n2)
            Console.Write("Median is " +
                        getMedian(ar1, ar2, n1));
        else
            Console.Write("arrays are of unequal size");
    }
}
```

PHP

```
 <?php
// A Simple Merge based O(n) solution
// to find median of two sorted arrays

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

    // Current index of i/p array ar2[]
    $j = 0;
    $count;
    $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])
        {
            // Store the prev median
            $m1 = $m2;
            $m2 = $ar1[$i];
            $i++;
        }
        else
        {
            // Store the prev median
            $m1 = $m2;
            $m2 = $ar2[$j];
            $j++;
        }
    }

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

// Driver Code
$ar1 = array(1, 12, 15, 26, 38);
$ar2 = array(2, 13, 17, 30, 45);

$n1 = sizeof($ar1);
$n2 = sizeof($ar2);
if ($n1 == $n2)
    echo("Median is " .
          getMedian($ar1, $ar2, $n1));
else
    echo("Doesn't work for arrays".
         "of unequal size");

// This code is contributed by Ajit.
?>
```

Output :

```
Median is 16
```

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
```

Examples :

```
   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++

```
 // A divide and conquer based
// efficient solution to find
// median of two sorted arrays
// of same size.
#include<bits/stdc++.h>
using namespace std;

/* to get median of a
   sorted array */
int median(int [], int);

/* 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 -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;

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

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

    /* 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);
        return getMedian(ar1 + n / 2,
                         ar2, n - n / 2);
    }

    /* if m1 > m2 then median must
       exist in ar1[....m1] and
                ar2[m2...] */
    if (n % 2 == 0)
        return getMedian(ar2 + n / 2 - 1,
                         ar1, n - n / 2 + 1);
    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 code
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)
        cout << "Median is "
             << getMedian(ar1, ar2, n1);
    else
        cout << "Doesn't work for arrays "
             << "of unequal size";
    return 0;
}

// This code is contributed
// by Shivi_Aggarwal
```

C

```
 // A divide and conquer based efficient solution to find median
// of two sorted arrays of same size.
#include<bits/stdc++.h>
using namespace std;

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)
{
    /* 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;

    int m1 = median(ar1, n); /* get the median of the first array */
    int 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);
        return getMedian(ar1 + n/2, ar2, n - n/2);
    }

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

Output :

```
Median is 5
```

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

Median of two sorted arrays of different sizes

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

Improved By : SaumyaBhatnagar, jit_t, nitin mittal, Shivi_Aggarwal

Source

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

Chapter 54

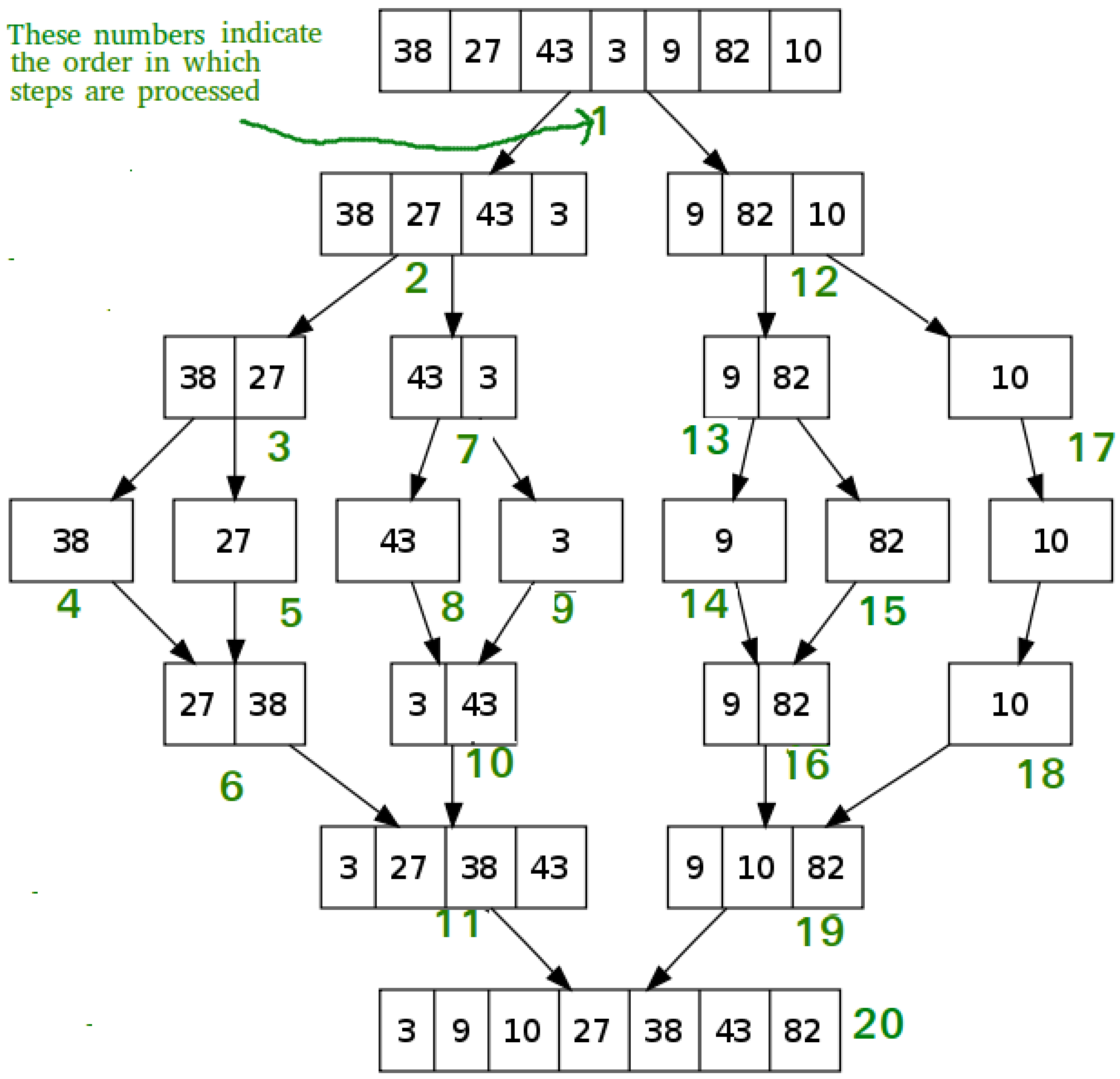
Merge Sort

Merge Sort - GeeksforGeeks

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



C/C++

```
 /* C program for Merge Sort */
#include<stdlib.h>
#include<stdio.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
```

```
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

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

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Java

```
 /* Java program for Merge Sort */
class MergeSort
{
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Create temp arrays */
        int L[] = new int [n1];
        int R[] = new int [n2];

        /*Copy data to temp arrays*/
        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];


        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        /* Copy remaining elements of L[] if any */
        while (i < n1)
        {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r)
    {
        if (l < r)
        {
            // Find the middle point
            int m = (l+r)/2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m+1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        System.out.println("Given Array");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length-1);

        System.out.println("\nSorted array");
        printArray(arr);
    }
}
/* This code is contributed by Rajat Mishra */
```

Python

```
 # Python program for implementation of MergeSort

# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r- m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0 , n1):
        L[i] = arr[l + i]

    for j in range(0 , n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray

    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted
def mergeSort(arr,l,r):
    if l < r:

        # Same as (l+r)/2, but avoids overflow for
        # large l and h
        m = (l+(r-1))/2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)


# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print ("Given array is")
for i in range(n):
    print ("%d" %arr[i]),

mergeSort(arr,0,n-1)
print ("\n\nSorted array is")
for i in range(n):
    print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra
```

C#

```
 // C# program for Merge Sort
using System;
```

```
class MergeSort {

    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int[] arr, int l,
               int m, int r)
    {

        // Find sizes of two subarrays
        // to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        // Create temp arrays
        int[] L = new int [n1];
        int[] R = new int [n2];

        // Copy data to temp arrays
        int i, j;
        for (i = 0; i < n1; ++i)
            L[i] = arr[l + i];

        for (j = 0; j < n2; ++j)
            R[j] = arr[m + 1+ j];


        // Merge the temp arrays

        // Initial indexes of first
        // and second subarrays
        i = 0;
        j = 0;

        // Initial index of merged
        // subarry array
        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }

            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements
        // of L[] if any
        while (i < n1)
        {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy remaining elements
        // of R[] if any
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts
    // arr[l..r] using merge()
    void sort(int[] arr, int l, int r)
    {
        if (l < r)
        {
            // Find the middle point
            int m = (l + r) / 2;

            // Sort first and
            // second halves
            sort(arr, l, m);
            sort(arr , m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    // A utility function to print
    // array of size n
    static void printArray(int[] arr)
    {
        int n = arr.Length;
        for (int i = 0; i < n; ++i)
            Console.Write(arr[i] + " ");

        Console.Write("\n");
    }

    // Driver Code
    public static void Main()
    {
        int[] arr = {12, 11, 13, 5, 6, 7};

        Console.Write("Given Array\n");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.Length - 1);

        Console.Write("\nSorted array\n");
        printArray(arr);
    }
}

// This code is contributed by ChitraNayal.
```

Output:

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + $\Theta(n)$
The above recurrence 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 $\Theta(nLogn)$.

Time complexity of Merge Sort is $\Theta(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

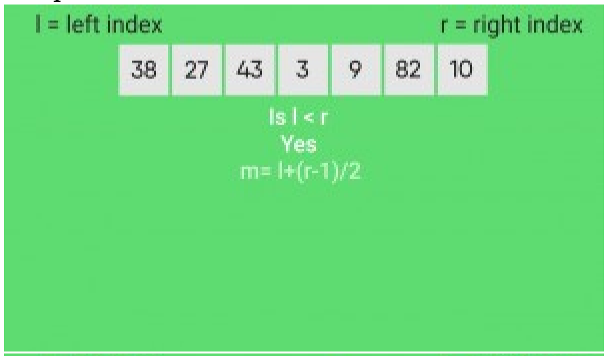
Stable: Yes

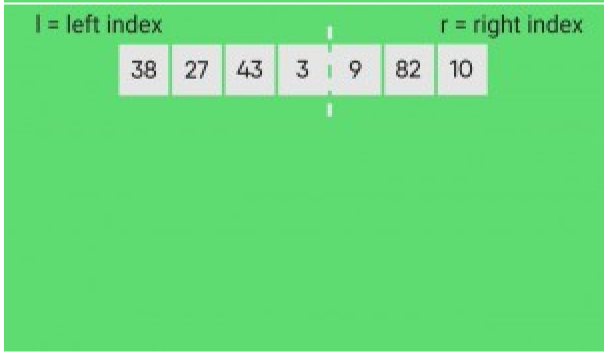
Applications of Merge Sort

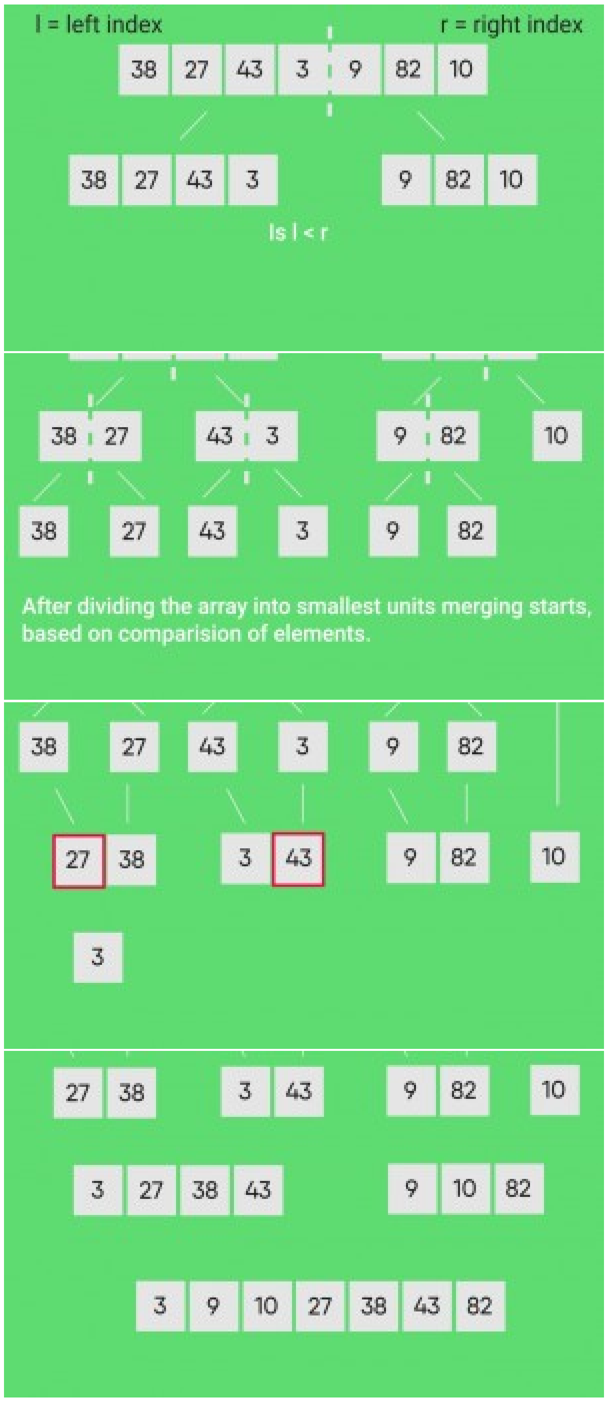
1. Merge Sort is useful for sorting linked lists in O(nLogn) time.In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

   In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.
2. Inversion Count Problem
3. Used in External Sorting

Snapshots:






l = left index
r = right index
Is l < r
After dividing the array into smallest units merging starts, based on comparision of elements.

- Recent Articles on Merge Sort
- Coding practice for sorting.
- Quiz on Merge Sort

Other Sorting Algorithms on GeeksforGeeks:
3-way Merge Sort, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort

Improved By : ChitraNayal

Source

https://www.geeksforgeeks.org/merge-sort/

Chapter 55

Merge Sort with O(1) extra space merge and O(n lg n) time

Merge Sort with O(1) extra space merge and O(n lg n) time - GeeksforGeeks

We have discussed Merge sort. How to modify the algorithm so that merge works in O(1) extra space and algorithm still works in O(n Log n) time. We may assume that the input values are integers only.

Examples:

```
Input : 5 4 3 2 1
Output : 1 2 3 4 5

Input : 999 612 589 856 56 945 243
Output : 56 243 589 612 856 945 999
```

For integer types, merge sort can be made inplace using some mathematics trick of modulus and division. That means storing two elements value at one index and can be extracted using modulus and division.
First we have to find a value greater than all the elements of the array. Now we can store the original value as modulus and the second value as division. Suppose we want to store **arr[i]** and **arr[j]** both at index i(means in arr[i]). First we have to find a **'maxval'** greater than both arr[i] and arr[j]. Now we can store as **arr[i] = arr[i] + arr[j]*maxval**. Now **arr[i]%maxval** will give the original value of arr[i] and **arr[i]/maxval** will give the value of arr[j]. So below is the implementation on merge sort.

C++

```
 // C++ program to sort an array using merge sort such
// that merge operation takes O(1) extra space.
#include <bits/stdc++.h>
```

```
using namespace std;
void merge(int arr[], int beg, int mid, int end, int maxele)
{
    int i = beg;
    int j = mid + 1;
    int k = beg;
    while (i <= mid && j <= end) {
        if (arr[i] % maxele <= arr[j] % maxele) {
            arr[k] = arr[k] + (arr[i] % maxele) * maxele;
            k++;
            i++;
        }
        else {
            arr[k] = arr[k] + (arr[j] % maxele) * maxele;
            k++;
            j++;
        }
    }
    while (i <= mid) {
        arr[k] = arr[k] + (arr[i] % maxele) * maxele;
        k++;
        i++;
    }
    while (j <= end) {
        arr[k] = arr[k] + (arr[j] % maxele) * maxele;
        k++;
        j++;
    }

    // Obtaining actual values
    for (int i = beg; i <= end; i++)
        arr[i] = arr[i] / maxele;
}

// Recursive merge sort with extra parameter, naxele
void mergeSortRec(int arr[], int beg, int end, int maxele)
{
    if (beg < end) {
        int mid = (beg + end) / 2;
        mergeSortRec(arr, beg, mid, maxele);
        mergeSortRec(arr, mid + 1, end, maxele);
        merge(arr, beg, mid, end, maxele);
    }
}

// This functions finds max element and calls recursive
// merge sort.
void mergeSort(int arr[], int n)
```

```
{
   int maxele = *max_element(arr, arr+n) + 1;
   mergeSortRec(arr, 0, n-1, maxele);
}

int main()
{
    int arr[] = { 999, 612, 589, 856, 56, 945, 243 };
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, n);

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

C#

```
 // C# program to sort an array
// using merge sort such that
// merge operation takes O(1)
// extra space.
using System;
using System.Linq;

class GFG
{
static void merge(int []arr, int beg,
                  int mid, int end,
                  int maxele)
{
    int i = beg;
    int j = mid + 1;
    int k = beg;
    while (i <= mid && j <= end)
    {
        if (arr[i] %
            maxele <= arr[j] % maxele)
        {
            arr[k] = arr[k] + (arr[i] %
                     maxele) * maxele;
            k++;
            i++;
        }
        else
        {
            arr[k] = arr[k] +
                    (arr[j] % maxele) *
                              maxele;
            k++;
            j++;
        }
    }
    while (i <= mid)
    {
        arr[k] = arr[k] + (arr[i] %
                 maxele) * maxele;
        k++;
        i++;
    }
    while (j <= end)
    {
        arr[k] = arr[k] + (arr[j] %
                 maxele) * maxele;
        k++;
        j++;
    }

    // Obtaining actual values
    for ( i = beg; i <= end; i++)
        arr[i] = arr[i] / maxele;
}

// Recursive merge sort
// with extra parameter, naxele
static void mergeSortRec(int []arr, int beg,
                         int end, int maxele)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSortRec(arr, beg,
                     mid, maxele);
        mergeSortRec(arr, mid + 1,
                     end, maxele);
        merge(arr, beg, mid,
              end, maxele);
    }
}

// This functions finds
// max element and calls
// recursive merge sort.
static void mergeSort(int []arr, int n)
```

```
{
    int maxele = arr.Max() + 1;
    mergeSortRec(arr, 0, n - 1, maxele);
}

//Driver code
public static void Main ()
{
    int []arr = {999, 612, 589,
                 856, 56, 945, 243};
    int n = arr.Length;

    mergeSort(arr, n);

    Console.WriteLine("Sorted array ");
    for (int i = 0; i < n; i++)
        Console.Write( arr[i] + " ");
}
}

// This code is contributed
// by inder_verma.
```

Output:

```
Sorted array
56 243 589 612 856 945 999
```

Improved By : inderDuMCA

Source

https://www.geeksforgeeks.org/merge-sort-with-o1-extra-space-merge-and-on-lg-n-time/

Chapter 56

Minimum difference between adjacent elements of array which contain elements from each row of a matrix

Minimum difference between adjacent elements of array which contain elements from each row of a matrix - GeeksforGeeks

Given a matrix of **N** rows and **M** columns, the task is to find the minimum absolute difference between any of the two adjacent elements of an array of size **N**, which is created by picking one element from each row of the matrix. Note the element picked from row 1 will become arr[0], element picked from row 2 will become arr[1] and so on.

Examples:

```
Input :  N = 2, M = 2
m[2][2] = { 8, 2,
            6, 8 }
Output : 0.
Picking 8 from row 1 and picking 8 from row 2, we create an array { 8, 8 } and minimum
difference between any of adjacent element is 0.

Input :  N = 3, M = 3
m[3][3] = { 1, 2, 3
            4, 5, 6
            7, 8, 9 }
Output : 1.
```

The idea is to sort all rows individually and then do binary search to find the closest element in next row for each element.

To do this in an efficient manner, sort each row of the matrix. Starting from row 1 to row N – 1 of matrix, for each element m[i][j] of current row in the matrix, find the smallest element in the next row which is greater than or equal to the current element, say p and the largest element which is smaller than the current element, say q. This can be done using Binary Search. Finally,find the minimum of the difference of current element from p and q and update the variable.

Below is implementation of this approach:

C/C++

```
 // C++ program to find the minimum absolute difference
// between any of the adjacent elements of an array
// which is created by picking one element from each
// row of the matrix.
#include<bits/stdc++.h>
using namespace std;
#define R 2
#define C 2

// Return smallest element greater than or equal
// to the current element.
int bsearch(int low, int high, int n, int arr[])
{
    int mid = (low + high)/2;

    if(low <= high)
    {
        if(arr[mid] < n)
            return bsearch(mid +1, high, n, arr);
        return bsearch(low, mid - 1, n, arr);
    }

    return low;
}

// Return the minimum absolute difference adjacent
// elements of array
int mindiff(int arr[R][C], int n, int m)
{
    // Sort each row of the matrix.
    for (int i = 0; i < n; i++)
        sort(arr[i], arr[i] + m);

    int ans = INT_MAX;

    // For each matrix element
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < m; j++)
        {
            // Search smallest element in the next row which
            // is greater than or equal to the current element
            int p = bsearch(0, m-1, arr[i][j], arr[i + 1]);
            ans = min(ans, abs(arr[i + 1][p] - arr[i][j]));

            // largest element which is smaller than the current
            // element in the next row must be just before
            // smallest element which is greater than or equal
            // to the current element because rows are sorted.
            if (p-1 >= 0)
                ans = min(ans, abs(arr[i + 1][p - 1] - arr[i][j]));
        }
    }
    return ans;
}

// Driven Program
int main()
{
    int m[R][C] =
    {
        8, 5,
        6, 8,
    };

    cout << mindiff(m, R, C) << endl;
    return 0;
}
```

Java

```
 // Java program to find the minimum
// absolute difference between any
// of the adjacent elements of an
// array which is created by picking
// one element from each row of the matrix
import java.util.Arrays;
class GFG
{
static final int R=2;
static final int C=2;

// Return smallest element greater than
// or equal to the current element.
static int bsearch(int low, int high, int n, int arr[])
```

```
{
    int mid = (low + high)/2;

    if(low <= high)
    {
        if(arr[mid] < n)
            return bsearch(mid +1, high, n, arr);
        return bsearch(low, mid - 1, n, arr);
    }

    return low;
}

// Return the minimum absolute difference adjacent
// elements of array
static int mindiff(int arr[][], int n, int m)
{

    // Sort each row of the matrix.
    for (int i = 0; i < n; i++)
        Arrays.sort(arr[i]);

    int ans = +2147483647;

    // For each matrix element
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < m; j++)
        {

        // Search smallest element in the
        // next row which is greater than
        // or equal to the current element
        int p = bsearch(0, m-1, arr[i][j], arr[i + 1]);
        ans = Math.min(ans, Math.abs(arr[i + 1][p] - arr[i][j]));

        // largest element which is smaller than the current
        // element in the next row must be just before
        // smallest element which is greater than or equal
        // to the current element because rows are sorted.
        if (p-1 >= 0)
            ans = Math.min(ans,
                        Math.abs(arr[i + 1][p - 1] - arr[i][j]));
        }
    }
    return ans;
}
```

```
    //Driver code
public static void main (String[] args)
{
    int m[][] ={{8, 5},
                {6, 8}};

    System.out.println(mindiff(m, R, C));
}
}
//This code is contributed by Anant Agarwal.
```

Python

```
 # Python program to find the minimum absolute difference
# between any of the adjacent elements of an array
# which is created by picking one element from each
# row of the matrix.
# R 2
# C 2

# Return smallest element greater than or equal
# to the current element.
def bsearch(low, high, n, arr):
    mid = (low + high)/2

    if(low <= high):
        if(arr[mid] < n):
            return bsearch(mid +1, high, n, arr);
        return bsearch(low, mid - 1, n, arr);

    return low;

# Return the minimum absolute difference adjacent
# elements of array
def mindiff(arr, n, m):

    # arr = [0 for i in range(R)][for j in range(C)]
    # Sort each row of the matrix.
    for i in range(n):
        sorted(arr)

    ans = 2147483647

    # For each matrix element
    for i in range(n-1):
        for j in range(m):
            # Search smallest element in the next row which
            # is greater than or equal to the current element
            p = bsearch(0, m-1, arr[i][j], arr[i + 1])
            ans = min(ans, abs(arr[i + 1][p] - arr[i][j]))

            # largest element which is smaller than the current
            # element in the next row must be just before
            # smallest element which is greater than or equal
            # to the current element because rows are sorted.
            if (p-1 >= 0):
                ans = min(ans, abs(arr[i + 1][p - 1] - arr[i][j]))
    return ans;

# Driver Program
m =[8, 5], [6, 8]
print mindiff(m, 2, 2)

# Contributed by: Afzal
```

Output:

```
0
```

Time Complexity : O(N*M*logM).

Source

https://www.geeksforgeeks.org/minimum-difference-adjacent-elements-array-contain-elements-row-matrix/

Chapter 57

Modular Exponentiation (Power in Modular Arithmetic)

Modular Exponentiation (Power in Modular Arithmetic) - GeeksforGeeks

Given three numbers x, y and p, compute (x^y) % p.

Examples :

```
Input:  x = 2, y = 3, p = 5
Output: 3
Explanation: 2^3 % 5 = 8 % 5 = 3.

Input:  x = 2, y = 5, p = 13
Output: 6
Explanation: 2^5 % 13 = 32 % 13 = 6.
```

We have discussed recursive and iterative solutions for power.

Below is discussed iterative solution.

```
 /* Iterative Function to calculate (x^y) in O(log y) */
int power(int x, unsigned int y)
{
    int res = 1;     // Initialize result

    while (y > 0)
    {
        // If y is odd, multiply x with result
        if (y & 1)
            res = res*x;
        // n must be even now
        y = y>>1; // y = y/2
        x = x*x;  // Change x to x^2
    }
    return res;
}
```

The problem with above solutions is, overflow may occur for large value of n or x. Therefore, power is generally evaluated under modulo of a large number.

Below is the fundamental modular property that is used for efficiently computing power under modular arithmetic.

```
(ab) mod p = ( (a mod p) (b mod p) ) mod p

For example a = 50,  b = 100, p = 13
50  mod 13  = 11
100 mod 13  = 9

(50 * 100) mod 13 = ( (50 mod 13) * (100 mod 13) ) mod 13
or (5000) mod 13 = ( 11 * 9 ) mod 13
or 8 = 8
```

Below is the implementation based on above property.

C

```
 // Iterative C program to compute modular power
#include <stdio.h>

/* Iterative Function to calculate (x^y)%p in O(log y) */
int power(int x, unsigned int y, int p)
{
    int res = 1;      // Initialize result

    x = x % p;  // Update x if it is more than or
                // equal to p

    while (y > 0)
    {
        // If y is odd, multiply x with result
        if (y & 1)
            res = (res*x) % p;
        // y must be even now
        y = y>>1; // y = y/2
        x = (x*x) % p;
    }
    return res;
}

// Driver program to test above functions
int main()
{
   int x = 2;
   int y = 5;
   int p = 13;
   printf("Power is %u", power(x, y, p));
   return 0;
}
```

Java

```
 // Iterative Java program to
// compute modular power
import java.io.*;

class GFG {

    /* Iterative Function to calculate
       (x^y)%p in O(log y) */
    static int power(int x, int y, int p)
    {
        // Initialize result
        int res = 1;

        // Update x if it is more
        // than or equal to p
        x = x % p;

        while (y > 0)
        {
            // If y is odd, multiply x
            // with result
            if((y & 1)==1)
                res = (res * x) % p;

            // y must be even now
            // y = y / 2
            y = y >> 1;
            x = (x * x) % p;
        }
        return res;
    }

    // Driver Program to test above functions
    public static void main(String args[])
    {
        int x = 2;
        int y = 5;
        int p = 13;
        System.out.println("Power is " + power(x, y, p));
    }
}

// This code is contributed by Nikita Tiwari.
```

Python3

```
 # Iterative Python3 program
# to compute modular power

# Iterative Function to calculate
# (x^y)%p in O(log y)
def power(x, y, p) :
    res = 1     # Initialize result

    # Update x if it is more
    # than or equal to p
    x = x % p

    while (y > 0) :

        # If y is odd, multiply
        # x with result
        if ((y & 1) == 1) :
            res = (res * x) % p

        # y must be even now
        y = y >> 1      # y = y/2
        x = (x * x) % p

    return res


# Driver Code

x = 2; y = 5; p = 13
print("Power is ", power(x, y, p))
```

```
# This code is contributed by Nikita Tiwari.
```

C#

```
 // Iterative C# program to
// compute modular power
class GFG
{

/* Iterative Function to calculate
(x^y)%p in O(log y) */
static int power(int x, int y, int p)
{
    // Initialize result
    int res = 1;

    // Update x if it is more
    // than or equal to p
    x = x % p;

    while (y > 0)
    {
        // If y is odd, multiply
        // x with result
        if((y & 1) == 1)
            res = (res * x) % p;

        // y must be even now
        // y = y / 2
        y = y >> 1;
        x = (x * x) % p;
    }
    return res;
}

// Driver Code
public static void Main()
{
    int x = 2;
    int y = 5;
    int p = 13;
    System.Console.WriteLine("Power is " +
                              power(x, y, p));
}
}

// This code is contributed by mits
```

PHP

```
 <?php
// Iterative PHP program to
// compute modular power

// Iterative Function to
// calculate (x^y)%p in O(log y)
function power($x, $y, $p)
{
    // Initialize result
    $res = 1;

    // Update x if it is more
    // than or equal to p
    $x = $x % $p;

    while ($y > 0)
    {
        // If y is odd, multiply
        // x with result
        if ($y & 1)
            $res = ($res * $x) % $p;

        // y must be even now

        // y = $y/2
        $y = $y >> 1;
        $x = ($x * $x) % $p;
    }
    return $res;
}

// Driver Code
$x = 2;
$y = 5;
$p = 13;
echo "Power is ", power($x, $y, $p);

// This code is contributed by aj_36
?>
```

Output :

```
 Power is 6
```

Time Complexity of above solution is O(Log y).

Modular exponentiation (Recursive)

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

Improved By : jit_t, rd10, Mithun Kumar

Source

https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/

Chapter 58

Modular exponentiation (Recursive)

Modular exponentiation (Recursive) - GeeksforGeeks

Given three numbers a, b and c, we need to find (a^b) % c

Now why do "% c" after exponentiation, because a^b will be really large even for relatively small values of a, b and that is a problem because the data type of the language that we try to code the problem, will most probably not let us store such a large number.

Examples:

```
Input : a = 2312 b = 3434 c = 6789
Output : 6343

Input : a = -3 b = 5 c = 89
Output : 24
```

The idea is based on below properties.

Property 1:
(m * n) % p has a very interesting property:
(m * n) % p =((m % p) * (n % p)) % p

Property 2:
if b is even:
(a ^ b) % c = ((a ^ b/2) * (a ^ b/2))%c ? this suggests divide and conquer
if b is odd:
(a ^ b) % c = (a * (a ^(b-1))%c

Property 3:
If we have to return the mod of a negative number x whose absolute value is less than y:
then (x + y) % y will do the trick

Note:
Also as the product of (a ^ b/2) * (a ^ b/2) and a * (a ^(b-1) may cause overflow, hence we must be careful about those scenarios

C

```
 // Recursive C program to compute modular power
#include <stdio.h>

int exponentMod(int A, int B, int C)
{
    // Base cases
    if (A == 0)
        return 0;
    if (B == 0)
        return 1;

    // If B is even
    long y;
    if (B % 2 == 0) {
        y = exponentMod(A, B / 2, C);
        y = (y * y) % C;
    }

    // If B is odd
    else {
        y = A % C;
        y = (y * exponentMod(A, B - 1, C) % C) % C;
    }

    return (int)((y + C) % C);
}

// Driver program to test above functions
int main()
{
   int A = 2, B = 5, C = 13;
   printf("Power is %d", exponentMod(A, B, C));
   return 0;
}
```

Java

```
 // Recursive Java program
// to compute modular power
import java.io.*;
```

```
class GFG
{
static int exponentMod(int A,
                       int B, int C)
{

    // Base cases
    if (A == 0)
        return 0;
    if (B == 0)
        return 1;

    // If B is even
    long y;
    if (B % 2 == 0)
    {
        y = exponentMod(A, B / 2, C);
        y = (y * y) % C;
    }

    // If B is odd
    else
    {
        y = A % C;
        y = (y * exponentMod(A, B - 1,
                             C) % C) % C;
    }

    return (int)((y + C) % C);
}

// Driver Code
public static void main(String args[])
{
    int A = 2, B = 5, C = 13;
    System.out.println("Power is " +
                        exponentMod(A, B, C));
}
}

// This code is contributed
// by Swetank Modi.
```

Python3

```
 # Recursive Python program
# to compute modular power
def exponentMod(A, B, C):
    # Base Cases
    if (A == 0):
        return 0
    if (B == 0):
        return 1

    # If B is Even
    y = 0
    if (B % 2 == 0):
        y = exponentMod(A, B / 2, C)
        y = (y * y) % C

    # If B is Odd
    else:
        y = A % C
        y = (y * exponentMod(A, B - 1,
                             C) % C) % C
    return ((y + C) % C)

# Driver Code
A = 2
B = 5
C = 13
print("Power is", exponentMod(A, B, C))

# This code is contributed
# by Swetank Modi.
```

C#

```
 // Recursive C# program
// to compute modular power
class GFG
{
static int exponentMod(int A, int B, int C)
{

    // Base cases
    if (A == 0)
        return 0;
    if (B == 0)
        return 1;

    // If B is even
    long y;
    if (B % 2 == 0)
    {
        y = exponentMod(A, B / 2, C);
        y = (y * y) % C;
    }

    // If B is odd
    else
    {
        y = A % C;
        y = (y * exponentMod(A, B - 1,
                             C) % C) % C;
    }

    return (int)((y + C) % C);
}

// Driver Code
public static void Main()
{
    int A = 2, B = 5, C = 13;
    System.Console.WriteLine("Power is " +
                    exponentMod(A, B, C));
}
}

// This code is contributed
// by mits
```

PHP

```
 <?php
// Recursive PHP program to
// compute modular power
function exponentMod($A, $B, $C)
{
    // Base cases
    if ($A == 0)
        return 0;
    if ($B == 0)
        return 1;

    // If B is even
    if ($B % 2 == 0)
    {
        $y = exponentMod($A, $B / 2, $C);
        $y = ($y * $y) % $C;
    }

    // If B is odd
    else
    {
        $y = $A % $C;
        $y = ($y * exponentMod($A, $B - 1,
                               $C) % $C) % $C;
    }

    return (($y + $C) % $C);
}


// Driver Code
$A = 2;
$B = 5;
$C = 13;
echo "Power is " . exponentMod($A, $B, $C);

// This code is contributed
// by Swetank Modi.
?>
```

Output:

```
Power is 6
```

Iterative modular exponentiation.

Improved By : swetankmodi, Mithun Kumar

Source

https://www.geeksforgeeks.org/modular-exponentiation-recursive/

Chapter 59

Multiply two polynomials

Multiply two polynomials - GeeksforGeeks

Given two polynomials represented by two arrays, write a function that multiplies given two polynomials.

Example:

```
Input:  A[] = {5, 0, 10, 6}
        B[] = {1, 2, 4}
Output: prod[] = {5, 10, 30, 26, 52, 24}

The first input array represents "5 + 0x^1 + 10x^2 + 6x^3"
The second array represents "1 + 2x^1 + 4x^2"
And Output is "5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5"
```

A simple solution is to one by one consider every term of first polynomial and multiply it with every term of second polynomial. Following is algorithm of this simple method.

```
multiply(A[0..m-1], B[0..n01])
1) Create a product array prod[] of size m+n-1.
2) Initialize all entries in prod[] as 0.
3) Travers array A[] and do following for every element A[i]
...(3.a) Traverse array B[] and do following for every element B[j]
          prod[i+j] = prod[i+j] + A[i] * B[j]
4) Return prod[].
```

The following is C++ implementation of above algorithm.

```
 // Simple C++ program to multiply two polynomials
```

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

// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *multiply(int A[], int B[], int m, int n)
{
   int *prod = new int[m+n-1];

   // Initialize the porduct polynomial
   for (int i = 0; i<m+n-1; i++)
     prod[i] = 0;

   // Multiply two polynomials term by term

   // Take ever term of first polynomial
   for (int i=0; i<m; i++)
   {
     // Multiply the current term of first polynomial
     // with every term of second polynomial.
     for (int j=0; j<n; j++)
         prod[i+j] += A[i]*B[j];
   }

   return prod;
}

// A utility function to print a polynomial
void printPoly(int poly[], int n)
{
    for (int i=0; i<n; i++)
    {
       cout << poly[i];
       if (i != 0)
        cout << "x^" << i ;
       if (i != n-1)
       cout << " + ";
    }
}

// Driver program to test above functions
int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};

    // The following array represents polynomial 1 + 2x + 4x^2
    int B[] = {1, 2, 4};
    int m = sizeof(A)/sizeof(A[0]);
    int n = sizeof(B)/sizeof(B[0]);

    cout << "First polynomial is n";
    printPoly(A, m);
    cout << "nSecond polynomial is n";
    printPoly(B, n);

    int *prod = multiply(A, B, m, n);

    cout << "nProduct polynomial is n";
    printPoly(prod, m+n-1);

    return 0;
}
```

Output

```
First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Product polynomial is
5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5
```

Time complexity of the above solution is O(mn). If size of two polynomials same, then time complexity is O(n^2).

Can we do better?
There are methods to do multiplication faster than O(n^2) time. These methods are mainly based on divide and conquer. Following is one simple method that divides the given polynomial (of degree n) into two polynomials one containing lower degree terms(lower than n/2) and other containing higher degree terns (higher than or equal to n/2)

```
Let the two given polynomials be A and B.
For simplicity, Let us assume that the given two polynomials are of
same degree and have degree in powers of 2, i.e., n = 2i

The polynomial 'A' can be written as A0 + A1*xn/2
The polynomial 'B' can be written as B0 + B1*xn/2

For example 1 + 10x + 6x2 - 4x3 + 5x4 can be
written as (1 + 10x) + (6 - 4x + 5x2)*x2

A * B  = (A0 + A1*xn/2) * (B0 + B1*xn/2)
```

```
       = A0*B0 + A0*B1*xn/2 + A1*B0*xn/2 + A1*B1*xn
       = A0*B0 + (A0*B1 + A1*B0)xn/2 + A1*B1*xn
```

So the above divide and conquer approach requires 4 multiplications and O(n) time to add all 4 results. Therefore the time complexity is T(n) = 4T(n/2) + O(n). The solution of the recurrence is O(n^2) which is same as the above simple solution.

The idea is to reduce number of multiplications to 3 and make the recurrence as T(n) = 3T(n/2) + O(n)

How to reduce number of multiplications?
This requires a little trick similar toStrassen's Matrix Multiplication. We do following 3 multiplications.

```
X = (A0 + A1)*(B0 + B1) // First Multiplication
Y = A0B0  // Second
Z = A1B1  // Third

The missing middle term in above multiplication equation A0*B0 + (A0*B1 +
A1*B0)xn/2 + A1*B1*xn can obtained using below.
A0B1 + A1B0 = X - Y - Z
```

In Depth Explanation
Conventional polynomial multiplication uses 4 coefficient multiplications:

```
(ax + b)(cx + d) = acx2 + (ad + bc)x + bd
```

However, notice the following relation:

```
(a + b)(c + d) = ad + bc + ac + bd
```

The rest of the two components are exactly the middle coefficient for product of two polynomials. Therefore, the product can be computed as:

```
(ax + b)(cx + d) = acx2 +
((a + b)(c + d) , ac , bd )x + bd
```

Hence, the latter expression has only three multiplications.

So the time taken by this algorithm is T(n) = 3T(n/2) + O(n)
The solution of above recurrence is O(n^{Lg3}) which is better than O(n^2).

We will soon be discussing implementation of above approach.

There is a O(nLogn) algorithm also that uses Fast Fourier Transform to multiply two polynomials (Refer this and thisfor details)

Sources:
http://www.cse.ust.hk/~dekai/271/notes/L03/L03.pdf

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.

Improved By : Mr.L

Source

https://www.geeksforgeeks.org/multiply-two-polynomials-2/

Chapter 60

Number of days after which tank will become empty

Number of days after which tank will become empty - GeeksforGeeks

Given a tank with capacity C liters which is completely filled in starting. Everyday tank is filled with l liters of water and in the case of overflow extra water is thrown out. Now on i-th day i liters of water is taken out for drinking. We need to find out the day at which tank will become empty the first time.

Examples:

```
Input : Capacity = 5
        l = 2
Output : 4
At the start of 1st day, water in tank = 5
and at the end of the 1st day = (5 - 1) = 4
At the start of 2nd day, water in tank = 4 + 2 = 6
but tank capacity is 5 so water = 5
and at the end of the 2nd day = (5 - 2) = 3
At the start of 3rd day, water in tank = 3 + 2 = 5
and at the end of the 3rd day = (5 - 3) = 2
At the start of 4th day, water in tank = 2 + 2 = 4
and at the end of the 4th day = (4 - 4) = 0
    So final answer will be 4
```

We can see that tank will be full for starting (l + 1) days because water taken out is less than water being filled. After that, each day water in the tank will be decreased by 1 more liter and on (l + 1 + i)th day (C – (i)(i + 1) / 2) liter water will remain before taking drinking water.
Now we need to find a minimal day (l + 1 + K), in which even after filling the tank by l liters we have water less than l in tank i.e. on (l + 1 + K – 1)th day tank becomes empty

so our goal is to find minimum K such that,
C – K(K + 1) / 2 <= l

We can solve above equation using binary search and then (l + K) will be our answer. Total time complexity of solution will be O(log C)

C++

```
 // C/C++ code to find number of days after which
// tank will become empty
#include <bits/stdc++.h>
using namespace std;

// Utility method to get sum of first n numbers
int getCumulateSum(int n)
{
    return (n * (n + 1)) / 2;
}

// Method returns minimum number of days after
// which tank will become empty
int minDaysToEmpty(int C, int l)
{
    // if water filling is more than capacity then
    // after C days only tank will become empty
    if (C <= l)
        return C;

    // initialize binary search variable
    int lo = 0;
    int hi = 1e4;
    int mid;

    // loop until low is less than high
    while (lo < hi) {
        mid = (lo + hi) / 2;

        // if cumulate sum is greater than (C - l)
        // then search on left side
        if (getCumulateSum(mid) >= (C - l))
            hi = mid;

        // if (C - l) is more then search on
        // right side
        else
            lo = mid + 1;
    }

    // final answer will be obtained by adding
    // 1 to binary search result
    return (1 + lo);
}

// Driver code to test above methods
int main()
{
    int C = 5;
    int l = 2;

    cout << minDaysToEmpty(C, l) << endl;
    return 0;
}
```

Java

```
 // Java code to find number of days after which
// tank will become empty
public class Tank_Empty {

    // Utility method to get sum of first n numbers
    static int getCumulateSum(int n)
    {
        return (n * (n + 1)) / 2;
    }

    // Method returns minimum number of days after
    // which tank will become empty
    static int minDaysToEmpty(int C, int l)
    {
        // if water filling is more than capacity then
        // after C days only tank will become empty
        if (C <= l)
            return C;

        // initialize binary search variable
        int lo = 0;
        int hi = (int)1e4;
        int mid;

        // loop until low is less than high
        while (lo < hi) {

            mid = (lo + hi) / 2;

            // if cumulate sum is greater than (C - l)
            // then search on left side
            if (getCumulateSum(mid) >= (C - l))
```

```
                hi = mid;

            // if (C - l) is more then search on
            // right side
            else
                lo = mid + 1;
        }

        // final answer will be obtained by adding
        // l to binary search result
        return (l + lo);
    }

    // Driver code to test above methods
    public static void main(String args[])
    {
        int C = 5;
        int l = 2;

        System.out.println(minDaysToEmpty(C, l));
    }
}
// This code is contributed by Sumit Ghosh
```

Python3

```
 # Python3 code to find number of days
# after which tank will become empty

# Utility method to get
# sum of first n numbers
def getCumulateSum(n):

    return int((n * (n + 1)) / 2)


# Method returns minimum number of days
# after  which tank will become empty
def minDaysToEmpty(C, l):

    # if water filling is more than
    # capacity then after C days only
    # tank will become empty
    if (C <= l) : return C

    # initialize binary search variable
    lo, hi = 0, 1e4
    # loop until low is less than high
    while (lo < hi):
        mid = int((lo + hi) / 2)

        # if cumulate sum is greater than (C - l)
        # then search on left side
        if (getCumulateSum(mid) >= (C - l)):
            hi = mid

        # if (C - l) is more then
        # search on right side
        else:
            lo = mid + 1

    # Final answer will be obtained by
    # adding l to binary search result
    return (l + lo)

# Driver code
C, l = 5, 2
print(minDaysToEmpty(C, l))

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 // C# code to find number
// of days after which
// tank will become empty
using System;

class GFG
{

    // Utility method to get
    // sum of first n numbers
    static int getCumulateSum(int n)
    {
        return (n * (n + 1)) / 2;
    }

    // Method returns minimum
    // number of days after
    // which tank will become empty
    static int minDaysToEmpty(int C,
                              int l)
    {
        // if water filling is more
        // than capacity then after
        // C days only tank will
        // become empty
        if (C <= l)
            return C;

        // initialize binary
        // search variable
        int lo = 0;
        int hi = (int)1e4;
        int mid;

        // loop until low is
        // less than high
        while (lo < hi)
        {

            mid = (lo + hi) / 2;

            // if cumulate sum is
            // greater than (C - l)
            // then search on left side
            if (getCumulateSum(mid) >= (C - l))
                hi = mid;

            // if (C - l) is more then
            // search on right side
            else
                lo = mid + 1;
        }

        // final answer will be
        // obtained by adding
        // l to binary search result
        return (l + lo);
    }

    // Driver code
    static public void Main ()
    {
        int C = 5;
        int l = 2;

        Console.WriteLine(minDaysToEmpty(C, l));
    }
}

// This code is contributed by ajit
```

Output:

```
4
```

Alternate Solution :
It can be solved mathematically with a simple formula:

Let's Assume C>L. Let d be the amount of days after the L*th* when the tank become empty.During that time, there will be **(d-1)**refills and **d** withdrawals.
Hence we need to solve this equation :

[illegible]

Sum of all withdrawals is a sum of arithmetic progression,therefore :

[illegible]

[illegible]

[illegible]

Discriminant = 1+8(C-L)>0,because C>L.
Skipping the negative root, we get the following formula:

[illegible]

Therefore, the final alwer is:

[illegible]

C++

```
 // C/C++ code to find number of days after which
// tank will become empty
#include <bits/stdc++.h>
using namespace std;

// Method returns minimum number of days after
// which tank will become empty
int minDaysToEmpty(int C, int l)
{
    if (l >= C)
        return C;

    double eq_root = (std::sqrt(1+8*(C-l)) - 1) / 2;
    return std::ceil(eq_root) + 1;
}

// Driver code to test above methods
int main()
{
    cout << minDaysToEmpty(5, 2) << endl;
    cout << minDaysToEmpty(6514683, 4965) << endl;
    return 0;
}
```

Java

```
 // Java code to find number of days
// after which tank will become empty
import java.lang.*;
class GFG {

// Method returns minimum number of days
// after which tank will become empty
static int minDaysToEmpty(int C, int l)
{
    if (l >= C) return C;

    double eq_root = (Math.sqrt(1 + 8 *
                     (C - l)) - 1) / 2;
    return (int)(Math.ceil(eq_root) + l);
}

// Driver code
public static void main(String[] args)
{
    System.out.println(minDaysToEmpty(5, 2));
    System.out.println(minDaysToEmpty(6514683, 4965));
}
}

// This code is contributed by Smitha Dinesh Semwal.
```

Python3

```
 # Python3 code to find number of days
# after which tank will become empty
import math

# Method returns minimum number of days
# after which tank will become empty
```

```
def minDaysToEmpty(C, l):

    if (l >= C): return C

    eq_root = (math.sqrt(1 + 8 * (C - l)) - 1) / 2
    return math.ceil(eq_root) + l

# Driver code
print(minDaysToEmpty(5, 2))
print(minDaysToEmpty(6514683, 4965))

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 // C# code to find number
// of days after which
// tank will become empty
using System;

class GFG
{

// Method returns minimum
// number of days after
// which tank will become empty
static int minDaysToEmpty(int C,
                          int l)
{
    if (l >= C) return C;

    double eq_root = (Math.Sqrt(1 + 8 *
                     (C - l)) - 1) / 2;
    return (int)(Math.Ceiling(eq_root) + l);
}

// Driver code
static public void Main ()
{
    Console.WriteLine(minDaysToEmpty(5, 2));
    Console.WriteLine(minDaysToEmpty(6514683,
                                     4965));
}
}

// This code is contributed by ajit
```

PHP

```
 <?php
// PHP code to find number
// of days after which
// tank will become empty

// Method returns minimum
// number of days after
// which tank will become empty
function minDaysToEmpty($C, $l)
{
    if ($l >= $C)
        return $C;

    $eq_root = (int)sqrt(1 + 8 *
                   ($C - $l) - 1) / 2;
    return ceil($eq_root) + $l;
}

// Driver code
echo minDaysToEmpty(5, 2), "\n";
echo minDaysToEmpty(6514683,
                    4965), "\n";

// This code is contributed
// by akt_mit
?>
```

Output :

```
4
8573
```

Thanks to **Andrey Khayrutdinov** for suggesting this solution.

Improved By : jit_t

Source

https://www.geeksforgeeks.org/number-days-tank-will-become-empty/

Chapter 61

Number of ways to divide a given number as a set of integers in decreasing order

Number of ways to divide a given number as a set of integers in decreasing order - GeeksforGeeks

Given two numbers a and m. The task is to find the number of ways in which **a** can be represented by a set [illegible] such that [illegible] and the summation of these numbers is equal to **a**. Also [illegible] (maximum size of the set cannot exceed **m**).

Examples:

> **Input** : a = 4, m = 4
> **Output** : 2 –> ({4}, {3, 1})
> **Note**: {2, 2} is not a valid set as values are not in decreasing order
>
> **Input** : a = 7, m = 5
> **Output** : 5 –> ({7}, {6, 1}, {5, 2}, {4, 3}, {4, 2, 1})

Approach: This problem can be solved by Divide and Conquer using a recursive approach which follows the following conditions:

- If **a** is equal to zero, one solution has been found.
- If a > 0 and m == 0, this set violates the condition as no further values can be added in the set.
- If calculation has already been done for given values of **a**, **m** and prev (last value included in the current set), return that value.
- Start a loop from **i** = **a** till 0 and if **i** < **prev**, count the number of solutions if we include **i** in the current set and return it.

Below is the implementation of the above approach:

```
 # Python3 code to calculate the number of ways
# in which a given number can be represented
# as set of finite numbers

# Import function to initialize the dictionary
from collections import defaultdict

# Initialize dictionary which is used
# to check if given solution is already
# visited or not to avoid
# calculating it again
visited = defaultdict(lambda : False)

# Initialize dictionary which is used to
# store the number of ways in which solution
# can be obtained for given values
numWays = defaultdict(lambda : 0)

# This function returns the total number
# of sets which satisfy given criteria
# a --> number to be divided into sets
# m --> maximum possible size of the set
# x --> previously selected value
def countNumOfWays(a, m, prev):

    # number is divided properly and
    # hence solution is obtained
    if a == 0:
        return 1

    # Solution can't be obtained
    elif a > 0 and m == 0:
        return 0

    # Return the solution if it has
    # already been calculated
    elif visited[(a, m, prev)] == True:
        return numWays[(a, m, prev)]

    else:
        visited[(a, m, prev)] = True

        for i in range(a, -1, -1):
            # Continue only if current value is
            # smaller compared to previous value
            if i < prev:
                numWays[(a,m,prev)] += countNumOfWays(a-i,m-1,i)

        return numWays[(a, m, prev)]

# Values of 'a' and 'm' for which
# solution is to be found
# MAX_CONST is extremely large value
# used for first comparison in the function
a, m, MAX_CONST = 7, 5, 10**5
print(countNumOfWays(a, m, MAX_CONST))
```

Output:

```
5
```

Time Complexity: O(a*log(a))

Source

https://www.geeksforgeeks.org/number-of-ways-to-divide-a-given-number-as-a-set-of-integers-in-decreasing-order/

Chapter 62

Numbers whose factorials end with n zeros

Numbers whose factorials end with n zeros - GeeksforGeeks

Given an integer n, we need to find the number of positive integers whose factorial ends with n zeros.

Examples:

```
Input : n = 1
Output : 5 6 7 8 9
Explanation: Here, 5! = 120, 6! = 720,
7! = 5040, 8! = 40320 and 9! = 362880.

Input : n = 2
Output : 10 11 12 13 14
```

Prerequisite : Trailing zeros in factorial.

Naive approach: We can just iterate through the range of integers and find the number of trailing zeros of all the numbers and print the numbers with n trailing zeros.

Efficient Approach: In this approach we use binary search. Use binary search for all the numbers in the range and get the first number with n trailing zeros. Find all the numbers with m trailing zeros after that number.

CPP

```
 // Binary search based CPP program to find
// numbers with n trailing zeros.
#include <bits/stdc++.h>
```

```
using namespace std;

// Function to calculate trailing zeros
int trailingZeroes(int n)
{
    int cnt = 0;
    while (n > 0) {
        n /= 5;
        cnt += n;
    }
    return cnt;
}

void binarySearch(int n)
{
    int low = 0;
    int high = 1e6; // range of numbers

    // binary search for first number with
    // n trailing zeros
    while (low < high) {
        int mid = (low + high) / 2;
        int count = trailingZeroes(mid);
        if (count < n)
            low = mid + 1;
        else
            high = mid;
    }

    // Print all numbers after low with n
    // trailing zeros.
    vector<int> result;
    while (trailingZeroes(low) == n) {
        result.push_back(low);
        low++;
    }

    // Print result
    for (int i = 0; i < result.size(); i++)
        cout << result[i] << " ";
}

// Driver code
int main()
{
    int n = 2;
    binarySearch(n);
    return 0;
}
```

Java

```
 // Binary search based Java
// program to find numbers
// with n trailing zeros.
import java.io.*;

class GFG {

    // Function to calculate
    // trailing zeros
    static int trailingZeroes(int n)
    {
        int cnt = 0;
        while (n > 0)
        {
            n /= 5;
            cnt += n;
        }
        return cnt;
    }

    static void binarySearch(int n)
    {
        int low = 0;

        // range of numbers
        int high = 1000000;

        // binary search for first number
        // with n trailing zeros
        while (low < high) {
            int mid = (low + high) / 2;
            int count = trailingZeroes(mid);
            if (count < n)
                low = mid + 1;
            else
                high = mid;
        }

        // Print all numbers after low
        // with n trailing zeros.
        int result[] = new int[1000];
        int k = 0;
        while (trailingZeroes(low) == n) {
            result[k] = low;
            k++;
            low++;
        }

        // Print result
        for (int i = 0; i < k; i++)
            System.out.print(result[i] + " ");
    }

    // Driver code
    public static void main(String args[])
    {
        int n = 3;
        binarySearch(n);
    }
}

// This code is contributed
// by Nikita Tiwari.
```

Python3

```
 # Binary search based Python3 code to find
# numbers with n trailing zeros.

# Function to calculate trailing zeros
def trailingZeroes( n ):
    cnt = 0
    while n > 0:
        n =int(n/5)
        cnt += n
    return cnt

def binarySearch( n ):
    low = 0
    high = 1e6  # range of numbers

    # binary search for first number with
    # n trailing zeros
    while low < high:
        mid = int((low + high) / 2)
        count = trailingZeroes(mid)
        if count < n:
            low = mid + 1
        else:
            high = mid

    # Print all numbers after low with n
    # trailing zeros.
    result = list()
    while trailingZeroes(low) == n:
        result.append(low)
        low+=1

    # Print result
    for i in range(len(result)):
        print(result[i],end=" ")

# Driver code
n = 2
binarySearch(n)

# This code is contributed by "Sharad_Bhardwaj".
```

C#

```
 // Binary search based C#
// program to find numbers
// with n trailing zeros.
using System;

class GFG {

    // Function to calculate
    // trailing zeros
    static int trailingZeroes(int n)
    {
        int cnt = 0;

        while (n > 0)
        {
            n /= 5;
            cnt += n;
        }

        return cnt;
    }

    static void binarySearch(int n)
    {
        int low = 0;

        // range of numbers
        int high = 1000000;

        // binary search for first number
        // with n trailing zeros
        while (low < high) {
            int mid = (low + high) / 2;
            int count = trailingZeroes(mid);

            if (count < n)
                low = mid + 1;
            else
                high = mid;
        }

        // Print all numbers after low
        // with n trailing zeros.
        int []result = new int[1000];
        int k = 0;
        while (trailingZeroes(low) == n) {
            result[k] = low;
            k++;
            low++;
        }

        // Print result
        for (int i = 0; i < k; i++)
            Console.Write(result[i] + " ");
    }

    // Driver code
    public static void Main()
    {
        int n = 2;

        binarySearch(n);
    }
}

// This code is contributed by vt_m.
```

Output:

```
10 11 12 13 14
```

Source

https://www.geeksforgeeks.org/numbers-whose-factorials-end-with-n-zeros/

Chapter 63

Place k elements such that minimum distance is maximized

Place k elements such that minimum distance is maximized - GeeksforGeeks

Given an array representing n positions along a straight line. Find k (where k <= n) elements from the array such that the maximum distance between any two (consecutive points among the k points) is maximized.

Examples :

```
Input : arr[] = {1, 2, 8, 4, 9}
            k = 3
Output : 3
Largest minimum distance = 3
3 elements arranged at positions 1, 4 and 8,
Resulting in a minimum distance of 3

Input  : arr[] = {1, 2, 7, 5, 11, 12}
             k = 3
Output : 5
Largest minimum distance = 5
3 elements arranged at positions 1, 7 and 12,
resulting in a minimum distance of 5 (between
7 and 12)
```

A **Naive Solution** is to consider all subsets of size 3 and find minimum distance for every subset. Finally return the largest of all minimum distances.

An **Efficient Solution** is based on Binary Search. We first sort the array. Now we know maximum possible value result is arr[n-1] – arr[0] (for k = 2). We do binary search for maximum result for given k. We start with middle of maximum possible result. If middle is a feasible solution, we search on right half of mid. Else we search is left half. To check feasibility, we place k elements under given mid distance.

C++

```
 // C++ program to find largest minimum distance
// among k points.
#include <bits/stdc++.h>
using namespace std;

// Returns true if it is possible to arrange
// k elements of arr[0..n-1] with minimum distance
// given as mid.
bool isFeasible(int mid, int arr[], int n, int k)
{
    // Place first element at arr[0] position
    int pos = arr[0];

    // Initialize count of elements placed.
    int elements = 1;

    // Try placing k elements with minimum
    // distance mid.
    for (int i=1; i<n; i++)
    {
        if (arr[i] - pos >= mid)
        {
            // Place next element if its
            // distance from the previously
            // placed element is greater
            // than current mid
            pos = arr[i];
            elements++;

            // Return if all elements are placed
            // successfully
            if (elements == k)
              return true;
        }
    }
    return 0;
}

// Returns largest minimum distance for k elements
// in arr[0..n-1]. If elements can't be placed,
// returns -1.
int largestMinDist(int arr[], int n, int k)
{
    // Sort the positions
    sort(arr,arr+n);

    // Initialize result.
    int res = -1;

    // Consider the maximum possible distance
    int left = 0, right = arr[n-1];

    // Do binary search for largest minimum distance
    while (left < right)
    {
        int mid = (left + right)/2;

        // If it is possible to place k elements
        // with minimum distance mid, search for
        // higher distance.
        if (isFeasible(mid, arr, n, k))
        {
            // Change value of variable max to mid iff
            // all elements can be successfully placed
            res = max(res, mid);
            left = mid + 1;
        }

        // If not possible to place k elements, search
        // for lower distance
        else
            right = mid;
    }

    return res;
}

// Driver code
int main()
{
    int arr[] = {1, 2, 8, 4, 9};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    cout << largestMinDist(arr, n, k);
    return 0;
}
```

Java

```
 // Java program to find largest
// minimum distance among k points.
import java.util.Arrays;
```

```
class GFG
{
// Returns true if it is possible to
// arrange k elements of arr[0..n-1]
// with minimum distance given as mid.
static boolean isFeasible(int mid, int arr[],
                                int n, int k)
{
    // Place first element at arr[0] position
    int pos = arr[0];

    // Initialize count of elements placed.
    int elements = 1;

    // Try placing k elements with minimum
    // distance mid.
    for (int i=1; i<n; i++)
    {
        if (arr[i] - pos >= mid)
        {
            // Place next element if its
            // distance from the previously
            // placed element is greater
            // than current mid
            pos = arr[i];
            elements++;

            // Return if all elements are
            // placed successfully
            if (elements == k)
            return true;
        }
    }
    return false;
}

// Returns largest minimum distance for
// k elements in arr[0..n-1]. If elements
// can't be placed, returns -1.
static int largestMinDist(int arr[], int n,
                                    int k)
{
    // Sort the positions
    Arrays.sort(arr);

    // Initialize result.
    int res = -1;
    // Consider the maximum possible distance
    int left = arr[0], right = arr[n-1];

    // Do binary search for largest
    // minimum distance
    while (left < right)
    {
        int mid = (left + right)/2;

        // If it is possible to place k
        // elements with minimum distance mid,
        // search for higher distance.
        if (isFeasible(mid, arr, n, k))
        {
            // Change value of variable max to
            // mid if all elements can be
            // successfully placed
            res = Math.max(res, mid);
            left = mid + 1;
        }

        // If not possible to place k elements,
        // search for lower distance
        else
            right = mid;
    }

    return res;
}

// driver code
public static void main (String[] args)
{
    int arr[] = {1, 2, 8, 4, 9};
    int n = arr.length;
    int k = 3;
    System.out.print(largestMinDist(arr, n, k));
}
}

// This code is contributed by Anant Agarwal.
```

C#

```
 // C# program to find largest
// minimum distance among k points.
using System;
```

```
public class GFG {

    // Returns true if it is possible to
    // arrange k elements of arr[0..n-1]
    // with minimum distance given as mid.
    static bool isFeasible(int mid, int []arr,
                                 int n, int k)
    {

        // Place first element at arr[0]
        // position
        int pos = arr[0];

        // Initialize count of elements placed.
        int elements = 1;

        // Try placing k elements with minimum
        // distance mid.
        for (int i = 1; i < n; i++)
        {
            if (arr[i] - pos >= mid)
            {

                // Place next element if its
                // distance from the previously
                // placed element is greater
                // than current mid
                pos = arr[i];
                elements++;

                // Return if all elements are
                // placed successfully
                if (elements == k)
                    return true;
            }
        }

        return false;
    }

    // Returns largest minimum distance for
    // k elements in arr[0..n-1]. If elements
    // can't be placed, returns -1.
    static int largestMinDist(int []arr, int n,
                                        int k)
    {
        // Sort the positions
        Array.Sort(arr);

        // Initialize result.
        int res = -1;

        // Consider the maximum possible
        // distance
        int left = arr[0], right = arr[n-1];

        // Do binary search for largest
        // minimum distance
        while (left < right)
        {
            int mid = (left + right) / 2;

            // If it is possible to place k
            // elements with minimum distance
            // mid, search for higher distance.
            if (isFeasible(mid, arr, n, k))
            {
                // Change value of variable
                // max to mid if all elements
                // can be successfully placed
                res = Math.Max(res, mid);
                left = mid + 1;
            }

            // If not possible to place k
            // elements, search for lower
            // distance
            else
                right = mid;
        }

        return res;
    }

    // driver code
    public static void Main ()
    {
        int []arr = {1, 2, 8, 4, 9};
        int n = arr.Length;
        int k = 3;

        Console.WriteLine(
            largestMinDist(arr, n, k));
    }
```

```
}

// This code is contributed by Sam007.
```

PHP

```
 <?php
// PHP program to find largest
// minimum distance among k points.

// Returns true if it is possible
// to arrange k elements of
// arr[0..n-1] with minimum
// distance given as mid.
function isFeasible($mid, $arr,
                    $n, $k)
{
    // Place first element
    // at arr[0] position
    $pos = $arr[0];

    // Initialize count of
    // elements placed.
    $elements = 1;

    // Try placing k elements
    // with minimum distance mid.
    for ($i = 1; $i < $n; $i++)
    {
        if ($arr[$i] - $pos >= $mid)
        {
            // Place next element if
            // its distance from the
            // previously placed
            // element is greater
            // than current mid
            $pos = $arr[$i];
            $elements++;

            // Return if all elements
            // are placed successfully
            if ($elements == $k)
            return true;
        }
    }
    return 0;
}
```

```
// Returns largest minimum
// distance for k elements
// in arr[0..n-1]. If elements
// can't be placed, returns -1.
function largestMinDist($arr, $n, $k)
{
    // Sort the positions
    sort($arr);

    // Initialize result.
    $res = -1;

    // Consider the maximum
    // possible distance
    $left = $arr[0];
    $right = $arr[$n - 1];

    // Do binary search for
    // largest minimum distance
    while ($left < $right)
    {
        $mid = ($left + $right) / 2;

        // If it is possible to place
        // k elements with minimum
        // distance mid, search for
        // higher distance.
        if (isFeasible($mid, $arr,
                       $n, $k))
        {
            // Change value of variable
            // max to mid iff all elements
            // can be successfully placed
            $res = max($res, $mid);
            $left = $mid + 1;
        }

        // If not possible to place
        // k elements, search for
        // lower distance
        else
            $right = $mid;
    }

    return $res;
}

// Driver Code
```

```
$arr = array(1, 2, 8, 4, 9);
$n = sizeof($arr);
$k = 3;
echo largestMinDist($arr, $n, $k);

// This code is contributed by aj_36
?>
```

Output :

```
3
```

Improved By : Sam007, jit_t

Source

https://www.geeksforgeeks.org/place-k-elements-such-that-minimum-distance-is-maximized/

Chapter 64

Program for Tower of Hanoi

Program for Tower of Hanoi - GeeksforGeeks

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Approach :

```
Take an example for 2 disks :
Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.
Step 2 : Shift second disk from 'A' to 'C'.
Step 3 : Shift first disk from 'B' to 'C'.

The pattern here is :
Shift 'n-1' disks from 'A' to 'B'.
Shift last disk from 'A' to 'C'.
Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks :
```

Examples:

```
Input : 2
Output : Disk 1 moved from A to B
```

```
         Disk 2 moved from A to C
         Disk 1 moved from B to C

Input : 3
Output : Disk 1 moved from A to C
         Disk 2 moved from A to B
         Disk 1 moved from C to B
         Disk 3 moved from A to C
         Disk 1 moved from B to A
         Disk 2 moved from B to C
         Disk 1 moved from A to C
```

C/C++

```
 #include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    return 0;
}
```

Java

```
 // Java recursive program to solve tower of hanoi puzzle

class GFG
{
    // Java recursive function to solve tower of hanoi puzzle
    static void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
    {
        if (n == 1)
        {
            System.out.println("Move disk 1 from rod " +  from_rod + " to rod " + to_rod);
            return;
        }
        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
        System.out.println("Move disk " + n + " from rod " +  from_rod + " to rod " + to_rod);
        towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
    }

    //  Driver method
    public static void main(String args[])
    {
        int n = 4; // Number of disks
        towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    }
}
```

Python

```
 # Recursive Python function to solve tower of hanoi

def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print "Move disk 1 from rod",from_rod,"to rod",to_rod
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print "Move disk",n,"from rod",from_rod,"to rod",to_rod
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

# Driver code
n = 4
TowerOfHanoi(n, 'A', 'C', 'B')
# A, C, B are the name of rods

# Contributed By Harshit Agrawal
```

PHP

```
 <?php
//PHP code to solve Tower of Hanoi problem.

// Recursive Function to solve Tower of Hanoi
function towerOfHanoi($n, $from_rod, $to_rod, $aux_rod) {

    if ($n === 1) {
    echo ("Move disk 1 from rod $from_rod to rod $to_rod \n");
    return;
    }
    towerOfHanoi($n-1, $from_rod, $aux_rod, $to_rod);
    echo ("Move disk $n from rod $from_rod to rod $to_rod \n");
    towerOfHanoi($n-1, $aux_rod, $to_rod, $from_rod);

}

// Driver code

// number of disks
$n = 4;

// A, B and C are names of rods
towerOfHanoi($n, 'A', 'C', 'B');

// This code is contributed by akash7981
?>
```

C#

```
 // C# recursive program to solve
// tower of hanoi puzzle
using System;

class geek
{

    // C# recursive function to solve
    // tower of hanoi puzzle
    static void towerOfHanoi(int n, char from_rod,
                             char to_rod, char aux_rod)
    {
        if (n == 1)
        {
            Console.WriteLine("Move disk 1 from rod " + from_rod
                                       + " to rod " + to_rod);
            return;
        }
        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
        Console.WriteLine("Move disk " + n + " from rod "
                          + from_rod + " to rod " + to_rod);
        towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
    }

    // Driver method
    public static void Main()
    {
        // Number of disks
        int n = 4;

        // A, B and C are names of rods
        towerOfHanoi(n, 'A', 'C', 'B');
    }
}

// This code is contributed by Sam007
```

Output:

```
 Move disk 1 from rod A to rod B
 Move disk 2 from rod A to rod C
 Move disk 1 from rod B to rod C
 Move disk 3 from rod A to rod B
 Move disk 1 from rod C to rod A
 Move disk 2 from rod C to rod B
 Move disk 1 from rod A to rod B
 Move disk 4 from rod A to rod C
 Move disk 1 from rod B to rod C
 Move disk 2 from rod B to rod A
 Move disk 1 from rod C to rod A
 Move disk 3 from rod B to rod C
 Move disk 1 from rod A to rod B
 Move disk 2 from rod A to rod C
 Move disk 1 from rod B to rod C
```

For n disks, total $2^n - 1$ moves are required.

eg: For 4 disks $2^4 - 1 = 15$ moves are required.

For n disks, total $2^{n+1} - 1$ function calls are made.

eg: For 4 disks $2^{4+1} - 1 = 31$ function calls are made.
Related Articles

- Recursive Functions
- Iterative solution to TOH puzzle
- Quiz on Recursion

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

Improved By : vaibhav29498, Rustam Ali

Source

https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/

Chapter 65

Program to count number of set bits in an (big) array

Program to count number of set bits in an (big) array - GeeksforGeeks

Given an integer array of length N (an arbitrarily large number). How to count number of set bits in the array?

The simple approach would be, create an efficient method to count set bits in a word (most prominent size, usually equal to bit length of processor), and add bits from individual elements of array.

Various methods of counting set bits of an integer exists, see this for example. These methods run at best O(logN) where N is number of bits. Note that on a processor N is fixed, count can be done in O(1) time on 32 bit machine irrespective of total set bits. Overall, the bits in array can be computed in O(n) time, where 'n' is array size.

However, a table look up will be more efficient method when array size is large. Storing table look up that can handle 2^{32} integers will be impractical.

The following code illustrates simple program to count set bits in a randomly generated 64 K integer array. The idea is to generate a look up for first 256 numbers (one byte), and break every element of array at byte boundary. A meta program using C/C++ preprocessor generates the look up table for counting set bits in a byte.

The mathematical derivation behind meta program is evident from the following table (Add the column and row indices to get the number, then look into the table to get set bits in that number. For example, to get set bits in 10, it can be extracted from row named as 8 and column named as 2),

```
   0, 1, 2, 3
 0 - 0, 1, 1, 2 -------- GROUP_A(0)
 4 - 1, 2, 2, 3 -------- GROUP_A(1)
 8 - 1, 2, 2, 3 -------- GROUP_A(1)
12 - 2, 3, 3, 4 -------- GROUP_A(2)
```

```
16 - 1, 2, 2, 3 -------- GROUP_A(1)
20 - 2, 3, 3, 4 -------- GROUP_A(2)
24 - 2, 3, 3, 4 -------- GROUP_A(2)
28 - 3, 4, 4, 5 -------- GROUP_A(3) ... so on
```

From the table, there is a patten emerging in multiples of 4, both in the table as well as in the group parameter. The sequence can be generalized as shown in the code.

Complexity:

All the operations takes O(1) except iterating over the array. The time complexity is O(n) where 'n' is size of array. Space complexity depends on the meta program that generates look up.

Code:

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

/* Size of array 64 K */
#define SIZE (1 << 16)

/* Meta program that generates set bit count
   array of first 256 integers */

/* GROUP_A - When combined with META_LOOK_UP
   generates count for 4x4 elements */

#define GROUP_A(x) x, x + 1, x + 1, x + 2

/* GROUP_B - When combined with META_LOOK_UP
   generates count for 4x4x4 elements */

#define GROUP_B(x) GROUP_A(x), GROUP_A(x+1), GROUP_A(x+1), GROUP_A(x+2)

/* GROUP_C - When combined with META_LOOK_UP
   generates count for 4x4x4x4 elements */

#define GROUP_C(x) GROUP_B(x), GROUP_B(x+1), GROUP_B(x+1), GROUP_B(x+2)

/* Provide appropriate letter to generate the table */

#define META_LOOK_UP(PARAMETER) \
   GROUP_##PARAMETER(0),  \
   GROUP_##PARAMETER(1),  \
   GROUP_##PARAMETER(1),  \
   GROUP_##PARAMETER(2)   \
int countSetBits(int array[], size_t array_size)
{
   int count = 0;

   /* META_LOOK_UP(C) - generates a table of 256 integers whose
      sequence will be number of bits in i-th position
      where 0 <= i < 256
   */

    /* A static table will be much faster to access */
       static unsigned char const look_up[] = { META_LOOK_UP(C) };

    /* No shifting funda (for better readability) */
    unsigned char *pData = NULL;

   for(size_t index = 0; index < array_size; index++)
   {
      /* It is fine, bypass the type system */
      pData = (unsigned char *)&array[index];

      /* Count set bits in individual bytes */
      count += look_up[pData[0]];
      count += look_up[pData[1]];
      count += look_up[pData[2]];
      count += look_up[pData[3]];
   }

   return count;
}

/* Driver program, generates table of random 64 K numbers */
int main()
{
   int index;
   int random[SIZE];

   /* Seed to the random-number generator */
   srand((unsigned)time(0));

   /* Generate random numbers. */
   for( index = 0; index < SIZE; index++ )
   {
      random[index] = rand();
   }

   printf("Total number of bits = %d\n", countSetBits(random, SIZE));
   return 0;
}
```

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

Source

https://www.geeksforgeeks.org/program-to-count-number-of-set-bits-in-an-big-array/

Chapter 66

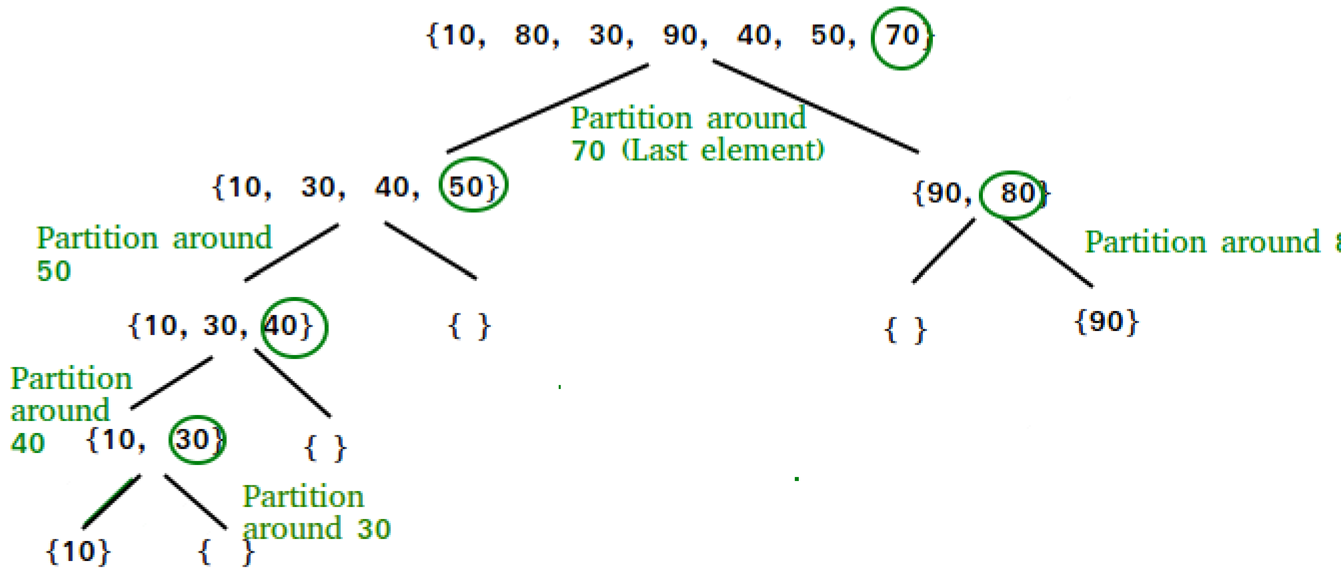
Quick Sort vs Merge Sort

Quick Sort vs Merge Sort - GeeksforGeeks

Prerequisite :Merge Sort and Quick Sort

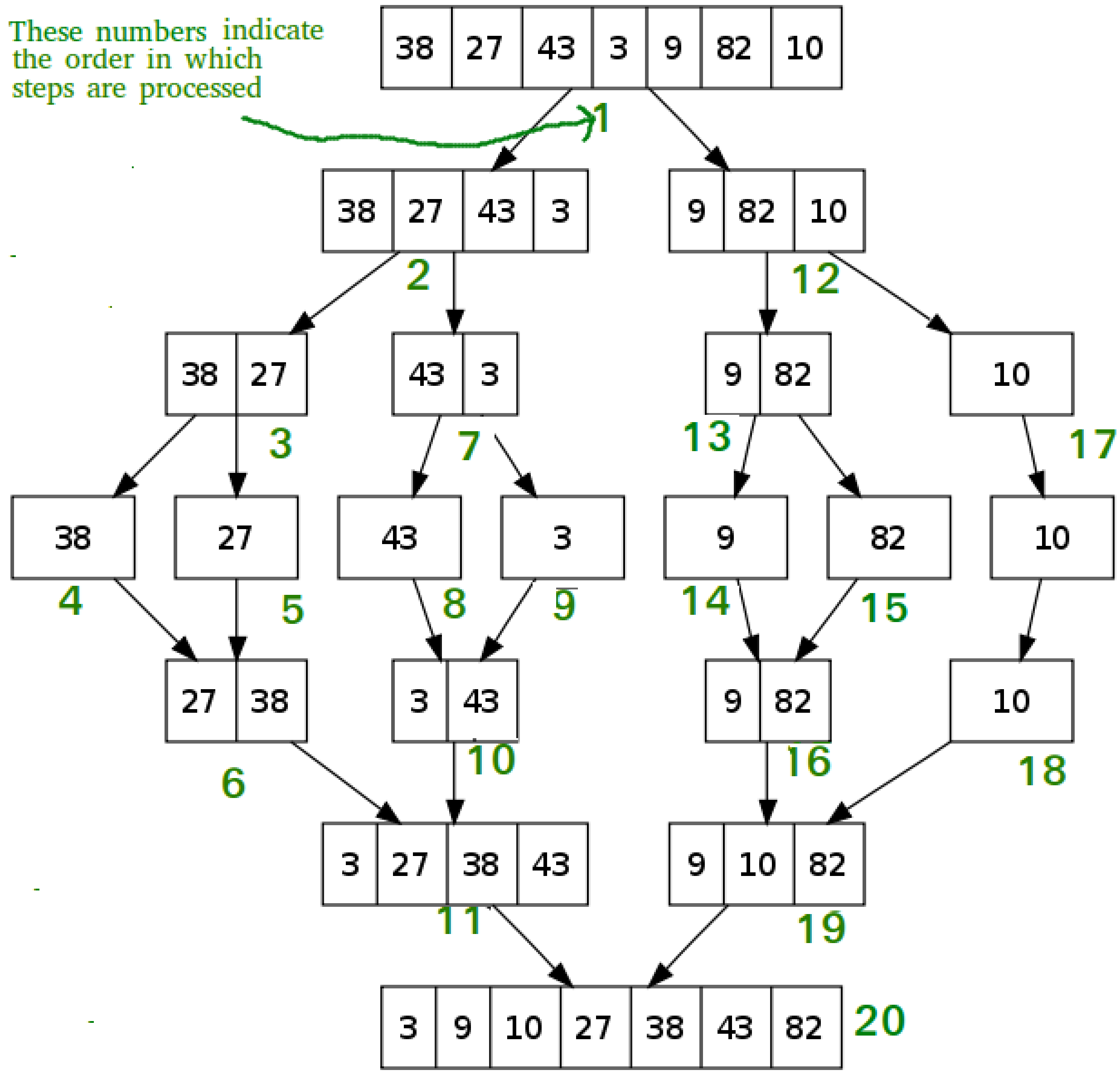
Quick sort is an internal algorithm which is based on divide and conquer strategy. In this:

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It is also known as **"partition exchange sort"**.
- It uses a key element (pivot) for partitioning the elements.
- One left partition contains all those elements that are smaller than the pivot and one right partition contains all those elements which are greater than the key element.



Merge sort is an external algorithm and based on divide and conquer strategy. In this:

- The elements are split into two sub-arrays (n/2) again and again until only one element is left.
- Merge sort uses additional storage for sorting the auxiliary array.
- Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.
- At last, the all sub arrays are merged to make it 'n' element size of the array.



1. **Partition of elements in the array** :
   In the merge sort, the array is parted into just 2 halves (i.e. n/2).
   whereas
   In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.
2. **Worst case complexity** :
   The worst case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition.
   whereas
   In merge sort, worst case and average case has same complexities O(n log n).
3. **Usage with datasets** :
   Merge sort can work well on any type of data sets irrespective of its size (either large or small).
   whereas
   The quick sort cannot work well with large datasets.
4. **Additional storage space requirement** :
   Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.
   whereas
   The quick sort is in place as it doesn't require any additional storage.
5. **Efficiency** :
   Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.
   whereas
   Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.
6. **Sorting method** :
   The quick sort is internal sorting method where the data is sorted in main memory.
   whereas
   The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.
7. **Stability** :
   Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.
   whereas
   Quick sort is unstable in this scenario. But it can be made stable using some changes in code.
8. **Preferred for** :
   Quick sort is preferred for arrays.
   whereas
   Merge sort is preferred for linked lists.
9. **Locality of reference** :
   Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	The splitting of a array of elements is in any ratio, not necessarily divided into half.
Worst case complexity	O(n2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor

Source

https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/

Chapter 67

QuickSort

QuickSort - GeeksforGeeks

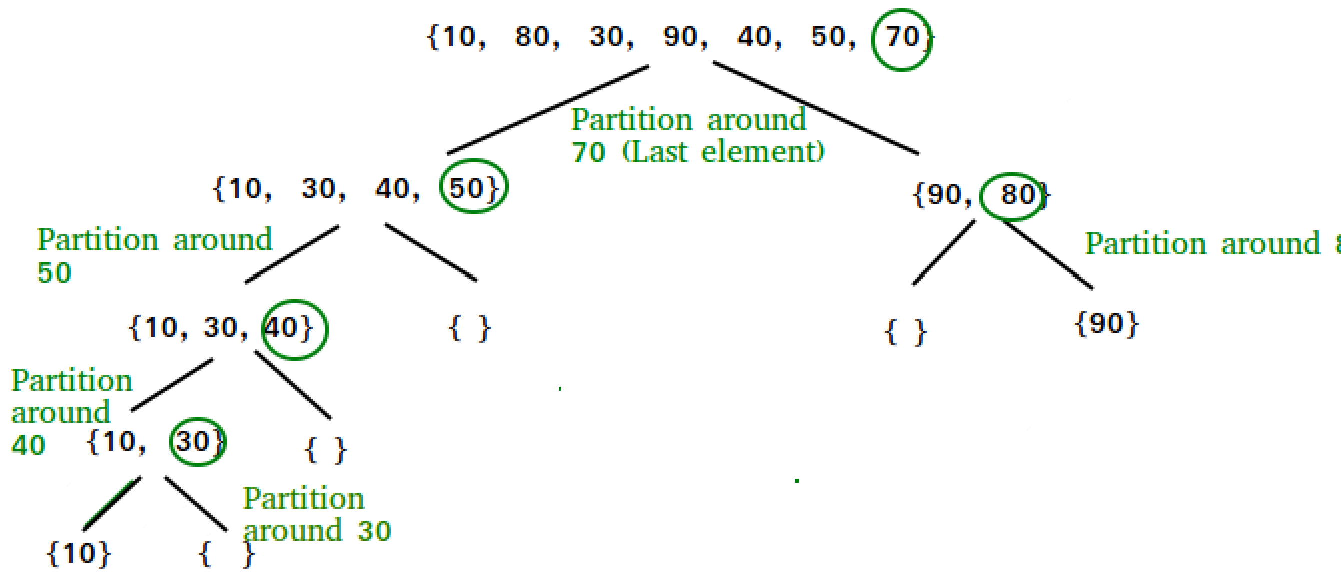
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* 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 */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
```

```
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
70 are before it and all elements greater than 70 are after
it.
```

Implementation:
Following are C++, Java and Python implementations of QuickSort.

C/C++

```
 /* C implementation QuickSort */
#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 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 low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* 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");
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

Java

```
 // Java program for implementation of QuickSort
class QuickSort
{
    /* 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 low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++)
        {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }


    /* The main function that implements QuickSort()
      arr[] --> Array to be sorted,
      low  --> Starting index,
      high  --> Ending index */
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
              now at right place */
            int pi = partition(arr, low, high);
            // Recursively sort elements before
            // partition and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
/*This code is contributed by Rajat Mishra */
```

Python

```
 # Python program for implementation of Quicksort 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
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if   arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra
```

C#

```
 // C# program for implementation of QuickSort
using System;

class GFG {

    /* 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 */
static int partition(int []arr, int low,
                                int high)
{
    int pivot = arr[high];

    // index of smaller element
    int i = (low - 1);
    for (int j = low; j < high; j++)
    {
        // If current element is smaller
        // than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++;

            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // swap arr[i+1] and arr[high] (or pivot)
    int temp1 = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp1;

    return i+1;
}


/* The main function that implements QuickSort()
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
static void quickSort(int []arr, int low, int high)
{
    if (low < high)
    {

        /* pi is partitioning index, arr[pi] is
        now at right place */
        int pi = partition(arr, low, high);

        // Recursively sort elements before
```

```
            // partition and after partition
            quickSort(arr, low, pi-1);
            quickSort(arr, pi+1, high);
        }
    }

    // A utility function to print array of size n
    static void printArray(int []arr, int n)
    {
        for (int i = 0; i < n; ++i)
            Console.Write(arr[i] + " ");

        Console.WriteLine();
    }

    // Driver program
    public static void Main()
    {
        int []arr = {10, 7, 8, 9, 1, 5};
        int n = arr.Length;
        quickSort(arr, 0, n-1);
        Console.WriteLine("sorted array ");
        printArray(arr, n);
    }
}

// This code is contributed by Sam007.
```

Output:

```
Sorted array:
1 5 7 8 9 10
```

Analysis of QuickSort
Time taken by QuickSort in general can be written as following.

```
 T(n) = T(k) + T(n-k-1) + (n)
```

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

```
 T(n) = T(0) + T(n-1) + (n)
which is equivalent to
 T(n) = T(n-1) + (n)
```

The solution of above recurrence is θ(n^2).

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

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

The solution of above recurrence is θ(nLogn). It can be solved using case 2 of Master Theorem.

Average Case:
To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

```
 T(n) = T(n/9) + T(9n/10) + (n)
```

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n^2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Is QuickSort stable?
The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

What is 3-Way QuickSort?
In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.
Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:
a) arr[l..i] elements less than pivot.
b) arr[i+1..j-1] elements equal to pivot.

c) arr[j..r] elements greater than pivot.
See this for implementation.

How to implement QuickSort for Linked Lists?
QuickSort on Singly Linked List
QuickSort on Doubly Linked List

Can we implement QuickSort Iteratively?
Yes, please refer Iterative Quick Sort.

Why Quick Sort is preferred over MergeSort for sorting Arrays
Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

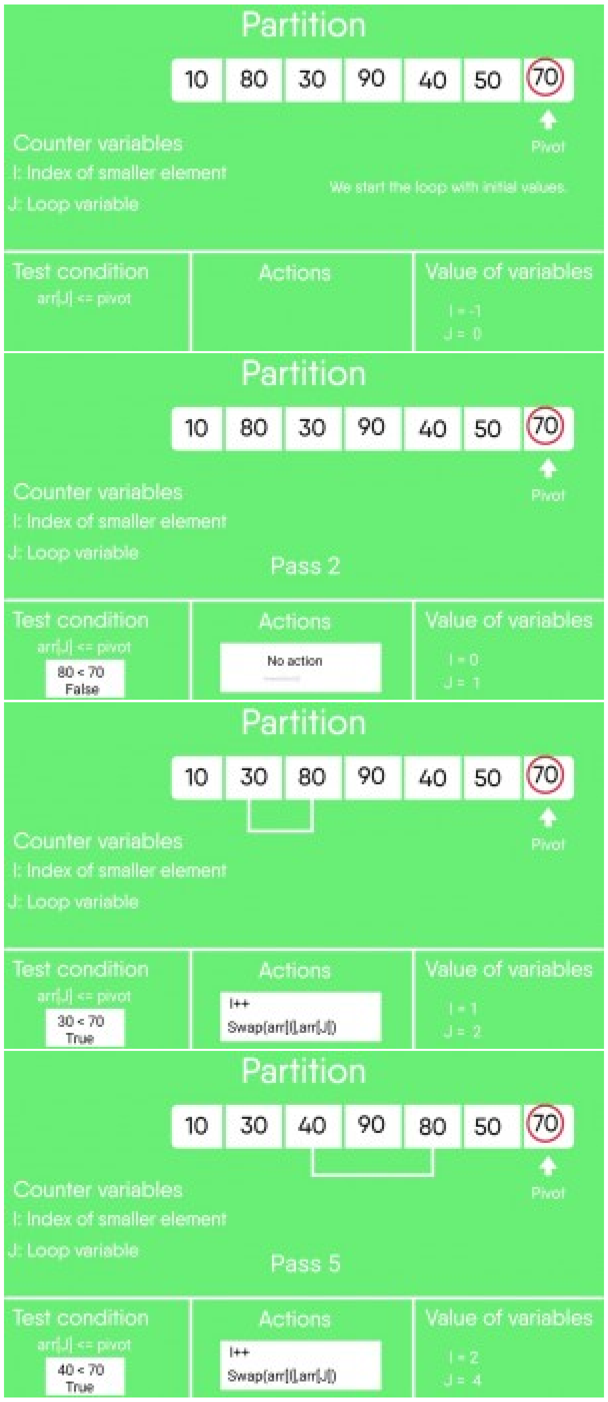
Quick Sort is also tail recursive, therefore tail call optimizations is done.

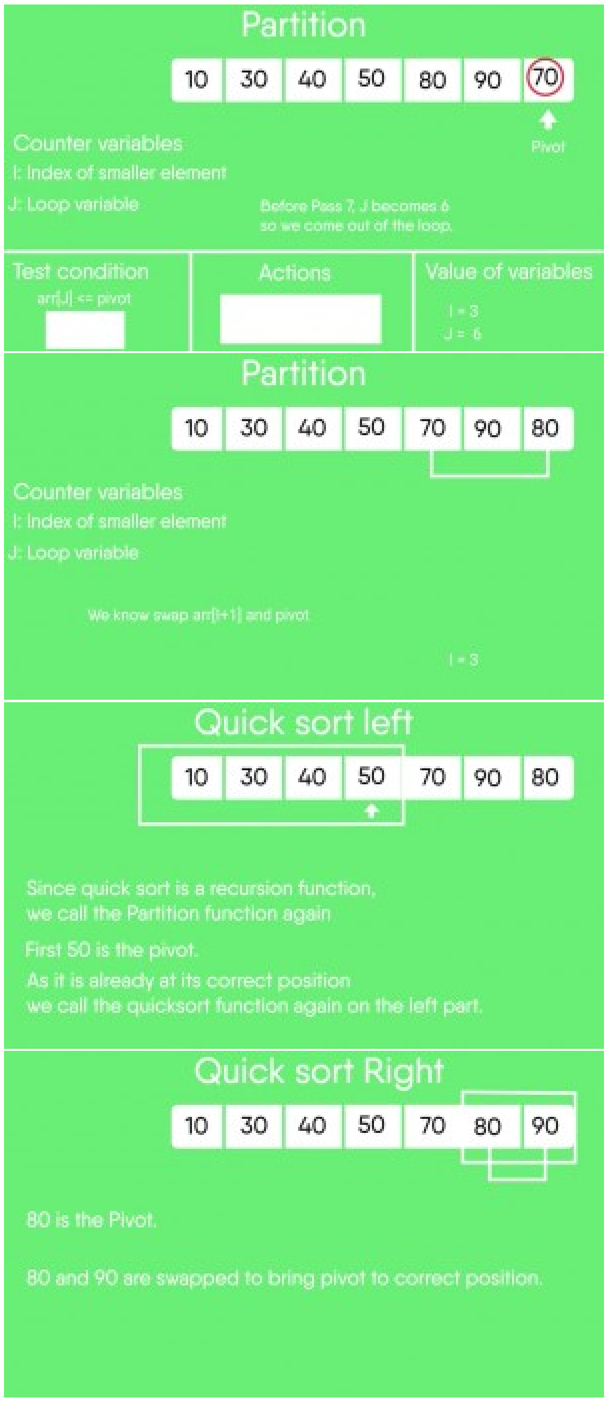
Why MergeSort is preferred over QuickSort for Linked Lists?
In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

How to optimize QuickSort so that it takes O(Log n) extra space in worst case?
Please see QuickSort Tail Call Optimization (Reducing worst case space to Log n)

Snapshots:


Partition
10 80 30 90 40 50 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
We start the loop with initial values.
Test condition
arr[J] <= pivot
Actions
Value of variables
I = -1
J = 0
Partition
10 80 30 90 40 50 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Pass 2
Test condition
arr[J] <= pivot
80 < 70
False
Actions
No action
Value of variables
I = 0
J = 1
Partition
10 30 80 90 40 50 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Test condition
arr[J] <= pivot
30 < 70
True
Actions
I++
Swap(arr[I],arr[J])
Value of variables
I = 1
J = 2
Partition
10 30 40 90 80 50 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Pass 5
Test condition
arr[J] <= pivot
40 < 70
True
Actions
I++
Swap(arr[I],arr[J])
Value of variables
I = 2
J = 4


Partition
10 30 40 50 80 90 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Before Pass 7, J becomes 6
so we come out of the loop.
Test condition
arr[J] <= pivot
Actions
Value of variables
I = 3
J = 6
Partition
10 30 40 50 70 90 80
Counter variables
I: Index of smaller element
J: Loop variable
We know swap arr[i+1] and pivot
I = 3
Quick sort left
10 30 40 50 70 90 80
Since quick sort is a recursion function,
we call the Partition function again
First 50 is the pivot.
As it is already at its correct position
we call the quicksort function again on the left part.
Quick sort Right
10 30 40 50 70 80 90
80 is the Pivot.
80 and 90 are swapped to bring pivot to correct position.

- Quiz on QuickSort
- Recent Articles on QuickSort
- Coding practice for sorting.

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

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:
Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort

Improved By : Palak Jain 5

Source

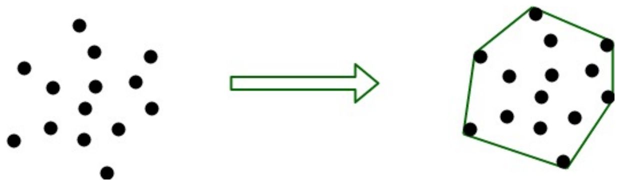
https://www.geeksforgeeks.org/quick-sort/

Chapter 68

Quickhull Algorithm for Convex Hull

Quickhull Algorithm for Convex Hull - GeeksforGeeks

Given a set of points, a Convex hull is the smallest convex polygon containing all the given points.



Input is an array of points specified by their x and y coordinates. Output is a convex hull of this set of points in ascending order of x coordinates.

Example :

```
Input : points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                    {0, 0}, {1, 2}, {3, 1}, {3, 3}};
Output :  The points in convex hull are:
          (0, 0) (0, 3) (3, 1) (4, 4)

Input : points[] = {{0, 3}, {1, 1}
Output : Not Possible
There must be at least three points to form a hull.

Input  : points[] = {(0, 0), (0, 4), (-4, 0), (5, 0),
                     (0, -6), (1, 0)};
Output : (-4, 0), (5, 0), (0, -6), (0, 4)
```

We have discussed following algorithms for Convex Hull problem.
Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping)
Convex Hull | Set 2 (Graham Scan)

The QuickHull algorithm is a Divide and Conquer algorithm similar to QuickSort. Let a[0…n-1] be the input array of points. Following are the steps for finding the convex hull of these points.

1. Find the point with minimum x-coordinate lets say, min_x and similarly the point with maximum x-coordinate, max_x.
2. Make a line joining these two points, say **L**. This line will divide the the whole set into two parts. Take both the parts one by one and proceed further.
3. For a part, find the point P with maximum distance from the line L. P forms a triangle with the points min_x, max_x. It is clear that the points residing inside this triangle can never be the part of convex hull.
4. The above step divides the problem into two sub-problems (solved recursively). Now the line joining the points P and min_x and the line joining the points P and max_x are new lines and the points residing outside the triangle is the set of points. Repeat point no. 3 till there no point left with the line. Add the end points of this point to the convex hull.

Below is C++ implementation of above idea. The implementation uses set to store points so that points can be printed in sorted order. A point is represented as a pair.

```
 // C++ program to implement Quick Hull algorithm
// to find convex hull.
#include<bits/stdc++.h>
using namespace std;

// iPair is integer pairs
#define iPair pair<int, int>

// Stores the result (points of convex hull)
set<iPair> hull;

// Returns the side of point p with respect to line
// joining points p1 and p2.
int findSide(iPair p1, iPair p2, iPair p)
{
    int val = (p.second - p1.second) * (p2.first - p1.first) -
              (p2.second - p1.second) * (p.first - p1.first);

    if (val > 0)
        return 1;
    if (val < 0)
        return -1;
    return 0;
}
```

```
// Returns the square of distance between
// p1 and p2.
int dist(iPair p, iPair q)
{
    return (p.second - q.second) * (p.second - q.second) +
           (p.first - q.first) * (p.first - q.first);
}

// returns a value proportional to the distance
// between the point p and the line joining the
// points p1 and p2
int lineDist(iPair p1, iPair p2, iPair p)
{
    return abs ((p.second - p1.second) * (p2.first - p1.first) -
               (p2.second - p1.second) * (p.first - p1.first));
}

// End points of line L are p1 and p2.  side can have value
// 1 or -1 specifying each of the parts made by the line L
void quickHull(iPair a[], int n, iPair p1, iPair p2, int side)
{
    int ind = -1;
    int max_dist = 0;

    // finding the point with maximum distance
    // from L and also on the specified side of L.
    for (int i=0; i<n; i++)
    {
        int temp = lineDist(p1, p2, a[i]);
        if (findSide(p1, p2, a[i]) == side && temp > max_dist)
        {
            ind = i;
            max_dist = temp;
        }
    }

    // If no point is found, add the end points
    // of L to the convex hull.
    if (ind == -1)
    {
        hull.insert(p1);
        hull.insert(p2);
        return;
    }

    // Recur for the two parts divided by a[ind]
    quickHull(a, n, a[ind], p1, -findSide(a[ind], p1, p2));
    quickHull(a, n, a[ind], p2, -findSide(a[ind], p2, p1));
}

void printHull(iPair a[], int n)
{
    // a[i].second -> y-coordinate of the ith point
    if (n < 3)
    {
        cout << "Convex hull not possible\n";
        return;
    }

    // Finding the point with minimum and
    // maximum x-coordinate
    int min_x = 0, max_x = 0;
    for (int i=1; i<n; i++)
    {
        if (a[i].first < a[min_x].first)
            min_x = i;
        if (a[i].first > a[max_x].first)
            max_x = i;
    }

    // Recursively find convex hull points on
    // one side of line joining a[min_x] and
    // a[max_x]
    quickHull(a, n, a[min_x], a[max_x], 1);

    // Recursively find convex hull points on
    // other side of line joining a[min_x] and
    // a[max_x]
    quickHull(a, n, a[min_x], a[max_x], -1);

    cout << "The points in Convex Hull are:\n";
    while (!hull.empty())
    {
        cout << "(" <<( *hull.begin()).first << ", "
             << (*hull.begin()).second << ") ";
        hull.erase(hull.begin());
    }
}

// Driver code
int main()
{
    iPair a[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                 {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    int n = sizeof(a)/sizeof(a[0]);
    printHull(a, n);
    return 0;
}
```

Input :

```
The points in Convex Hull are:
(0, 0) (0, 3) (3, 1) (4, 4)
```

Time Complexity: The analysis is similar to Quick Sort. On average, we get time complexity as O(n Log n), but in worst case, it can become O(n^2)

Source

https://www.geeksforgeeks.org/quickhull-algorithm-convex-hull/

Chapter 69

Randomized Binary Search Algorithm

Randomized Binary Search Algorithm - GeeksforGeeks

We are given a sorted array A[] of n elements. We need to find if x is present in A or not.In binary search we always used middle element, here we will randomly pick one element in given range.

In Binary Search we had

```
middle = (start + end)/2
```

In Randomized binary search we do following

```
Generate a random number t
Since range of number in which we want a random
number is [start, end]
Hence we do, t = t % (end-start+1)
Then, t = start + t;
Hence t is a random number between start and end
```

It is a Las Vegas randomized algorithm as it always finds the correct result.

Expected Time complexity of Randomized Binary Search Algorithm
For n elements let say expected time required be T(n), After we choose one random pivot, array size reduces to say k. Since pivot is chosen with equal probability for all possible pivots, hence p = 1/n.

T(n) is sum of time of all possible sizes after choosing pivot multiplied by probability of choosing that pivot plus time take to generate random pivot index.Hence

```
T(n) = p*T(1) + p*T(2) + ..... + p*T(n) + 1
putting p = 1/n
T(n) = ( T(1) + T(2) + ..... + T(n) ) / n + 1
n*T(n) = T(1) + T(2) + .... + T(n) + n      .... eq(1)
Similarly for n-1
(n-1)*T(n-1) = T(1) + T(2) + ..... + T(n-1) + n-1    .... eq(2)
Subtract eq(1) - eq(2)
n*T(n) - (n-1)*T(n-1) = T(n) + 1
(n-1)*T(n) - (n-1)*T(n-1) =  1
(n-1)*T(n) = (n-1)*T(n-1) + 1
T(n) = 1/(n-1) + T(n-1)
T(n) = 1/(n-1) + 1/(n-2) + T(n-2)
T(n) = 1/(n-1) + 1/(n-2) + 1/(n-3) + T(n-3)
Similarly,
T(n) = 1 + 1/2 + 1/3 + ... + 1/(n-1)
Hence T(n) is equal to (n-1)th Harmonic number,
n-th harmonic number is O(log n)
Hence T(n) is O(log n)
```

Recursive C++ implementation of Randomized Binary Search

```
 // C++ program to implement recursive
// randomized algorithm.
#include <iostream>
#include <ctime>
using namespace std;

// To generate random number
// between x and y ie.. [x, y]
int getRandom(int x, int y)
{
    srand(time(NULL));
    return (x + rand() % (y-x+1));
}

// A recursive randomized binary search function.
// It returns location of x in
// given array arr[l..r] is present, otherwise -1
int randomizedBinarySearch(int arr[], int l,
                            int r, int x)
{
    if (r >= l)
    {
        // Here we have defined middle as
        // random index between l and r ie.. [l, r]
        int mid = getRandom(l, r);
        // 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 randomizedBinarySearch(arr, l,
                                    mid-1, x);

        // Else the element can only be present
        // in right subarray
        return randomizedBinarySearch(arr, mid+1,
                                         r, x);
    }

    // We reach here when element is not present
    // in array
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = randomizedBinarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
    : printf("Element is present at index %d", result);
    return 0;
}
```

Output:

```
Element is present at index 3
```

Iterative C++ implementation of Randomized Binary Search

```
 // C++ program to implement iterative
// randomized algorithm.
#include <iostream>
#include <ctime>
using namespace std;
```

```
// To generate random number
// between x and y ie.. [x, y]
int getRandom(int x, int y)
{
    srand(time(NULL));
    return (x + rand()%(y-x+1));
}

// A iterative randomized binary search function.
// It returns location of x in
// given array arr[l..r] if present, otherwise -1
int randomizedBinarySearch(int arr[], int l,
                               int r, int x)
{
    while (l <= r)
    {
        // Here we have defined middle as
        // random index between l and r ie.. [l, r]
        int m = getRandom(l, r);

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = randomizedBinarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

Output:

```
Element is present at index 3
```

Source

https://www.geeksforgeeks.org/randomized-binary-search-algorithm/

Chapter 70

Search element in a sorted matrix

Search element in a sorted matrix - GeeksforGeeks

Given a sorted matrix mat[n][m] and an element 'x'. Find position of x in the matrix if it is present, else print -1. Matrix is sorted in a way such that all elements in a row are sorted in increasing order and for row 'i', where 1 <= i <= n-1, first element of row 'i' is greater than or equal to the last element of row 'i-1'. The approach should have O(log n + log m) time complexity. Examples:

```
Input : mat[][] = { {1, 5, 9},
                    {14, 20, 21},
                    {30, 34, 43} }
        x = 14
Output : Found at (1, 0)

Input : mat[][] = { {1, 5, 9, 11},
                    {14, 20, 21, 26},
                    {30, 34, 43, 50} }
        x = 42
Output : -1
```

Please note that this problem is different from Search in a row wise and column wise sorted matrix. Here matrix is more strictly sorted as first element of a row is greater than last element of previous row.

A **Simple Solution** is to one by one compare x with every element of matrix. If matches, then return position. If we reach end, return -1. Time complexity of this solution is O(n x m).

An **efficient solution** is to typecast given 2D array to 1D array, then apply binary search on the typecasted array.

Another efficient approach that doesn't require typecasting is explained below.

```
1) Perform binary search on the middle column
   till only two elements are left or till the
   middle element of some row in the search is
   the required element 'x'. This search is done
   to skip the rows that are not required
2) The two left elements must be adjacent. Consider
   the rows of two elements and do following
   a) check whether the element 'x' equals to the
      middle element of any one of the 2 rows
   b) otherwise according to the value of the
      element 'x' check whether it is present in
      the 1st half of 1st row, 2nd half of 1st row,
      1st half of 2nd row or 2nd half of 2nd row.

Note: This approach works for the matrix n x m
      where 2 <= n. The algorithm can be modified
      for matrix 1 x m, we just need to check whether
      2nd row exists or not
```

Example:

```
Consider:     | 1  2  3  4|
x = 3, mat = | 5  6  7  8|   Middle column:
             | 9 10 11 12|    = {2, 6, 10, 14}
             |13 14 15 16|   perform binary search on them
                              since, x < 6, discard the
                              last 2 rows as 'a' will
                              not lie in them(sorted matrix)
Now, only two rows are left
             | 1  2  3  4|
x = 3, mat = | 5  6  7  8|   Check whether element is present
                              on the middle elements of these
                              rows = {2, 6}
                              x != 2 or 6
If not, consider the four sub-parts
1st half of 1st row = {1}, 2nd half of 1st row = {3, 4}
1st half of 2nd row = {5}, 2nd half of 2nd row = {7, 8}

According the value of 'x' it will be searched in the
2nd half of 1st row = {3, 4} and found at (i, j): (0, 2)
```

C++

```
 // C++ implementation to search an element in a
// sorted matrix
#include <bits/stdc++.h>
using namespace std;

const int MAX = 100;

// This function does Binary search for x in i-th
// row. It does the search from mat[i][j_low] to
// mat[i][j_high]
void binarySearch(int mat[][MAX], int i, int j_low,
                                int j_high, int x)
{
    while (j_low <= j_high)
    {
        int j_mid = (j_low + j_high) / 2;

        // Element found
        if (mat[i][j_mid] == x)
        {
            cout << "Found at (" << i << ", "
                 << j_mid << ")";
            return;
        }

        else if (mat[i][j_mid] > x)
            j_high = j_mid - 1;

        else
            j_low = j_mid + 1;
    }

    // element not found
    cout << "Element no found";
}

// Function to perform binary search on the mid
// values of row to get the desired pair of rows
// where the element can be found
void sortedMatrixSearch(int mat[][MAX], int n,
                                  int m, int x)
{
    // Single row matrix
    if (n == 1)
    {
        binarySearch(mat, 0, 0, m-1, x);
        return;
    }
```

```
// Do binary search in middle column.
// Condition to terminate the loop when the
// 2 desired rows are found
int i_low = 0;
int i_high = n-1;
int j_mid = m/2;
while ((i_low+1) < i_high)
{
    int i_mid = (i_low + i_high) / 2;

    // element found
    if (mat[i_mid][j_mid] == x)
    {
        cout << "Found at (" << i_mid << ", "
             << j_mid << ")";
        return;
    }

    else if (mat[i_mid][j_mid] > x)
        i_high = i_mid;

    else
        i_low = i_mid;
}

// If element is present on the mid of the
// two rows
if (mat[i_low][j_mid] == x)
    cout << "Found at (" << i_low << ","
         << j_mid << ")";
else if (mat[i_low+1][j_mid] == x)
    cout << "Found at (" << (i_low+1)
         << ", " << j_mid << ")";

// Ssearch element on 1st half of 1st row
else if (x <= mat[i_low][j_mid-1])
    binarySearch(mat, i_low, 0, j_mid-1, x);

// Search element on 2nd half of 1st row
else if (x >= mat[i_low][j_mid+1]  &&
         x <= mat[i_low][m-1])
   binarySearch(mat, i_low, j_mid+1, m-1, x);

// Search element on 1st half of 2nd row
else if (x <= mat[i_low+1][j_mid-1])
    binarySearch(mat, i_low+1, 0, j_mid-1, x);
    // search element on 2nd half of 2nd row
    else
        binarySearch(mat, i_low+1, j_mid+1, m-1, x);
}

// Driver program to test above
int main()
{
    int n = 4, m = 5, x = 8;
    int mat[][MAX] = {{0, 6, 8, 9, 11},
                     {20, 22, 28, 29, 31},
                     {36, 38, 50, 61, 63},
                     {64, 66, 100, 122, 128}};

    sortedMatrixSearch(mat, n, m, x);
    return 0;
}
```

Java

```
 // java implementation to search
// an element in a sorted matrix
import java.io.*;

class GFG
{
    static int MAX = 100;

    // This function does Binary search for x in i-th
    // row. It does the search from mat[i][j_low] to
    // mat[i][j_high]
    static void binarySearch(int mat[][], int i, int j_low,
                                int j_high, int x)
    {
        while (j_low <= j_high)
        {
            int j_mid = (j_low + j_high) / 2;

            // Element found
            if (mat[i][j_mid] == x)
            {
                System.out.println ( "Found at (" + i
                                     + ", " + j_mid +")");
                return;
            }

            else if (mat[i][j_mid] > x)
                j_high = j_mid - 1;
            else
                j_low = j_mid + 1;
        }

        // element not found
        System.out.println ( "Element no found");
    }

    // Function to perform binary search on the mid
    // values of row to get the desired pair of rows
    // where the element can be found
    static void sortedMatrixSearch(int mat[][], int n,
                                       int m, int x)
    {
        // Single row matrix
        if (n == 1)
        {
            binarySearch(mat, 0, 0, m - 1, x);
            return;
        }

        // Do binary search in middle column.
        // Condition to terminate the loop when the
        // 2 desired rows are found
        int i_low = 0;
        int i_high = n - 1;
        int j_mid = m / 2;
        while ((i_low + 1) < i_high)
        {
            int i_mid = (i_low + i_high) / 2;

            // element found
            if (mat[i_mid][j_mid] == x)
            {
                System.out.println ( "Found at (" + i_mid +", "
                                     + j_mid +")");
                return;
            }

            else if (mat[i_mid][j_mid] > x)
                i_high = i_mid;

            else
                i_low = i_mid;
        }

        // If element is present on
        // the mid of the two rows
        if (mat[i_low][j_mid] == x)
            System.out.println ( "Found at (" + i_low + ","
                                 + j_mid +")");
        else if (mat[i_low + 1][j_mid] == x)
            System.out.println ( "Found at (" + (i_low + 1)
                               + ", " + j_mid +")");

        // Ssearch element on 1st half of 1st row
        else if (x <= mat[i_low][j_mid - 1])
            binarySearch(mat, i_low, 0, j_mid - 1, x);

        // Search element on 2nd half of 1st row
        else if (x >= mat[i_low][j_mid + 1] &&
                 x <= mat[i_low][m - 1])
        binarySearch(mat, i_low, j_mid + 1, m - 1, x);

        // Search element on 1st half of 2nd row
        else if (x <= mat[i_low + 1][j_mid - 1])
            binarySearch(mat, i_low + 1, 0, j_mid - 1, x);

        // search element on 2nd half of 2nd row
        else
            binarySearch(mat, i_low + 1, j_mid + 1, m - 1, x);
    }

    // Driver program
    public static void main (String[] args)
    {
        int n = 4, m = 5, x = 8;
        int mat[][] = {{0, 6, 8, 9, 11},
                       {20, 22, 28, 29, 31},
                       {36, 38, 50, 61, 63},
                       {64, 66, 100, 122, 128}};

        sortedMatrixSearch(mat, n, m, x);

    }
}

// This code is contributed by vt_m
```

C#

```
 // C# implementation to search
// an element in a sorted matrix
using System;
```

```
class GFG
{
    // This function does Binary search for x in i-th
    // row. It does the search from mat[i][j_low] to
    // mat[i][j_high]
    static void binarySearch(int [,]mat, int i, int j_low,
                                    int j_high, int x)
    {
        while (j_low <= j_high)
        {
            int j_mid = (j_low + j_high) / 2;

            // Element found
            if (mat[i,j_mid] == x)
            {
                Console.Write ( "Found at (" + i +
                                ", " + j_mid +")");
                return;
            }

            else if (mat[i,j_mid] > x)
                j_high = j_mid - 1;

            else
                j_low = j_mid + 1;
        }

        // element not found
        Console.Write ( "Element no found");
    }

    // Function to perform binary search on the mid
    // values of row to get the desired pair of rows
    // where the element can be found
    static void sortedMatrixSearch(int [,]mat, int n,
                                    int m, int x)
    {
        // Single row matrix
        if (n == 1)
        {
            binarySearch(mat, 0, 0, m - 1, x);
            return;
        }

        // Do binary search in middle column.
        // Condition to terminate the loop when the
        // 2 desired rows are found
        int i_low = 0;
        int i_high = n - 1;
        int j_mid = m / 2;
        while ((i_low + 1) < i_high)
        {
            int i_mid = (i_low + i_high) / 2;

            // element found
            if (mat[i_mid,j_mid] == x)
            {

                Console.Write ( "Found at (" + i_mid +
                                ", "    + j_mid +")");
                return;
            }

            else if (mat[i_mid,j_mid] > x)
                i_high = i_mid;

            else
                i_low = i_mid;
        }

        // If element is present on
        // the mid of the two rows
        if (mat[i_low,j_mid] == x)
        Console.Write ( "Found at (" + i_low +
                           "," + j_mid +")");
        else if (mat[i_low + 1,j_mid] == x)
        Console.Write ( "Found at (" + (i_low
                   + 1) + ", " + j_mid +")");

        // Ssearch element on 1st half of 1st row
        else if (x <= mat[i_low,j_mid - 1])
            binarySearch(mat, i_low, 0, j_mid - 1, x);

        // Search element on 2nd half of 1st row
        else if (x >= mat[i_low,j_mid + 1] &&
                 x <= mat[i_low,m - 1])
        binarySearch(mat, i_low, j_mid + 1, m - 1, x);

        // Search element on 1st half of 2nd row
        else if (x <= mat[i_low + 1,j_mid - 1])
            binarySearch(mat, i_low + 1, 0, j_mid - 1, x);

        // search element on 2nd half of 2nd row
        else
            binarySearch(mat, i_low + 1, j_mid + 1, m - 1, x);
    }
    // Driver program
    public static void Main (String[] args)
    {
        int n = 4, m = 5, x = 8;
        int [,]mat = {{0, 6, 8, 9, 11},
                    {20, 22, 28, 29, 31},
                    {36, 38, 50, 61, 63},
                    {64, 66, 100, 122, 128}};

        sortedMatrixSearch(mat, n, m, x);
    }
}

// This code is contributed by parashar...
```

Output:

```
Found at (0,2)
```

Time complexity: O(log n + log m). O(Log n) time is required to find the two desired rows. Then O(Log m) time is required for binary search in one of the four parts with size equal to m/2.

Improved By : parashar

Source

https://www.geeksforgeeks.org/search-element-sorted-matrix/

Chapter 71

Search equal, bigger or smaller in a sorted array in Java

Search equal, bigger or smaller in a sorted array in Java - GeeksforGeeks

Given array of sorted integer, search key and search preferences find array position. A search preferences can be:
1) EQUAL – search only for equal key or -1 if not found. It's a regular binary search.
2) EQUAL_OR_SMALLER – search only for equal key or the closest smaller. -1 if not found.
3) EQUAL_OR_BIGGER – search only for equal key or the closest bigger. -1 if not found.

Examples:

```
Input : { 0, 2, 4, 6 }, key -1, EQUAL
Output : -1

Input : { 0, 2, 4, 6 }, key -1, EQUAL_OR_BIGGER
Output : 0

Input : { 0, 2, 4, 6 }, key 7, EQUAL_OR_BIGGER
Output : -1

Input : { 0, 2, 4, 6 }, key 7, EQUAL_OR_SMALLER
Output : 3
```

In regular binary search algorithm evaluation and division perform as far as subarray size is bigger than 0.
In our case if we want to keep single function we need to perform final evaluation in subarray of size=2. Only in subarray size==2 is possible to check both EQUAL_OR_SMALLER and EQUAL_OR_BIGGER conditions.

In below code, **SC stands for Search Criteria**.

```
public class BinarySearchEqualOrClose {

   private static void printResult(int key, int pos,
                                SC sC)
   {
       System.out.println("" + key + ", " + sC + ":" + pos);
   }

   enum SC {
       EQUAL,
       EQUAL_OR_BIGGER,
       EQUAL_OR_SMALLER
   };

   public static int searchEqualOrClose(int key, int[] arr, SC sC)
   {
       if (arr == null || arr.length == 0) {
           return -1;
       }

       if (arr.length == 1) { // just eliminate case of length==1

           // since algorithm needs min array size==2
           // to start final evaluations
           if (arr[0] == key) {
               return 0;
           }
           if (arr[0] < key && sC == SC.EQUAL_OR_SMALLER) {
               return 0;
           }
           if (arr[0] > key && sC == SC.EQUAL_OR_BIGGER) {
               return 0;
           }
           return -1;
       }
       return searchEqualOrClose(arr, key, 0, arr.length - 1, sC);
   }

   private static int searchEqualOrClose(int[] arr, int key,
                                      int start, int end, SC sC)
   {
       int midPos = (start + end) / 2;
       int midVal = arr[midPos];
       if (midVal == key) {
           return midPos; // equal is top priority
       }

       if (start >= end - 1) {
           if (arr[end] == key) {
               return end;
           }
           if (sC == SC.EQUAL_OR_SMALLER) {

               // find biggest of smaller element
               if (arr[start] > key && start != 0) {

                   // even before if "start" is not a first
                   return start - 1;
               }
               if (arr[end] < key) {
                   return end;
               }
               if (arr[start] < key) {
                   return start;
               }
               return -1;
           }
           if (sC == SC.EQUAL_OR_BIGGER) {

               // find smallest of bigger element
               if (arr[end] < key && end != arr.length - 1) {

                   // even after if "end" is not a last
                   return end + 1;
               }
               if (arr[start] > key) {
                   return start;
               }
               if (arr[end] > key) {
                   return end;
               }
               return -1;
           }
           return -1;
       }
       if (midVal > key) {
           return searchEqualOrClose(arr, key, start, midPos - 1, sC);
       }
       return searchEqualOrClose(arr, key, midPos + 1, end, sC);
   }

   public static void main(String[] args)
   {
       int[] arr = new int[] { 0, 2, 4, 6 };

       // test full range of xs and SearchCriteria
```

```
        for (int x = -1; x <= 7; x++) {
            int pos = searchEqualOrClose(x, arr, SC.EQUAL);
            printResult(x, pos, SC.EQUAL);
            pos = searchEqualOrClose(x, arr, SC.EQUAL_OR_SMALLER);
            printResult(x, pos, SC.EQUAL_OR_SMALLER);
            pos = searchEqualOrClose(x, arr, SC.EQUAL_OR_BIGGER);
            printResult(x, pos, SC.EQUAL_OR_BIGGER);
        }
    }
}
```

Output:

```
-1, EQUAL:-1
-1, EQUAL_OR_SMALLER:-1
-1, EQUAL_OR_BIGGER:0
0, EQUAL:0
0, EQUAL_OR_SMALLER:0
0, EQUAL_OR_BIGGER:0
1, EQUAL:-1
1, EQUAL_OR_SMALLER:0
1, EQUAL_OR_BIGGER:1
2, EQUAL:1
2, EQUAL_OR_SMALLER:1
2, EQUAL_OR_BIGGER:1
3, EQUAL:-1
3, EQUAL_OR_SMALLER:1
3, EQUAL_OR_BIGGER:2
4, EQUAL:2
4, EQUAL_OR_SMALLER:2
4, EQUAL_OR_BIGGER:2
5, EQUAL:-1
5, EQUAL_OR_SMALLER:2
5, EQUAL_OR_BIGGER:3
6, EQUAL:3
6, EQUAL_OR_SMALLER:3
6, EQUAL_OR_BIGGER:3
7, EQUAL:-1
7, EQUAL_OR_SMALLER:3
7, EQUAL_OR_BIGGER:-1
```

Time Complexity: O(log n)

Source

https://www.geeksforgeeks.org/search-equal-bigger-or-smaller-in-a-sorted-array-in-java/

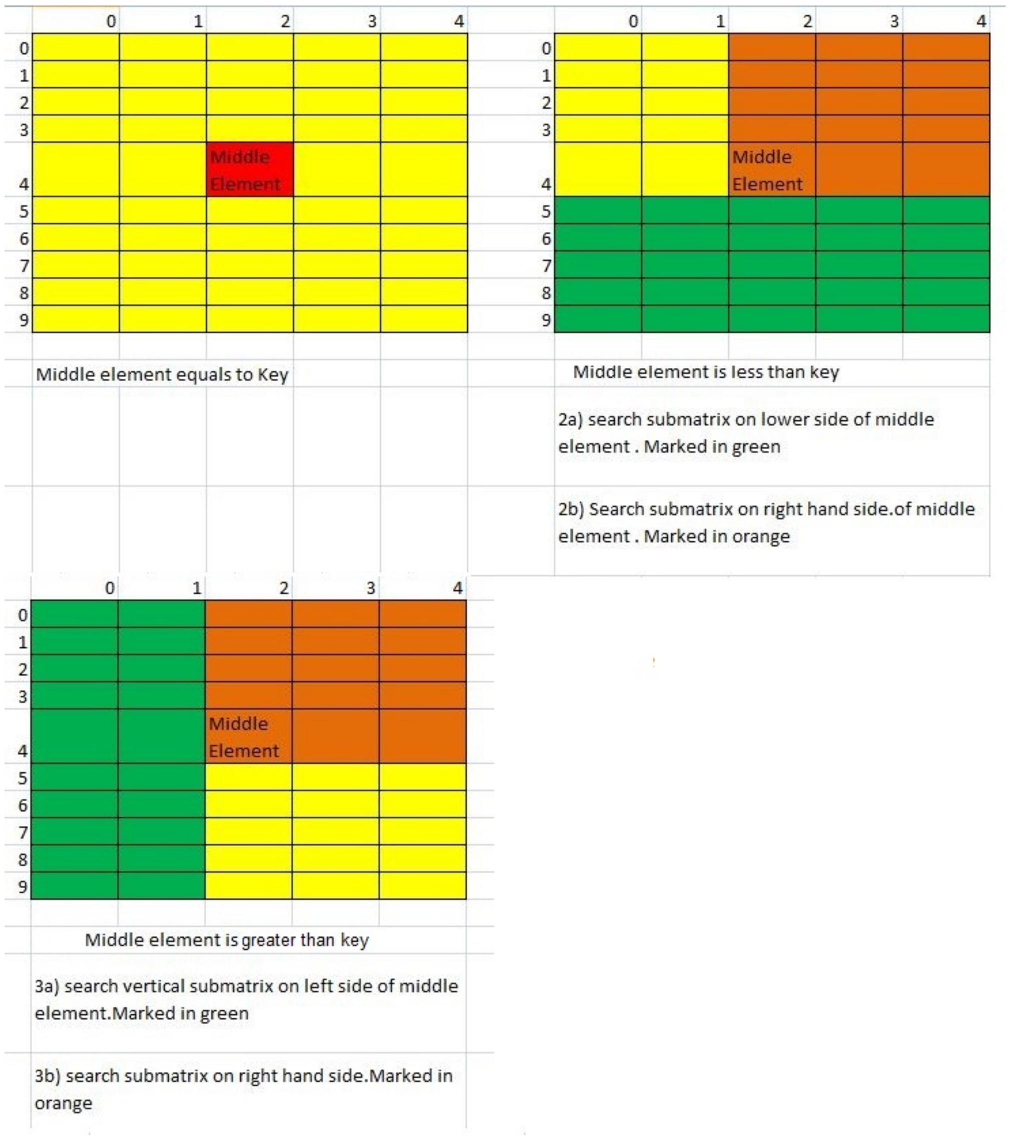
Chapter 72

Search in a Row-wise and Column-wise Sorted 2D Array using Divide and Conquer algorithm

Search in a Row-wise and Column-wise Sorted 2D Array using Divide and Conquer algorithm - GeeksforGeeks

Given an n x n matrix, where every row and column is sorted in increasing order. Given a key, how to decide whether this key is in the matrix.
A linear time complexity is discussed in the previous post. This problem can also be a very good example for divide and conquer algorithms. Following is divide and conquer algorithm.

1) Find the middle element.
2) If middle element is same as key return.
3) If middle element is lesser than key then
….3a) search submatrix on lower side of middle element
….3b) Search submatrix on right hand side.of middle element
4) If middle element is greater than key then
….4a) search vertical submatrix on left side of middle element
….4b) search submatrix on right hand side.


Middle Element
Middle element equals to Key
Middle Element
Middle element is less than key
2a) search submatrix on lower side of middle element . Marked in green
2b) Search submatrix on right hand side.of middle element . Marked in orange
Middle Element
Middle element is greater than key
3a) search vertical submatrix on left side of middle element.Marked in green
3b) search submatrix on right hand side.Marked in orange

Following Java implementation of above algorithm.

```
 // Java program for implementation of divide and conquer algorithm
// to find a given key in a row-wise and column-wise sorted 2D array
class SearchInMatrix
{
    public static void main(String[] args)
    {
        int[][] mat = new int[][] { {10, 20, 30, 40},
                                    {15, 25, 35, 45},
                                    {27, 29, 37, 48},
                                    {32, 33, 39, 50}};
        int rowcount = 4,colCount=4,key=50;
        for (int i=0; i<rowcount; i++)
          for (int j=0; j<colCount; j++)
             search(mat, 0, rowcount-1, 0, colCount-1, mat[i][j]);
    }

    // A divide and conquer method to search a given key in mat[]
    // in rows from fromRow to toRow and columns from fromCol to
    // toCol
    public static void search(int[][] mat, int fromRow, int toRow,
                              int fromCol, int toCol, int key)
    {
        // Find middle and compare with middle
        int i = fromRow + (toRow-fromRow )/2;
        int j = fromCol + (toCol-fromCol )/2;
        if (mat[i][j] == key) // If key is present at middle
          System.out.println("Found "+ key + " at "+ i +
                               " " + j);
        else
        {
            // right-up quarter of matrix is searched in all cases.
            // Provided it is different from current call
            if (i!=toRow || j!=fromCol)
             search(mat,fromRow,i,j,toCol,key);

            // Special case for iteration with 1*2 matrix
            // mat[i][j] and mat[i][j+1] are only two elements.
            // So just check second element
            if (fromRow == toRow && fromCol + 1 == toCol)
              if (mat[fromRow][toCol] == key)
                System.out.println("Found "+ key+ " at "+
                                   fromRow + " " + toCol);

            // If middle key is lesser then search lower horizontal
            // matrix and right hand side matrix
            if (mat[i][j] < key)
            {
                // search lower horizontal if such matrix exists
                if (i+1<=toRow)
                  search(mat, i+1, toRow, fromCol, toCol, key);
            }

            // If middle key is greater then search left vertical
            // matrix and right hand side matrix
            else
            {
                // search left vertical if such matrix exists
                if (j-1>=fromCol)
```

```
                  search(mat, fromRow, toRow, fromCol, j-1, key);
            }
        }
    }
}
```

Time complexity:
We are given a n*n matrix, the algorithm can be seen as recurring for 3 matrices of size n/2 x n/2. Following is recurrence for time complexity

```
 T(n) = 3T(n/2) + O(1)
```

The solution of recurrence is O(n$^{1.58}$) using Master Method.
But the actual implementation calls for one submatrix of size n x n/2 or n/2 x n, and other submatrix of size n/2 x n/2.

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

Source

https://www.geeksforgeeks.org/search-in-a-row-wise-and-column-wise-sorted-2d-array-using-divide-and-conquer-al

Chapter 73

Sequences of given length where every element is more than or equal to twice of previous

Sequences of given length where every element is more than or equal to twice of previous - GeeksforGeeks

Given two integers m & n, find the number of possible sequences of length n such that each of the next element is greater than or equal to twice of the previous element but less than or equal to m.

Examples :

```
Input : m = 10, n = 4
Output : 4
There should be n elements and value of last
element should be at-most m.
The sequences are {1, 2, 4, 8}, {1, 2, 4, 9},
                 {1, 2, 4, 10}, {1, 2, 5, 10}

Input : m = 5, n = 2
Output : 6
The sequences are {1, 2}, {1, 3}, {1, 4},
                  {1, 5}, {2, 4}, {2, 5}
```

As per the given condition the n-th value of the sequence can be at most m. There can be two cases for n-th element:

1. If it is m, then the (n-1)th element is at most m/2. We recur for m/2 and n-1.
2. If it is not m, then the n-1th element is at most is m-1. We recur for (m-1) and n.

The total number of sequences is the sum of the number of sequences including m and the number of sequences where m is not included. Thus the original problem of finding number of sequences of length n with max value m can be subdivided into independent subproblems of finding number of sequences of length n with max value m-1 and number of sequences of length n-1 with max value m/2.

C++

```
 // C program to count total number of special sequences
// of length n where
#include <stdio.h>

// Recursive function to find the number of special
// sequences
int  getTotalNumberOfSequences(int m, int n)
{
    // A special sequence cannot exist if length
    // n is more than the maximum value m.
    if (m < n)
        return 0;

    // If n is 0, found an empty special sequence
    if (n == 0)
        return 1;

    // There can be two possibilities : (1) Reduce
    // last element value (2) Consider last element
    // as m and reduce number of terms
    return getTotalNumberOfSequences (m-1, n) +
           getTotalNumberOfSequences (m/2, n-1);
}

// Driver Code
int main()
{
    int m = 10;
    int n = 4;
    printf("Total number of possible sequences %d",
                   getTotalNumberOfSequences(m, n));
    return 0;
}
```

Java

```
 // Java program to count total number
// of special sequences of length n where
class Sequences
{
    // Recursive function to find the number of special
    // sequences
    static int  getTotalNumberOfSequences(int m, int n)
    {
        // A special sequence cannot exist if length
        // n is more than the maximum value m.
        if(m < n)
            return 0;

        // If n is 0, found an empty special sequence
        if(n == 0)
            return 1;

        // There can be two possibilities : (1) Reduce
        // last element value (2) Consider last element
        // as m and reduce number of terms
        return getTotalNumberOfSequences (m-1, n) +
               getTotalNumberOfSequences (m/2, n-1);
    }

    // main function
    public static void main (String[] args)
    {
        int m = 10;
        int n = 4;
        System.out.println("Total number of possible sequences "+
                      getTotalNumberOfSequences(m, n));
    }
}
```

C#

```
 // C# program to count total number
// of special sequences of length n
// where every element is more than
// or equal to twice of previous
using System;

class GFG
{
    // Recursive function to find
    // the number of special sequences
    static int getTotalNumberOfSequences(int m, int n)
    {
        // A special sequence cannot exist if length
        // n is more than the maximum value m.
        if(m < n)
            return 0;
        // If n is 0, found an empty special sequence
        if(n == 0)
            return 1;

        // There can be two possibilities : (1) Reduce
        // last element value (2) Consider last element
        // as m and reduce number of terms
        return getTotalNumberOfSequences (m-1, n) +
               getTotalNumberOfSequences (m/2, n-1);
    }

    // Driver code
    public static void Main ()
    {
        int m = 10;
        int n = 4;
        Console.Write("Total number of possible sequences " +
                          getTotalNumberOfSequences(m, n));
    }
}

// This code is contributed by nitin mittal.
```

PHP

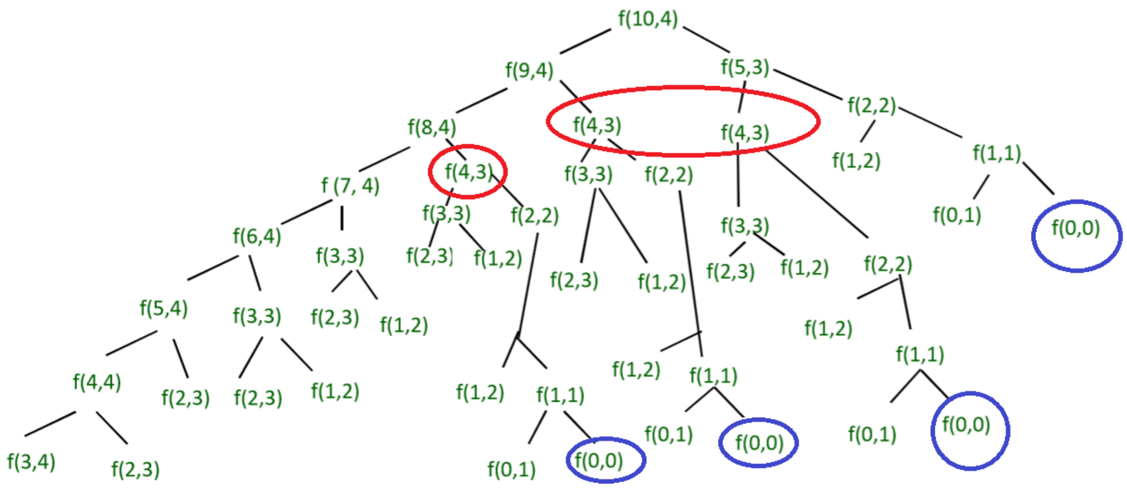
```
 <?php
// PHP program to count total
// number of special sequences
// of length n where

// Recursive function to find
// the number of special sequences
function getTotalNumberOfSequences($m, $n)
{

    // A special sequence cannot
    // exist if length n is more
    // than the maximum value m.
    if ($m < $n)
        return 0;

    // If n is 0, found an empty
    // special sequence
    if ($n == 0)
        return 1;

    // There can be two possibilities :
```

```
    // (1) Reduce last element value
    // (2) Consider last element
    // as m and reduce number of terms
    return getTotalNumberOfSequences($m - 1, $n) +
           getTotalNumberOfSequences($m / 2, $n - 1);
}

    // Driver Code
    $m = 10;
    $n = 4;
    echo("Total number of possible sequences ");
    echo (getTotalNumberOfSequences($m, $n));

// This code is contributed by nitin mittal.
?>
```

Output:

```
Total number of possible sequences 4
```

Note that the above function computes the same sub problems again and again. Consider the following tree for f(10, 4).



Recursive Tree for m= 10 and N =4

We can solve this problem using dynamic programming.

C++

```
 // C program to count total number of special sequences
// of length N where
#include <stdio.h>

// DP based function to find the number of special
// sequences
int  getTotalNumberOfSequences(int m, int n)
{
        // define T and build in bottom manner to store
        // number of special sequences of length n and
        // maximum value m
        int T[m+1][n+1];
        for (int i=0; i<m+1; i++)
        {
            for (int j=0; j<n+1; j++)
            {
                // Base case : If length of sequence is 0
                // or maximum value is 0, there cannot
                // exist any special sequence
                if (i == 0 || j == 0)
                    T[i][j] = 0;

                // if length of sequence is more than
                // the maximum value, special sequence
                // cannot exist
                else if (i < j)
                    T[i][j] = 0;

                // If length of sequence is 1 then the
                // number of special sequences is equal
                // to the maximum value
                // For example with maximum value 2 and
                // length 1, there can be 2 special
                // sequences {1}, {2}
                else if (j == 1)
                    T[i][j] = i;

                // otherwise calculate
                else
                    T[i][j] = T[i-1][j] + T[i/2][j-1];
            }
        }
        return T[m][n];
}

// Driver Code
int main()
{
    int m = 10;
    int n = 4;
    printf("Total number of possible sequences %d",
                   getTotalNumberOfSequences(m, n));
    return 0;
}
```

Java

```
 // Efficient java program to count total number
// of special sequences of length n where
class Sequences
{
    // DP based function to find the number of special
    // sequences
    static int  getTotalNumberOfSequences(int m, int n)
    {
            // define T and build in bottom manner to store
            // number of special sequences of length n and
            // maximum value m
            int T[][]=new int[m+1][n+1];
            for (int i=0; i<m+1; i++)
            {
                for (int j=0; j<n+1; j++)
                {
                    // Base case : If length of sequence is 0
                    // or maximum value is 0, there cannot
                    // exist any special sequence
                    if (i == 0 || j == 0)
                        T[i][j] = 0;

                    // if length of sequence is more than
                    // the maximum value, special sequence
                    // cannot exist
                    else if (i < j)
                        T[i][j] = 0;

                    // If length of sequence is 1 then the
                    // number of special sequences is equal
                    // to the maximum value
                    // For example with maximum value 2 and
                    // length 1, there can be 2 special
                    // sequences {1}, {2}
                    else if (j == 1)
                        T[i][j] = i;

                    // otherwise calculate
                    else
```

```
                        T[i][j] = T[i-1][j] + T[i/2][j-1];
                }
            }
            return T[m][n];
    }

    // main function
    public static void main (String[] args)
    {
        int m = 10;
        int n = 4;
        System.out.println("Total number of possible sequences "+
                        getTotalNumberOfSequences(m, n));
    }
}
```

C#

```
 // Efficient C# program to count total number
// of special sequences of length n where
using System;
class Sequences {

    // DP based function to find
    // the number of special
    // sequences
    static int getTotalNumberOfSequences(int m, int n)
    {

            // define T and build in
            // bottom manner to store
            // number of special sequences
            // of length n and maximum value m
            int [,]T=new int[m + 1, n + 1];

            for (int i = 0; i < m + 1; i++)
            {
                for (int j = 0; j < n + 1; j++)
                {

                    // Base case : If length
                    // of sequence is 0
                    // or maximum value is
                    // 0, there cannot
                    // exist any special
                    // sequence
                    if (i == 0 || j == 0)
                        T[i, j] = 0;
                    // if length of sequence
                    // is more than the maximum
                    // value, special sequence
                    // cannot exist
                    else if (i < j)
                        T[i,j] = 0;

                    // If length of sequence is 1 then the
                    // number of special sequences is equal
                    // to the maximum value
                    // For example with maximum value 2 and
                    // length 1, there can be 2 special
                    // sequences {1}, {2}
                    else if (j == 1)
                        T[i,j] = i;

                    // otherwise calculate
                    else
                        T[i,j] = T[i - 1, j] + T[i / 2, j - 1];
                }
            }
            return T[m,n];
    }

    // Driver Code
    public static void Main ()
    {
        int m = 10;
        int n = 4;
        Console.WriteLine("Total number of possible sequences "+
                                getTotalNumberOfSequences(m, n));
    }
}

// This code is contributed by anuj_67.
```

PHP

```
 <?php
// PHP program to count total
// number of special sequences
// of length N where

// DP based function to find
// the number of special
// sequences
function getTotalNumberOfSequences($m, $n)
```

```
{

        // define T and build in bottom
        // manner to store number of
        // special sequences of length
        // n and maximum value m
        $T = array(array());

        for ($i = 0; $i < $m + 1; $i++)
        {
            for ($j = 0; $j < $n + 1; $j++)
            {

                // Base case : If length of
                // sequence is 0 or maximum
                // value is 0, there cannot
                // exist any special sequence
                if ($i == 0 or $j == 0)
                    $T[$i][$j] = 0;

                // if length of sequence is
                // more than the maximum value,
                // special sequence cannot exist
                else if ($i < $j)
                    $T[$i][$j] = 0;

                // If length of sequence is
                // 1 then the number of
                // special sequences is equal
                // to the maximum value
                // For example with maximum
                // value 2 and length 1, there
                // can be 2 special sequences
                // {1}, {2}
                else if ($j == 1)
                    $T[$i][$j] = $i;

                // otherwise calculate
                else
                    $T[$i][$j] = $T[$i - 1][$j] +
                                 $T[$i / 2][$j - 1];
            }
        }
        return $T[$m][$n];
}

    // Driver Code
    $m = 10;
    $n = 4;
    echo "Total number of possible sequences ",
            getTotalNumberOfSequences($m, $n);

// This code is contributed by anuj_67.
?>
```

Output:

```
4
```

Time Complexity : O(m x n)
Auxiliary Space : O(m x n)

Improved By : nitin mittal, vt_m

Source

https://www.geeksforgeeks.org/sequences-given-length-every-element-equal-twice-previous/

Chapter 74

Shuffle 2n integers in format {a1, b1, a2, b2, a3, b3, ……, an, bn} without using extra space

Shuffle 2n integers in format {a1, b1, a2, b2, a3, b3, ……, an, bn} without using extra space - GeeksforGeeks

Given an array of **2n** elements in the following format { a1, a2, a3, a4, ….., an, b1, b2, b3, b4, …., bn }. The task is shuffle the array to {a1, b1, a2, b2, a3, b3, ……, an, bn } without using extra space.

Examples:

```
Input : arr[] = { 1, 2, 9, 15 }
Output : 1 9 2 15

Input :  arr[] = { 1, 2, 3, 4, 5, 6 }
Output : 1 4 2 5 3 6
```

Method 1: Brute Force
A brute force solution involves two nested loops to rotate the elements in the second half of the array to the left. The first loop runs n times to cover all elements in the second half of the array. The second loop rotates the elements to the left. Note that the start index in the second loop depends on which element we are rotating and the end index depends on how many positions we need to move to the left.

Below is implementation of this approach:

C++

```
 // C++ Naive program to shuffle an array of size 2n
```

```
#include <bits/stdc++.h>
using namespace std;

// function to shuffle an array of size 2n
void shuffleArray(int a[], int n)
{
    // Rotate the element to the left
    for (int i = 0, q = 1, k = n; i < n; i++, k++, q++)
        for (int j = k; j > i + q; j--)
            swap(a[j-1], a[j]);
}

// Driven Program
int main()
{
    int a[] = { 1, 3, 5, 7, 2, 4, 6, 8 };
    int n = sizeof(a) / sizeof(a[0]);

    shuffleArray(a, n/2);

    for (int i = 0; i < n; i++)
        cout << a[i] << " ";

    return 0;
}
```

Java

```
 // Java Naive program to shuffle an array of size 2n

import java.util.Arrays;

public class GFG
{
    // method to shuffle an array of size 2n
    static void shuffleArray(int a[], int n)
    {
        // Rotate the element to the left
        for (int i = 0, q = 1, k = n; i < n; i++, k++, q++)
            for (int j = k; j > i + q; j--){
                // swap a[j-1], a[j]
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
    }

    // Driver Method
    public static void main(String[] args)
    {
        int a[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

        shuffleArray(a, a.length/2);

        System.out.println(Arrays.toString(a));
    }
}
```

Python3

```
 # Python3 Naive program to
# shuffle an array of size 2n

# Function to shuffle an array of size 2n
def shuffleArray(a, n):

    # Rotate the element to the left
    i, q, k = 0, 1, n
    while(i < n):
        j = k
        while(j > i + q):
            a[j - 1], a[j] = a[j], a[j - 1]
            j -= 1
        i += 1
        k += 1
        q += 1

# Driver Code
a = [1, 3, 5, 7, 2, 4, 6, 8]
n = len(a)
shuffleArray(a, int(n / 2))
for i in range(0, n):
    print(a[i], end = " ")

# This code is contributed by Smitha Dinesh Semwal.
```

Output:

```
1 2 3 4 5 6 7 8
```

Time Complexity: O(n^2)

Method 2: (Divide and Conquer)
The idea is to use Divide and Conquer Technique. Divide the given array into half (say

arr1[] and arr2[]) and swap second half element of arr1[] with first half element of arr2[]. Recursively do this for arr1 and arr2.

Let us explain with the help of an example.

1. Let the array be a1, a2, a3, a4, b1, b2, b3, b4
2. Split the array into two halves: a1, a2, a3, a4 : b1, b2, b3, b4
3. Exchange element around the center: exchange a3, a4 with b1, b2 correspondingly.
   you get: a1, a2, b1, b2, a3, a4, b3, b4
4. Recursively spilt a1, a2, b1, b2 into a1, a2 : b1, b2
   then split a3, a4, b3, b4 into a3, a4 : b3, b4.
5. Exchange elements around the center for each subarray we get:
   a1, b1, a2, b2 and a3, b3, a4, b4.

Note: This solution only handles the case when n = 2^i where i = 0, 1, 2, …etc.

Below is implementation of this approach:

C++

```
 // C++ Effective  program to shuffle an array of size 2n

#include <bits/stdc++.h>
using namespace std;

// function to shuffle an array of size 2n
void shufleArray(int a[], int f, int l)
{
    // If only 2 element, return
    if (l - f == 1)
        return;

    // finding mid to divide the array
    int mid = (f + l) / 2;

    // using temp for swapping first half of second array
    int temp = mid + 1;

    // mmid is use for swapping second half for first array
    int mmid = (f + mid) / 2;

    // Swapping the element
    for (int i = mmid + 1; i <= mid; i++)
        swap(a[i], a[temp++]);

    // Recursively doing for first half and second half
    shufleArray(a, f, mid);
    shufleArray(a, mid + 1, l);
}
```

```
// Driven Program
int main()
{
    int a[] = { 1, 3, 5, 7, 2, 4, 6, 8 };
    int n = sizeof(a) / sizeof(a[0]);

    shufleArray(a, 0, n - 1);

    for (int i = 0; i < n; i++)
        cout << a[i] << " ";

    return 0;
}
```

Java

```
 // Java Effective  program to shuffle an array of size 2n

import java.util.Arrays;

public class GFG
{
    // method to shuffle an array of size 2n
    static void shufleArray(int a[], int f, int l)
    {
        // If only 2 element, return
        if (l - f == 1)
            return;

        // finding mid to divide the array
        int mid = (f + l) / 2;

        // using temp for swapping first half of second array
        int temp = mid + 1;

        // mmid is use for swapping second half for first array
        int mmid = (f + mid) / 2;

        // Swapping the element
        for (int i = mmid + 1; i <= mid; i++)
        {
            // swap a[i], a[temp++]
            int temp1 = a[i];
            a[i] = a[temp];
            a[temp++] = temp1;
        }
        // Recursively doing for first half and second half
        shufleArray(a, f, mid);
        shufleArray(a, mid + 1, l);
    }

    // Driver Method
    public static void main(String[] args)
    {
        int a[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

        shufleArray(a, 0, a.length - 1);

        System.out.println(Arrays.toString(a));
    }
}
```

Python3

```
 # Python3 effective program to
# shuffle an array of size 2n

# Function to shuffle an array of size 2n
def shufleArray(a, f, l):

    # If only 2 element, return
    if (l - f == 1):
        return

    # Finding mid to divide the array
    mid = int((f + l) / 2)

    # Using temp for swapping first
    # half of second array
    temp = mid + 1

    # Mid is use for swapping second
    # half for first array
    mmid = int((f + mid) / 2)

    # Swapping the element
    for i in range(mmid + 1, mid + 1):
        (a[i], a[temp]) = (a[temp], a[i])
        temp += 1

    # Recursively doing for first
    # half and second half
    shufleArray(a, f, mid)
    shufleArray(a, mid + 1, l)
```

```
# Driver Code
a = [1, 3, 5, 7, 2, 4, 6, 8]
n = len(a)
shufleArray(a, 0, n - 1)

for i in range(0, n):
    print(a[i], end = " ")

# This code is contributed by Smitha Dinesh Semwal
```

Output:

```
1 2 3 4 5 6 7 8
```

Time Complexity: O(n log n)

Linear time solution

Source

https://www.geeksforgeeks.org/shuffle-2n-integers-format-a1-b1-a2-b2-a3-b3-bn-without-using-extra-space/

Chapter 75

Smallest number with given sum of digits and sum of square of digits

Smallest number with given sum of digits and sum of square of digits - GeeksforGeeks

Given sum of digits a and sum of square of digits b. Find the smallest number with given sum of digits and sum of the square of digits. The number should not contain more than 100 digits. Print -1 if no such number exists or if the number of digits is more than 100.

Examples:

> **Input :** a = 18, b = 162
> **Output :** 99
> **Explanation :** 99 is the smallest possible number whose sum of digits = 9 + 9 = 18 and sum of squares of digits is 9^2+9^2 = 162.
>
> **Input :** a = 12, b = 9
> **Output :** -1

Approach:
Since the smallest number can be of 100 digits, it cannot be stored. Hence the first step to solve it will be to find the minimum number of digits which can give us the sum of digits as a and sum of the square of digits as b. To find the minimum number of digits, we can use Dynamic Programming. DP[a][b] signifies the minimum number of digits in a number whose sum of the digits will be a and sum of the square of digits will be b. If there does not exist any such number then DP[a][b] will be -1.

Since the number cannot exceed 100 digits, DP array will be of size **101*8101**. Iterate for every digit, and try all possible combination of digits which gives us the sum of digits as a and sum of the square of digits as b. Store the minimum number of digits in DP[a][b] using the below recurrence relation:

DP[a][b] = min(minimumNumberOfDigits(a – i, b – (i * i)) + 1)
where 1<=i<=9

After getting the minimum number of digits, find the digits. To find the digits, check for all combinations and print those digits which satisfies the condition below:

1 + dp[a – i][b – i * i] == dp[a][b]
where 1<=i<=9

If the condition above is met by any of i, reduce a by i and b by i*i and break. Keep on repeating the above process to find all the digits till a is 0 and b is 0.

Below is the C++ implementation of above approach:

```
 // CPP program to find the Smallest number
// with given sum of digits and
// sum of square of digits
#include <bits/stdc++.h>
using namespace std;

int dp[901][8101];

// Top down dp to find minimum number of digits with
// given sum of dits a and sum of square of digits as b
int minimumNumberOfDigits(int a, int b)
{
    // Invalid condition
    if (a > b || a < 0 || b < 0 || a > 900 || b > 8100)
        return -1;

    // Number of digits satisfied
    if (a == 0 && b == 0)
        return 0;

    // Memoization
    if (dp[a][b] != -1)
        return dp[a][b];

    // Intialize ans as maximum as we have to find the
    // minimum number of digits
    int ans = 101;

    // Check for all possible combinations of digits
    for (int i = 9; i >= 1; i--) {

        // recurrence call
        int k = minimumNumberOfDigits(a - i, b - (i * i));
        // If the combination of digits cannot give sum as a
        // and sum of square of digits as b
        if (k != -1)
            ans = min(ans, k + 1);
    }

    // Returns the minimum number of digits
    return dp[a][b] = ans;
}

// Function to print the digits that gives
// sum as a and sum of square of digits as b
void printSmallestNumber(int a,int b)
{

    // initialize the dp array as -1
    memset(dp, -1, sizeof(dp));

    // base condition
    dp[0][0] = 0;

    // function call to get the minimum number of digits
    int k = minimumNumberOfDigits(a, b);

    // When there does not exists any number
    if (k == -1 || k > 100)
        cout << "-1";
    else {
        // Printing the digits from the most significant digit
        while (a > 0 && b > 0) {

            // Trying all combinations
            for (int i = 1; i <= 9; i++) {
                // checking conditions for minimum digits
                if (a >= i && b >= i * i &&
                    1 + dp[a - i][b - i * i] == dp[a][b]) {
                    cout << i;
                    a -= i;
                    b -= i * i;
                    break;
                }
            }
        }
    }
}

// Driver Code
```

```
int main()
{
    int a = 18, b = 162;
    // Function call to print the smallest number
    printSmallestNumber(a,b);
}
```

Output:

```
99
```

Time Complexity : O(900*8100*9)
Auxiliary Space : O(900*8100)

Note: Time complexity is in terms of numbers as we are trying all possible combinations of digits.

Source

https://www.geeksforgeeks.org/smallest-number-with-given-sum-of-digits-and-sum-of-square-of-digits/

Chapter 76

Square root of an integer

Square root of an integer - GeeksforGeeks

Given an integer x, find square root of it. If x is not a perfect square, then return floor(√x).

Examples :

```
Input: x = 4
Output: 2

Input: x = 11
Output: 3
```

There can be many ways to solve this problem. For example Babylonian Method is one way.

A **Simple Solution** to find floor of square root is to try all numbers starting from 1. For every tried number i, if i*i is smaller than x, then increment i. We stop when i*i becomes more than or equal to x. Below is the implementation of above idea.

C++

```
 // A C++ program to find floor(sqrt(x)
#include<bits/stdc++.h>
using namespace std;

// Returns floor of square root of x
int floorSqrt(int x)
{
    // Base cases
    if (x == 0 || x == 1)
    return x;
    // Staring from 1, try all numbers until
    // i*i is greater than or equal to x.
    int i = 1, result = 1;
    while (result <= x)
    {
      i++;
      result = i * i;
    }
    return i - 1;
}

// Driver program
int main()
{
    int x = 11;
    cout << floorSqrt(x) << endl;
    return 0;
}
```

Java

```
 // A Java program to find floor(sqrt(x))

class GFG {

    // Returns floor of square root of x
    static int floorSqrt(int x)
    {
        // Base cases
        if (x == 0 || x == 1)
            return x;

        // Staring from 1, try all numbers until
        // i*i is greater than or equal to x.
        int i = 1, result = 1;

        while (result <= x) {
            i++;
            result = i * i;
        }
        return i - 1;
    }

    // Driver program
    public static void main(String[] args)
    {
        int x = 11;
        System.out.print(floorSqrt(x));
    }
}

// This code is contributed by Smitha Dinesh Semwal.
```

Python3

```
 # Python3 program to find floor(sqrt(x)

# Returns floor of square root of x
def floorSqrt(x):

    # Base cases
    if (x == 0 or x == 1):
        return x

    # Staring from 1, try all numbers until
    # i*i is greater than or equal to x.
    i = 1; result = 1
    while (result <= x):

        i += 1
        result = i * i

    return i - 1

# Driver Code
x = 11
print(floorSqrt(x))

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 // A C# program to
// find floor(sqrt(x))
using System;

class GFG
{
    // Returns floor of
    // square root of x
    static int floorSqrt(int x)
    {
        // Base cases
        if (x == 0 || x == 1)
            return x;
        // Staring from 1, try all
        // numbers until i*i is
        // greater than or equal to x.
        int i = 1, result = 1;

        while (result <= x)
        {
            i++;
            result = i * i;
        }
        return i - 1;
    }

    // Driver Code
    static public void Main ()
    {
        int x = 11;
        Console.WriteLine(floorSqrt(x));
    }
}

// This code is contributed by ajit
```

PHP

```
 <?php
// A PHP program to find floor(sqrt(x)

// Returns floor of square root of x
function floorSqrt($x)
{
    // Base cases
    if ($x == 0 || $x == 1)
    return $x;

    // Staring from 1, try all
    // numbers until i*i is
    // greater than or equal to x.
    $i = 1;
    $result = 1;
    while ($result <= $x)
    {
        $i++;
        $result = $i * $i;
    }
    return $i - 1;
}
```

```
// Driver Code
$x = 11;
echo floorSqrt($x), "\n";

// This code is contributed by ajit
?>
```

Output :

```
3
```

Time complexity of the above solution is O(√ n). Thanks Fattepur Mahesh for suggesting this solution.

A **Better Solution** to do Binary Search.

```
Let  's' be the answer.  We know that 0 <=  s <= x.

Consider any random number r.

    If r*r = r

    If r*r > x, s < r.
```

Algorithm:
1) Start with 'start' = 0, end = 'x',
2) Do following while 'start' is smaller than or equal to 'end'.
    a) Compute 'mid' as (start + end)/2
    b) compare mid*mid with x.
    c) If x is equal to mid*mid, return mid.
    d) If x is greater, do binary search between mid+1 and end. In this case, we also update ans (Note that we need floor).
    e) If x is smaller, do binary search between start and mid

Below is the implementation of above idea.

C/C++

```
 // A C++ program to find floor(sqrt(x)
#include<bits/stdc++.h>
using namespace std;

// Returns floor of square root of x
int floorSqrt(int x)
{
    // Base cases
    if (x == 0 || x == 1)
       return x;

    // Do Binary Search for floor(sqrt(x))
    int start = 1, end = x, ans;
    while (start <= end)
    {
        int mid = (start + end) / 2;

        // If x is a perfect square
        if (mid*mid == x)
            return mid;

        // Since we need floor, we update answer when mid*mid is
        // smaller than x, and move closer to sqrt(x)
        if (mid*mid < x)
        {
            start = mid + 1;
            ans = mid;
        }
        else // If mid*mid is greater than x
            end = mid;
    }
    return ans;
}

// Driver program
int main()
{
    int x = 11;
    cout << floorSqrt(x) << endl;
    return 0;
}
```

Java

```
 // A Java program to find floor(sqrt(x)
public class Test
{
    public static int floorSqrt(int x)
    {
        // Base Cases
        if (x == 0 || x == 1)
            return x;

        // Do Binary Search for floor(sqrt(x))
        int start = 1, end = x, ans=0;
        while (start <= end)
        {
            int mid = (start + end) / 2;

            // If x is a perfect square
            if (mid*mid == x)
                return mid;

            // Since we need floor, we update answer when mid*mid is
            // smaller than x, and move closer to sqrt(x)
            if (mid*mid < x)
            {
                start = mid + 1;
                ans = mid;
            }
            else   // If mid*mid is greater than x
                end = mid;
        }
        return ans;
    }

    // Driver Method
    public static void main(String args[])
    {
        int x = 11;
        System.out.println(floorSqrt(x));
    }
}
// Contributed by InnerPeace
```

Python3

```
 # Python 3 program to find floor(sqrt(x)

# Returns floor of square root of x
def floorSqrt(x) :

    # Base cases
    if (x == 0 or x == 1) :
        return x

    # Do Binary Search for floor(sqrt(x))
    start = 1
    end = x
    while (start <= end) :
        mid = (start + end) // 2

        # If x is a perfect square
        if (mid*mid == x) :
            return mid

        # Since we need floor, we update
        # answer when mid*mid is smaller
        # than x, and move closer to sqrt(x)
        if (mid * mid < x) :
            start = mid + 1
            ans = mid

        else :

            # If mid*mid is greater than x
            end = mid

    return ans

# driver code
x = 11
print(floorSqrt(x))

# This code is contributed by Nikita Tiwari.
```

C#

```
 // A C# program to
// find floor(sqrt(x)
using System;

class GFG
{
    public static int floorSqrt(int x)
    {
        // Base Cases
        if (x == 0 || x == 1)
            return x;

        // Do Binary Search
        // for floor(sqrt(x))
        int start = 1, end = x, ans = 0;
        while (start <= end)
        {
            int mid = (start + end) / 2;

            // If x is a
            // perfect square
            if (mid * mid == x)
                return mid;
            // Since we need floor, we
            // update answer when mid *
            // mid is smaller than x,
            // and move closer to sqrt(x)
            if (mid * mid < x)
            {
                start = mid + 1;
                ans = mid;
            }

            // If mid*mid is
            // greater than x
            else
                end = mid;
        }
        return ans;
    }

    // Driver Code
    static public void Main ()
    {
        int x = 11;
        Console.WriteLine(floorSqrt(x));
    }
}

// This code is Contributed by m_kit
```

PHP

```
 <?php
// A PHP program to find floor(sqrt(x)

// Returns floor of
// square root of x
function floorSqrt($x)
{
    // Base cases
    if ($x == 0 || $x == 1)
    return $x;

    // Do Binary Search
    // for floor(sqrt(x))
    $start = 1; $end = $x; $ans;
    while ($start <= $end)
    {
        $mid = ($start + $end) / 2;
        // If x is a perfect square
        if ($mid * $mid == $x)
            return $mid;

        // Since we need floor, we update
        // answer when mid*mid is  smaller
        // than x, and move closer to sqrt(x)
        if ($mid * $mid < $x)
        {
            $start = $mid + 1;
            $ans = $mid;
        }

        // If mid*mid is
        // greater than x
        else
            $end = $mid;
    }
    return $ans;
}

// Driver Code
$x = 11;
echo floorSqrt($x), "\n";

// This code is contributed by ajit
?>
```

Output :

```
3
```

Time Complexity: O(Log x)

Thanks to Gaurav Ahirwar for suggesting above method.

Note: The Binary Search can be further optimized to start with 'start' = 0 and 'end' = x/2. Floor of square root of x cannot be more than x/2 when x > 1.

Thanks to **vinit** for suggesting above optimization.

Improved By : jit_t, FarhanSheik

Source

https://www.geeksforgeeks.org/square-root-of-an-integer/

Chapter 77

Sudo Placement | Placement Tour

Sudo Placement | Placement Tour - GeeksforGeeks

Given an array A of N positive integers and a budget B. Your task is to decide the maximum number of elements to be picked from the array such that the cumulative cost of all picked elements is less than or equal to budget B. Cost of picking the ith element is given by : A[i] + (i * K) where, K is a constant whose value is equal to the number of elements picked. The indexing(i) is 1 based. Print the maximum number and its respective cumulative cost.

Examples:

> **Input** : arr[] = { 2, 3, 5 }, B = 11
> **Output** : 2 11
> **Explanation** : Cost of picking maximum elements = {2 + (1 * 2) } + {3 + (2 * 2)} = 4 + 7 = 11 (which is equal to budget)
>
> **Input** : arr[] = { 1, 2, 5, 6, 3 }, B = 90
> **Output** : 4 54

Prerequisites : Binary Search

Approach: The idea here is to use binary search on all possible values of K i.e. the optimal number of elements to be picked. *Start* with zero as lower bound and *End* with total number of elements i.e. N as upper bound. Check if by setting K as current *Mid*, obtained cumulative cost is less than or equal to budget. If it satisfies the condition, then try to increase K by setting *Start* as *(Mid + 1)*, otherwise try to decrease K by setting *End* as *(Mid – 1)*.
Checking of the condition can be done in a brute force manner by simply modifying the array according to the given formula and adding the K (current number of elements to be picked) smallest modified values to get the cumulative cost.

Below is the implementation of above approach.

```
 // CPP Program to find the optimal number of
// elements such that the cumulative value
// should be less than given number
#include <bits/stdc++.h>

using namespace std;

// This function returns true if the value cumulative
// according to received integer K is less than budget
// B, otherwise returns false
bool canBeOptimalValue(int K, int arr[], int N, int B,
                       int& value)
{
    // Initialize a temporary array which stores
    // the cumulative value of the original array
    int tmp[N];

    for (int i = 0; i < N; i++)
        tmp[i] = (arr[i] + K * (i + 1));

    // Sort the array to find the smallest K values
    sort(tmp, tmp + N);

    value = 0;
    for (int i = 0; i < K; i++)
        value += tmp[i];

    // Check if the value is less than budget
    return value <= B;
}

// This function prints the optimal number of elements
// and respective cumulative value which is less than
// the given number
void findNoOfElementsandValue(int arr[], int N, int B)
{
    int start = 0; // Min Value or lower bound

    int end = N; // Max Value or upper bound

    // Initialize answer as zero as optimal value
    // may not exists
    int ans = 0;

    int cumulativeValue = 0;

    while (start <= end) {
        int mid = (start + end) / 2;
        // If the current Mid Value is an optimal
        // value, then try to maximize it
        if (canBeOptimalValue(mid, arr, N, B,
                              cumulativeValue)) {
            ans = mid;
            start = mid + 1;
        }
        else
            end = mid - 1;
    }
    // Call Again to set the corresponding cumulative
    // value for the optimal ans
    canBeOptimalValue(ans, arr, N, B, cumulativeValue);

    cout << ans << " " << cumulativeValue << endl;
}

// Driver Code
int main()
{
    int arr[] = { 1, 2, 5, 6, 3 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Budget
    int B = 90;
    findNoOfElementsandValue(arr, N, B);
    return 0;
}
```

Output:

```
4 54
```

Time Complexity: O(N * (log N)^2), where N is the number of elements in the given array.

Source

https://www.geeksforgeeks.org/sudo-placement-placement-tour/

Chapter 78

Sudo Placement | Range Queries

Sudo Placement | Range Queries - GeeksforGeeks

Given Q queries, with each query consisting of two integers L and R, the task is to find the total numbers between L and R (Both inclusive), having atmost three set bits in their binary representation.

Examples:

```
Input : Q = 2
        L = 3, R = 7
        L = 10, R = 16
Output : 5
         6
For the first query, valid numbers are 3, 4, 5, 6, and 7.
For the second query, valid numbers are 10, 11, 12, 13, 14 and 16.
```

Prerequisites : Bit Manipulation and Binary Search

Method 1 (Simple): A naive approach is to traverse all the numbers between L and R and find the number of set bits in each of those numbers. Increment a counter variable if a number does not have more than 3 set bits. Return answer as counter. **Note** : This approach is very inefficient since the numbers L and R may have large values (upto 10^{18}).

Method 2 (Efficient) : An efficient approach required here is precomputation. Since the values of L and R lie within the range [0, 10^{18}] (both inclusive), thus their binary representation can have at most 60 bits. Now, since the valid numbers are those having atmost 3 set bits, find them by generating all bit sequences of 60 bits with less than or equal to 3 set bits. This can be done by fixing, i^{th}, j^{th} and k^{th} bits for all i, j, k from (0, 60). Once, all the valid numbers are generated in sorted order, apply binary search to find the count of those numbers that lie within the given range.

Below is the implementation of above approach.

C++

```
 // CPP program to find the numbers
// having atmost 3 set bits within
// a given range
#include <bits/stdc++.h>

using namespace std;

#define LL long long int

// This function prints the required answer for each query
void answerQueries(LL Q, vector<pair<LL, LL> > query)
{
    // Set of Numbers having at most 3 set bits
    // arranged in non-descending order
    set<LL> s;

    // 0 set bits
    s.insert(0);

    // Iterate over all possible combinations of
    // i, j and k for 60 bits
    for (int i = 0; i <= 60; i++) {
        for (int j = i; j <= 60; j++) {
            for (int k = j; k <= 60; k++) {
                // 1 set bit
                if (j == i && i == k)
                    s.insert(1LL << i);

                // 2 set bits
                else if (j == k && i != j) {
                    LL x = (1LL << i) + (1LL << j);
                    s.insert(x);
                }
                else if (i == j && i != k) {
                    LL x = (1LL << i) + (1LL << k);
                    s.insert(x);
                }
                else if (i == k && i != j) {
                    LL x = (1LL << k) + (1LL << j);
                    s.insert(x);
                }

                // 3 set bits
                else {
                    LL x = (1LL << i) + (1LL << j) + (1LL << k);
                    s.insert(x);
                }
            }
        }
    }
    vector<LL> validNumbers;
    for (auto val : s)
        validNumbers.push_back(val);

    // Answer Queries by applying binary search
    for (int i = 0; i < Q; i++) {
        LL L = query[i].first;
        LL R = query[i].second;

        // Swap both the numbers if L is greater than R
        if (R < L)
            swap(L, R);
        if (L == 0)
            cout << (upper_bound(validNumbers.begin(), validNumbers.end(),
                    R) - validNumbers.begin()) << endl;
        else
            cout << (upper_bound(validNumbers.begin(), validNumbers.end(),
                   R) - upper_bound(validNumbers.begin(), validNumbers.end(),
                   L - 1)) << endl;
    }
}

// Driver Code
int main()
{
    // Number of Queries
    int Q = 2;
    vector<pair<LL, LL> > query(Q);
    query[0].first = 3;
    query[0].second = 7;
    query[1].first = 10;
    query[1].second = 16;

    answerQueries(Q, query);
    return 0;
}
```

Java

```
 // Java program to find the numbers
// having atmost 3 set bits within
// a given range
```

```
import java.util.*;
import java.io.*;

public class RangeQueries {

    //Class to store the L and R range of a query
    static class Query {
        long L;
        long R;
    }

    //It returns index of first element which is grater than searched value
     //If searched element is bigger than any array element function
     // returns first index after last element.
    public static int upperBound(ArrayList<Long> validNumbers,
                                Long value)
    {
        int low = 0;
        int high = validNumbers.size()-1;

        while(low < high){
            int mid = (low + high)/2;
            if(value >= validNumbers.get(mid)){
                low = mid+1;
            } else {
                high = mid;
            }
        }
        return low;
    }

    public static void answerQueries(ArrayList<Query> queries){
        // Set of Numbers having at most 3 set bits
        // arranged in non-descending order
        Set<Long> allNum = new HashSet<>();

        //0 Set bits
        allNum.add(0L);

        //Iterate over all possible combinations of i, j, k for
        // 60 bits. And add all the numbers with 0, 1 or 2 set bits into
        // the set allNum.
        for(int i=0; i<=60; i++){
            for(int j=0; j<=60; j++){
                for(int k=0; k<=60; k++){

                    //For one set bit, check if i, j, k are equal
                    //if yes, then set that bit and add it to the set
                    if(i==j && j==k){
                        allNum.add(1L << i);
                    }

                    //For two set bits, two of the three variable i,j,k
                    //will be equal and the third will not be. Set both
                    //the bits where two variabls are equal and the bit
                    //which is not equal, and add it to the set
                    else if(i==j && j != k){
                        long toAdd = (1L << i) + (1L << k);
                        allNum.add(toAdd);
                    }
                    else if(i==k && k != j){
                        long toAdd = (1L << i) + (1L << j);
                        allNum.add(toAdd);
                    }
                    else if(j==k && k != i){
                        long toAdd = (1L << j) + (1L << i);
                        allNum.add(toAdd);
                    }

                    //Setting all the 3 bits
                    else {
                        long toAdd = (1L << i) + (1L << j) + (1L << k);
                        allNum.add(toAdd);
                    }

                }
            }
        }

        //Adding all the numbers to an array list so that it can be sorted
        ArrayList<Long> validNumbers = new ArrayList<>();
        for(Long num: allNum){
            validNumbers.add(num);
        }

        Collections.sort(validNumbers);

        //Answer queries by applying binary search
        for(int i=0; i<queries.size(); i++){
            long L = queries.get(i).L;
            long R = queries.get(i).R;

            //Swap L and R if R is smaller than L
            if(R < L){
                long temp = L;
                L = R;
```

```
                R = temp;
            }

            if(L == 0){
                int indxOfLastNum = upperBound(validNumbers, R);
                System.out.println(indxOfLastNum+1);
            }
            else {
                int indxOfFirstNum = upperBound(validNumbers, L);
                int indxOfLastNum = upperBound(validNumbers, R);
                System.out.println((indxOfLastNum - indxOfFirstNum +1));
            }

        }

    }

    public static void main(String[] args){
        int Q = 2;
        ArrayList<Query> queries = new ArrayList<>();

        Query q1 = new Query();
        q1.L = 3;
        q1.R = 7;

        Query q2 = new Query();
        q2.L = 10;
        q2.R = 16;

        queries.add(q1);
        queries.add(q2);

        answerQueries(queries);
    }

}
```

Time Complexity : O((Maximum Number of Bits)^3 + Q * logN), where Q is the number of queries and N is the size of set containing all valid numbers. l valid numbers.

Source

https://www.geeksforgeeks.org/sudo-placement-range-queries/

Chapter 79

The Skyline Problem using Divide and Conquer algorithm

The Skyline Problem using Divide and Conquer algorithm - GeeksforGeeks

Given n rectangular buildings in a 2-dimensional city, computes the skyline of these buildings, eliminating hidden lines. The main task is to view buildings from a side and remove all sections that are not visible.

All buildings share common bottom and every **building** is represented by triplet (left, ht, right)

'left': is x coordinated of left side (or wall).
'right': is x coordinate of right side
'ht': is height of building.

A **skyline** is a collection of rectangular strips. A rectangular **strip** is represented as a pair (left, ht) where left is x coordinate of left side of strip and ht is height of strip.

Examples:

```
Input: Array of buildings
       { (1,11,5), (2,6,7), (3,13,9), (12,7,16), (14,3,25),
         (19,18,22), (23,13,29), (24,4,28) }
Output: Skyline (an array of rectangular strips)
        A strip has x coordinate of left side and height
        (1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18),
        (22, 3), (25, 0)
Below image is for input 1 :


Consider following as another example when there is only one
building
Input:  {(1, 11, 5)}
```

```
Output: (1, 11), (5, 0)
```

A **Simple Solution** is to initialize skyline or result as empty, then one by one add buildings to skyline. A building is added by first finding the overlapping strip(s). If there are no overlapping strips, the new building adds new strip(s). If overlapping strip is found, then height of the existing strip may increase. Time complexity of this solution is $O(n^2)$

We can find Skyline in $\Theta(n\text{Log}n)$ time using **Divide and Conquer**. The idea is similar to Merge Sort, divide the given set of buildings in two subsets. Recursively construct skyline for two halves and finally merge the two skylines.

How to Merge two Skylines?
The idea is similar to merge of merge sort, start from first strips of two skylines, compare x coordinates. Pick the strip with smaller x coordinate and add it to result. The height of added strip is considered as maximum of current heights from skyline1 and skyline2.
Example to show working of merge:

```
  Height of new Strip is always obtained by takin maximum of following
     (a) Current height from skyline1, say 'h1'.
     (b) Current height from skyline2, say 'h2'
  h1 and h2 are initialized as 0. h1 is updated when a strip from
  SkyLine1 is added to result and h2 is updated when a strip from
  SkyLine2 is added.

  Skyline1 = {(1, 11),  (3, 13),  (9, 0),  (12, 7),  (16, 0)}
  Skyline2 = {(14, 3),  (19, 18), (22, 3), (23, 13),  (29, 0)}
  Result = {}
  h1 = 0, h2 = 0

  Compare (1, 11) and (14, 3).  Since first strip has smaller left x,
  add it to result and increment index for Skyline1.
  h1 = 11, New Height  = max(11, 0)
  Result =   {(1, 11)}

  Compare (3, 13) and (14, 3). Since first strip has smaller left x,
  add it to result and increment index for Skyline1
  h1 = 13, New Height =  max(13, 0)
  Result =  {(1, 11), (3, 13)}

  Similarly (9, 0) and (12, 7) are added.
  h1 = 7, New Height =  max(7, 0) = 7
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7)}

  Compare (16, 0) and (14, 3). Since second strip has smaller left x,
  it is added to result.
  h2 = 3, New Height =  max(7, 3) = 7
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7)}
  Compare (16, 0) and (19, 18). Since first strip has smaller left x,
  it is added to result.
  h1 = 0, New Height =  max(0, 3) = 3
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 3), (16, 3)}

Since Skyline1 has no more items, all remaining items of Skyline2
are added
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 3), (16, 3),
              (19, 18), (22, 3), (23, 13), (29, 0)}

One observation about above output is, the strip (16, 3) is redundant
(There is already an strip of same height). We remove all redundant
strips.
  Result =   {(1, 11), (3, 13), (9, 0), (12, 7), (14, 3), (19, 18),
              (22, 3), (23, 13), (29, 0)}

In below code, redundancy is handled by not appending a strip if the
previous strip in result has same height.
```

Below is C++ implementation of above idea.

```
 // A divide and conquer based C++ program to find skyline of given
// buildings
#include<iostream>
using namespace std;

// A structure for building
struct Building
{
    int left;  // x coordinate of left side
    int ht;    // height
    int right; // x coordinate of right side
};

// A strip in skyline
class Strip
{
    int left;  // x coordinate of left side
    int ht; // height
public:
    Strip(int l=0, int h=0)
    {
        left = l;
        ht = h;
    }
    friend class SkyLine;
};
```

```
// Skyline:  To represent Output (An array of strips)
class SkyLine
{
    Strip *arr;   // Array of strips
    int capacity; // Capacity of strip array
    int n;   // Actual number of strips in array
public:
    ~SkyLine() {  delete[] arr;  }
    int count()  { return n;   }

    // A function to merge another skyline
    // to this skyline
    SkyLine* Merge(SkyLine *other);

    // Constructor
    SkyLine(int cap)
    {
        capacity = cap;
        arr = new Strip[cap];
        n = 0;
    }

    // Function to add a strip 'st' to array
    void append(Strip *st)
    {
        // Check for redundant strip, a strip is
        // redundant if it has same height or left as previous
        if (n>0 && arr[n-1].ht == st->ht)
            return;
        if (n>0 && arr[n-1].left == st->left)
        {
            arr[n-1].ht = max(arr[n-1].ht, st->ht);
            return;
        }

        arr[n] = *st;
        n++;
    }

    // A utility function to print all strips of
    // skyline
    void print()
    {
        for (int i=0; i<n; i++)
        {
            cout << " (" << arr[i].left << ", "
                 << arr[i].ht << "), ";
        }
    }
};

// This function returns skyline for a given array of buildings
// arr[l..h].  This function is similar to mergeSort().
SkyLine *findSkyline(Building arr[], int l, int h)
{
    if (l == h)
    {
        SkyLine *res = new SkyLine(2);
        res->append(new Strip(arr[l].left, arr[l].ht));
        res->append(new Strip(arr[l].right, 0));
        return res;
    }

    int mid = (l + h)/2;

    // Recur for left and right halves and merge the two results
    SkyLine *sl = findSkyline(arr, l, mid);
    SkyLine *sr = findSkyline(arr, mid+1, h);
    SkyLine *res = sl->Merge(sr);

    // To avoid memory leak
    delete sl;
    delete sr;

    // Return merged skyline
    return res;
}

// Similar to merge() in MergeSort
// This function merges another skyline 'other' to the skyline
// for which it is called.  The function returns pointer to
// the resultant skyline
SkyLine *SkyLine::Merge(SkyLine *other)
{
    // Create a resultant skyline with capacity as sum of two
    // skylines
    SkyLine *res = new SkyLine(this->n + other->n);

    // To store current heights of two skylines
    int h1 = 0, h2 = 0;

    // Indexes of strips in two skylines
    int i = 0, j = 0;
    while (i < this->n && j < other->n)
    {
        // Compare x coordinates of left sides of two
        // skylines and put the smaller one in result
        if (this->arr[i].left < other->arr[j].left)
        {
            int x1 = this->arr[i].left;
            h1 = this->arr[i].ht;

            // Choose height as max of two heights
            int maxh = max(h1, h2);

            res->append(new Strip(x1, maxh));
            i++;
        }
        else
        {
            int x2 = other->arr[j].left;
            h2 = other->arr[j].ht;
            int maxh = max(h1, h2);
            res->append(new Strip(x2, maxh));
            j++;
        }
    }

    // If there are strips left in this skyline or other
    // skyline
    while (i < this->n)
    {
        res->append(&arr[i]);
        i++;
    }
    while (j < other->n)
    {
        res->append(&other->arr[j]);
        j++;
    }
    return res;
}

// drive program
int main()
{
    Building arr[] = {{1, 11, 5}, {2, 6, 7}, {3, 13, 9},
                      {12, 7, 16}, {14, 3, 25}, {19, 18, 22},
                      {23, 13, 29}, {24, 4, 28}};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Find skyline for given buildings and print the skyline
    SkyLine *ptr = findSkyline(arr, 0, n-1);
    cout << " Skyline for given buildings is \n";
    ptr->print();
    return 0;
}
```

```
 Skyline for given buildings is
 (1, 11),  (3, 13),  (9, 0),  (12, 7),  (16, 3),  (19, 18),
 (22, 3),  (23, 13),  (29, 0),
```

Time complexity of above recursive implementation is same as Merge Sort.

T(n) = T(n/2) + Θ(n)

Solution of above recurrence is Θ(nLogn)

References:
http://faculty.kfupm.edu.sa/ics/darwish/stuff/ics353handouts/Ch4Ch5.pdf
www.cs.ucf.edu/~sarahb/COP3503/Lectures/DivideAndConquer.ppt

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

Source

https://www.geeksforgeeks.org/the-skyline-problem-using-divide-and-conquer-algorithm/

Chapter 80

The painter's partition problem

The painter's partition problem - GeeksforGeeks

We have to paint n boards of length {A1, A2…An}. There are k painters available and each takes 1 unit time to paint 1 unit of board. The problem is to find the minimum time to get this job done under the constraints that any painter will only paint continuous sections of boards, say board {2, 3, 4} or only board {1} or nothing but not board {2, 4, 5}.

Examples:

```
Input : k = 2, A = {10, 10, 10, 10}
Output : 20.
Here we can divide the boards into 2
equal sized partitions, so each painter
gets 20 units of board and the total
time taken is 20.

Input : k = 2, A = {10, 20, 30, 40}
Output : 60.
Here we can divide first 3 boards for
one painter and the last board for
second painter.
```

From the above examples, it is obvious that the strategy of dividing the boards into k equal partitions won't work for all the cases. We can observe that the problem can be **broken down** into: Given an array A of non-negative integers and a positive integer k, we have to divide A into k of fewer partitions such that the maximum sum of the elements in a partition, overall partitions is minimized. So for the second example above, possible **divisions** are:
* One partition: so time is 100.
* Two partitions: (10) & (20, 30, 40), so time is 90. Similarly we can put the first divider after 20 (=> time 70) or 30 (=> time 60); so this means the minimum time: (100, 90, 70, 60) is 60.

A **brute force** solution is to consider all possible set of contiguous partitions and calculate the maximum sum partition in each case and return the minimum of all these cases.

1) Optimal Substructure:
We can implement the naive solution using recursion with the following optimal substructure property:
Assuming that we already have k-1 partitions in place (using k-2 dividers), we now have to put the k-1 th divider to get k partitions.
How can we do this? We can put the k-1 th divider between the i th and i+1 th element where i = 1 to n. Please note that putting it before the first element is the same as putting it after the last element.

The total cost of this arrangement can be calculated as the **maximum** of the following:
a) The cost of the last partition: sum(Ai..An), where the k-1 th divider is
before element i.
b) The maximum cost of any partition already formed to the left of the k-1 th divider.

Here a) can be found out using a simple **helper function** to calculate sum
of elements between two indices in the array. How to find out b) ?
We can observe that b) actually is to place the k-2 separators as fairly as
possible, so it is a **subproblem** of the given problem. Thus we can write the optimal substructure property as the following recurrence relation:

$$T(n,k) = min\left\{ max_{i=1}^{n} \left\{ T(i,k-1), \sum_{j=i+1}^{n} A_j \right\} \right\}$$

The base case are:

$$T(1,k) = A1$$

$$T(n,1) = \sum_{i=1}^{n} A_i$$

Following is the implementation of the above recursive equation:

C++

```
 // CPP program for The painter's partition problem
#include <climits>
#include <iostream>
using namespace std;

// function to calculate sum between two indices
// in array
int sum(int arr[], int from, int to)
```

```
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// for n boards and k partitions
int partition(int arr[], int n, int k)
{
    // base cases
    if (k == 1) // one partition
        return sum(arr, 0, n - 1);
    if (n == 1)  // one board
        return arr[0];

    int best = INT_MAX;

    // find minimum of all possible maximum
    // k-1 partitions to the left of arr[i],
    // with i elements, put k-1 th divider
    // between arr[i-1] & arr[i] to get k-th
    // partition
    for (int i = 1; i <= n; i++)
        best = min(best, max(partition(arr, i, k - 1),
                                sum(arr, i, n - 1)));

    return best;
}

int main()
{
    int arr[] = { 10, 20, 60, 50, 30, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;
    cout << partition(arr, n, k) << endl;

    return 0;
}
```

Java

```
 // Java Program for The painter's partition problem
import java.util.*;
import java.io.*;

class GFG
{
```

```
// function to calculate sum between two indices
// in array
static int sum(int arr[], int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// for n boards and k partitions
static int partition(int arr[], int n, int k)
{
    // base cases
    if (k == 1) // one partition
        return sum(arr, 0, n - 1);
    if (n == 1)  // one board
        return arr[0];

    int best = Integer.MAX_VALUE;

    // find minimum of all possible maximum
    // k-1 partitions to the left of arr[i],
    // with i elements, put k-1 th divider
    // between arr[i-1] & arr[i] to get k-th
    // partition
    for (int i = 1; i <= n; i++)
        best = Math.min(best, Math.max(partition(arr, i, k - 1),
                                sum(arr, i, n - 1)));

    return best;
}

// Driver code
public static void main(String args[])
{
 int arr[] = { 10, 20, 60, 50, 30, 40 };

    // Calculate size of array.
    int n = arr.length;
        int k = 3;
 System.out.println(partition(arr, n, k));
}
}

// This code is contributed by Sahil_Bansall
```

C#

```
 // C# Program for The painter's partition problem
using System;

class GFG {

// function to calculate sum
// between two indices in array
static int sum(int []arr, int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// for n boards and k partitions
static int partition(int []arr, int n, int k)
{
    // base cases
    if (k == 1) // one partition
        return sum(arr, 0, n - 1);

    if (n == 1) // one board
        return arr[0];

    int best = int.MaxValue;

    // find minimum of all possible maximum
    // k-1 partitions to the left of arr[i],
    // with i elements, put k-1 th divider
    // between arr[i-1] & arr[i] to get k-th
    // partition
    for (int i = 1; i <= n; i++)
        best = Math.Min(best, Math.Max(partition(arr, i, k - 1),
                                       sum(arr, i, n - 1)));

    return best;
}

// Driver code
public static void Main()
{
    int []arr = {10, 20, 60, 50, 30, 40};

    // Calculate size of array.
    int n = arr.Length;
    int k = 3;
    // Function calling
    Console.WriteLine(partition(arr, n, k));
}
}

// This code is contributed by vt_m
```

PHP

```
 <?php
// PHP program for The
// painter's partition problem

// function to calculate sum
// between two indices in array
function sum($arr, $from, $to)
{
    $total = 0;
    for ($i = $from; $i <= $to; $i++)
        $total += $arr[$i];
    return $total;
}

// for n boards
// and k partitions
function partition($arr, $n, $k)
{
    // base cases
    if ($k == 1) // one partition
        return sum($arr, 0, $n - 1);
    if ($n == 1) // one board
        return $arr[0];

    $best = PHP_INT_MAX;

    // find minimum of all possible
    // maximum k-1 partitions to the
    // left of arr[i], with i elements,
    // put k-1 th divider between
    // arr[i-1] & arr[i] to get k-th
    // partition
    for ($i = 1; $i <= $n; $i++)
        $best = min($best,
                max(partition($arr, $i, $k - 1),
                          sum($arr, $i, $n - 1)));

    return $best;
}
```

```
// Driver Code
$arr = array(10, 20, 60,
             50, 30, 40);
$n = sizeof($arr);
$k = 3;
echo partition($arr, $n, $k), "\n";

// This code is contributed by ajit
?>
```

Output :

```
90
```

The **time complexity** of the above solution is exponential.

2) Overlapping subproblems:
Following is the partial recursion tree for T(4, 3) in above equation.

```
      T(4, 3)
     /    /    \ ..
T(1, 2)  T(2, 2) T(3, 2)
          /..      /..
      T(1, 1)    T(1, 1)
```

We can observe that many subproblems like T(1, 1) in the above problem are being solved again and again. Because of these two properties of this problem, we can solve it using **dynamic programming**, either by top down memoized method or bottom up tabular method. Following is the bottom up tabular implementation:

C++

```
 // A DP based CPP program for painter's partition problem
#include <climits>
#include <iostream>
using namespace std;

// function to calculate sum between two indices
// in array
int sum(int arr[], int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// bottom up tabular dp
int findMax(int arr[], int n, int k)
{
    // initialize table
    int dp[k + 1][n + 1] = { 0 };

    // base cases
    // k=1
    for (int i = 1; i <= n; i++)
        dp[1][i] = sum(arr, 0, i - 1);

    // n=1
    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];

    // 2 to k partitions
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {

            // track minimum
            int best = INT_MAX;

            // i-1 th separator before position arr[p=1..j]
            for (int p = 1; p <= j; p++)
                best = min(best, max(dp[i - 1][p],
                              sum(arr, p, j - 1)));

            dp[i][j] = best;
        }
    }

    // required
    return dp[k][n];
}

// driver function
int main()
{
    int arr[] = { 10, 20, 60, 50, 30, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;
    cout << findMax(arr, n, k) << endl;
    return 0;
}
```

Java

```
 // A DP based Java program for
// painter's partition problem
import java.util.*;
import java.io.*;

class GFG
{
// function to calculate sum between two indices
// in array
static int sum(int arr[], int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// bottom up tabular dp
static int findMax(int arr[], int n, int k)
{
    // initialize table
    int dp[][] = new int[k+1][n+1];

    // base cases
    // k=1
    for (int i = 1; i <= n; i++)
        dp[1][i] = sum(arr, 0, i - 1);

    // n=1
    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];

    // 2 to k partitions
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {

            // track minimum
            int best = Integer.MAX_VALUE;

            // i-1 th separator before position arr[p=1..j]
            for (int p = 1; p <= j; p++)
                best = Math.min(best, Math.max(dp[i - 1][p],
                              sum(arr, p, j - 1)));

            dp[i][j] = best;
        }
    }
    // required
    return dp[k][n];
}

// Driver code
public static void main(String args[])
{
 int arr[] = { 10, 20, 60, 50, 30, 40 };

    // Calculate size of array.
    int n = arr.length;
        int k = 3;
 System.out.println(findMax(arr, n, k));
}
}

// This code is contributed by Sahil_Bansall
```

C#

```
 // A DP based C# program for
// painter's partition problem
using System;

class GFG {

// function to calculate sum between
// two indices in array
static int sum(int []arr, int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// bottom up tabular dp
static int findMax(int []arr, int n, int k)
{
    // initialize table
    int [,]dp = new int[k+1,n+1];

    // base cases
    // k=1
    for (int i = 1; i <= n; i++)
        dp[1,i] = sum(arr, 0, i - 1);

    // n=1
    for (int i = 1; i <= k; i++)
        dp[i,1] = arr[0];

    // 2 to k partitions
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {

            // track minimum
            int best = int.MaxValue;

            // i-1 th separator before position arr[p=1..j]
            for (int p = 1; p <= j; p++)
                best = Math.Min(best, Math.Max(dp[i - 1,p],
                                      sum(arr, p, j - 1)));

            dp[i,j] = best;
        }
    }

    // required
    return dp[k,n];
}

// Driver code
public static void Main()
{
    int []arr = {10, 20, 60, 50, 30, 40};

    // Calculate size of array.
    int n = arr.Length;
    int k = 3;
    Console.WriteLine(findMax(arr, n, k));
}
}

// This code is contributed by vt_m
```

PHP

```
 <?php
// A DP based PHP program for
// painter's partition problem

// function to calculate sum
// between two indices in array
function sum($arr, $from, $to)
{
    $total = 0;
    for ($i = $from; $i <= $to; $i++)
        $total += $arr[$i];
    return $total;
}

// bottom up tabular dp
function findMax($arr, $n, $k)
{
    // initialize table
    $dp[$k + 1][$n + 1] = array( 0 );

    // base cases
    // k=1
    for ($i = 1; $i <= $n; $i++)
        $dp[1][$i] = sum($arr, 0,
                         $i - 1);

    // n=1
    for ($i = 1; $i <= $k; $i++)
        $dp[$i][1] = $arr[0];

    // 2 to k partitions
    for ($i = 2; $i <= $k; $i++)
    {
        // 2 to n boards
        for ($j = 2; $j <= $n; $j++)
        {

            // track minimum
            $best = PHP_INT_MAX;

            // i-1 th separator before
            // position arr[p=1..j]
            for ($p = 1; $p <= $j; $p++)
                $best = min($best, max($dp[$i - 1][$p],
                               sum($arr, $p, $j - 1)));

            $dp[$i][$j] = $best;
        }
    }

    // required
    return $dp[$k][$n];
}

// Driver Code
$arr = array (10, 20, 60,
              50, 30, 40 );
```

```
$n = sizeof($arr);
$k = 3;
echo findMax($arr, $n, $k) ,"\n";

// This code is contribted by m_kit
?>
```

Output:

```
90
```

Optimizations:

1) The **time complexity** of the above program is $O(k * N^3)$. It can be easily brought down to $O(k * N^2)$ by precomputing the cumulative sums in an array thus avoiding repeated calls to the sum function:

```
 int sum[n+1] = {0};

 // sum from 1 to i elements of arr
 for (int i = 1; i <= n; i++)
   sum[i] = sum[i-1] + arr[i-1];

 for (int i = 1; i <= n; i++)
   dp[1][i] = sum[i];

and using it to calculate the result as:
best = min(best, max(dp[i-1][p], sum[j] - sum[p]));
```

2) Though here we consider to divide A into k or fewer partitions, we can observe that the **optimal case** always occurs when we divide A into exactly k partitions. So we can use:

```
 for (int i = k-1; i <= n; i++)
    best = min(best, max( partition(arr, i, k-1),
                            sum(arr, i, n-1)));
```

and modify the other implementations accordingly.

Exercise:
1) Can you come up with a solution using binary search which runs in O(N lg N) time? Prerequisite for this: https://www.topcoder.com/community/data-science/data-science-tutorials/binary-search/

References:
https://articles.leetcode.com/the-painters-partition-problem/

Asked in: Google, CodeNation

Improved By : vt_m, jit_t

Source

https://www.geeksforgeeks.org/painters-partition-problem/

Chapter 81

The painter's partition problem | Set 2

The painter's partition problem | Set 2 - GeeksforGeeks

We have to paint n boards of length {A1, A2, .. An}. There are k painters available and each takes 1 unit time to paint 1 unit of board. The problem is to find the minimum time to get this job done under the constraints that any painter will only paint continuous sections of boards, say board {2, 3, 4} or only board {1} or nothing but not board {2, 4, 5}.

Examples :

```
Input : k = 2, A = {10, 10, 10, 10}
Output : 20.
Here we can divide the boards into 2
equal sized partitions, so each painter
gets 20 units of board and the total
time taken is 20.

Input : k = 2, A = {10, 20, 30, 40}
Output : 60.
Here we can divide first 3 boards for
one painter and the last board for
second painter.
```

In the previous post we discussed a dynamic programming based approach having time complexity of $O(k * N^2)$ and $O(k * N)$ extra space.
In this post we will look into a more efficient approach using binary search. We know that the invariant of binary search has two main parts:
* the target value would always be in the searching range.
* the searching range will decrease in each loop so that the termination can be reached.

We also know that the values in this range must be in sorted order. Here our target value is the maximum sum of a contiguous section in the optimal allocation of boards. Now how can we apply binary search for this? We can fix the possible low to high range for the target value and narrow down our search to get the optimal allocation.

We can see that the highest possible value in this range is the sum of all the elementsin the array and this happens when we allot 1 painter all the sections of the board. The lowest possible value of this range is the maximum value of the array max, as in this allocation we can allot max to one painter and divide the other sections such
that the cost of them is less than or equal to max and as close as possible to max. Now if we consider we use x painters in the above scenarios, it is obvious that as the value in the range increases, the value of x decreases and vice-versa. From this we can find the target value when x=k and use a helper function to find x, the minimum number of painters required when the maximum length of section a painter can paint is given.

C++

```
 // CPP program for painter's partition problem
#include <iostream>
using namespace std;

// return the maximum element from the array
int getMax(int arr[], int n)
{
    int max = INT_MIN;
    for (int i = 0; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// return the sum of the elements in the array
int getSum(int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    return total;
}

// find minimum required painters for given maxlen
// which is the maximum length a painter can paint
int numberOfPainters(int arr[], int n, int maxLen)
{
    int total = 0, numPainters = 1;

    for (int i = 0; i < n; i++) {
        total += arr[i];
        if (total > maxLen) {

            // for next count
            total = arr[i];
            numPainters++;
        }
    }

    return numPainters;
}

int partition(int arr[], int n, int k)
{
    int lo = getMax(arr, n);
    int hi = getSum(arr, n);

    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        int requiredPainters = numberOfPainters(arr, n, mid);

        // find better optimum in lower half
        // here mid is included because we
        // may not get anything better
        if (requiredPainters <= k)
            hi = mid;

        // find better optimum in upper half
        // here mid is excluded because it gives
        // required Painters > k, which is invalid
        else
            lo = mid + 1;
    }

    // required
    return lo;
}

// driver function
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;
    cout << partition(arr, n, k) << endl;
    return 0;
}
```

Java

```
 // Java Program for painter's partition problem
import java.util.*;
import java.io.*;

class GFG
{
// return the maximum element from the array
static int getMax(int arr[], int n)
{
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// return the sum of the elements in the array
static int getSum(int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    return total;
}

// find minimum required painters for given maxlen
// which is the maximum length a painter can paint
static int numberOfPainters(int arr[], int n, int maxLen)
{
    int total = 0, numPainters = 1;

    for (int i = 0; i < n; i++) {
        total += arr[i];

        if (total > maxLen) {

            // for next count
            total = arr[i];
            numPainters++;
        }
    }

    return numPainters;
}

static int partition(int arr[], int n, int k)
{
    int lo = getMax(arr, n);
    int hi = getSum(arr, n);

    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        int requiredPainters = numberOfPainters(arr, n, mid);

        // find better optimum in lower half
        // here mid is included because we
        // may not get anything better
        if (requiredPainters <= k)
            hi = mid;

        // find better optimum in upper half
        // here mid is excluded because it gives
        // required Painters > k, which is invalid
        else
            lo = mid + 1;
    }

    // required
    return lo;
}

// Driver code
public static void main(String args[])
{
 int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // Calculate size of array.
    int n = arr.length;
        int k = 3;
 System.out.println(partition(arr, n, k));
}
}

// This code is contributed by Sahil_Bansall
```

C#

```
 // C# Program for painter's
// partition problem
using System;

class GFG
{

// return the maximum
// element from the array
```

```
static int getMax(int []arr, int n)
{
    int max = int.MinValue;
    for (int i = 0; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// return the sum of the
// elements in the array
static int getSum(int []arr, int n)
{
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    return total;
}

// find minimum required
// painters for given
// maxlen which is the
// maximum length a painter
// can paint
static int numberOfPainters(int []arr,
                            int n, int maxLen)
{
    int total = 0, numPainters = 1;

    for (int i = 0; i < n; i++)
    {
        total += arr[i];

        if (total > maxLen)
        {

            // for next count
            total = arr[i];
            numPainters++;
        }
    }

    return numPainters;
}

static int partition(int []arr,
                     int n, int k)
{
    int lo = getMax(arr, n);
    int hi = getSum(arr, n);

    while (lo < hi)
    {
        int mid = lo + (hi - lo) / 2;
        int requiredPainters =
                       numberOfPainters(arr, n, mid);

        // find better optimum in lower
        // half here mid is included
        // because we may not get
        // anything better
        if (requiredPainters <= k)
            hi = mid;

        // find better optimum in upper
        // half here mid is excluded
        // because it gives required
        // Painters > k, which is invalid
        else
            lo = mid + 1;
    }

    // required
    return lo;
}

// Driver code
static public void Main ()
{
    int []arr = {1, 2, 3, 4, 5,
                 6, 7, 8, 9};

    // Calculate size of array.
    int n = arr.Length;
    int k = 3;
    Console.WriteLine(partition(arr, n, k));
}
}

// This code is contributed by ajit
```

PHP

```
 <?php
// PHP program for painter's
// partition problem
// return the maximum
// element from the array
function getMax($arr, $n)
{
    $max = PHP_INT_MIN;
    for ($i = 0; $i < $n; $i++)
        if ($arr[$i] > $max)
            $max = $arr[$i];
    return $max;
}

// return the sum of the
// elements in the array
function getSum($arr, $n)
{
    $total = 0;
    for ($i = 0; $i < $n; $i++)
        $total += $arr[$i];
    return $total;
}

// find minimum required painters
// for given maxlen which is the
// maximum length a painter can paint
function numberOfPainters($arr, $n,
                          $maxLen)
{
    $total = 0; $numPainters = 1;

    for ($i = 0; $i < $n; $i++)
    {
        $total += $arr[$i];

        if ($total > $maxLen)
        {

            // for next count
            $total = $arr[$i];
            $numPainters++;
        }
    }

    return $numPainters;
}

function partition($arr, $n, $k)
{
    $lo = getMax($arr, $n);
    $hi = getSum($arr, $n);

    while ($lo < $hi)
    {
        $mid = $lo + ($hi - $lo) / 2;
        $requiredPainters =
                    numberOfPainters($arr,
                                     $n, $mid);

        // find better optimum in
        // lower half here mid is
        // included because we may
        // not get anything better
        if ($requiredPainters <= $k)
            $hi = $mid;

        // find better optimum in
        // upper half here mid is
        // excluded because it
        // gives required Painters > k,
        // which is invalid
        else
            $lo = $mid + 1;
    }

    // required
    return floor($lo);
}

// Driver Code
$arr = array(1, 2, 3,
             4, 5, 6,
             7, 8, 9);
$n = sizeof($arr);
$k = 3;

echo partition($arr, $n, $k) , "\n";

// This code is contributed by ajit
?>
```

Output :

```
17
```

For better understanding, please trace the example given in the program in pen and paper.

The time complexity of the above approach is $O(N * \log(sum(arr[])))$.

References:
https://articles.leetcode.com/the-painters-partition-problem-part-ii/
https://www.topcoder.com/community/data-science/data-science-tutorials/binary-search/

Asked in: Google, Codenation.

Improved By : jit_t

Source

https://www.geeksforgeeks.org/painters-partition-problem-set-2/

Chapter 82

Tiling Problem using Divide and Conquer algorithm

Tiling Problem using Divide and Conquer algorithm - GeeksforGeeks

Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

.

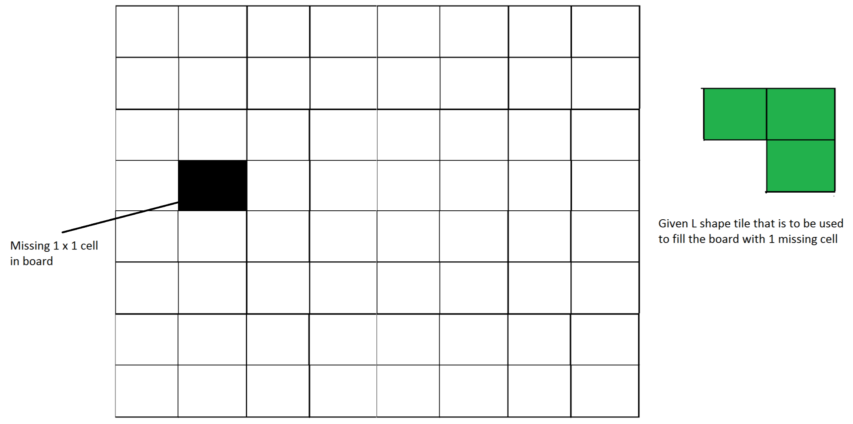

Missing 1 x 1 cell
in board
Given L shape tile that is to be used
to fill the board with 1 missing cell

Figure 1: An example input

This problem can be solved using Divide and Conquer. Below is the recursive algorithm.

```
// n is size of given square, p is location of missing cell
Tile(int n, Point p)

1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing
   but a tile and can be filled with a single tile.
```

```
2) Place a L shaped tile at the center such that it does not cover
   the n/2 * n/2 subsquare that has a missing square. Now all four
   subsquares of size n/2 x n/2 have a missing cell (a cell that doesn't
   need to be filled).  See figure 2 below.

3) Solve the problem recursively for following four. Let p1, p2, p3 and
   p4 be positions of the 4 missing cells in 4 squares.
   a) Tile(n/2, p1)
   b) Tile(n/2, p2)
   c) Tile(n/2, p3)
   d) Tile(n/2, p3)
```

The below diagrams show working of above algorithm

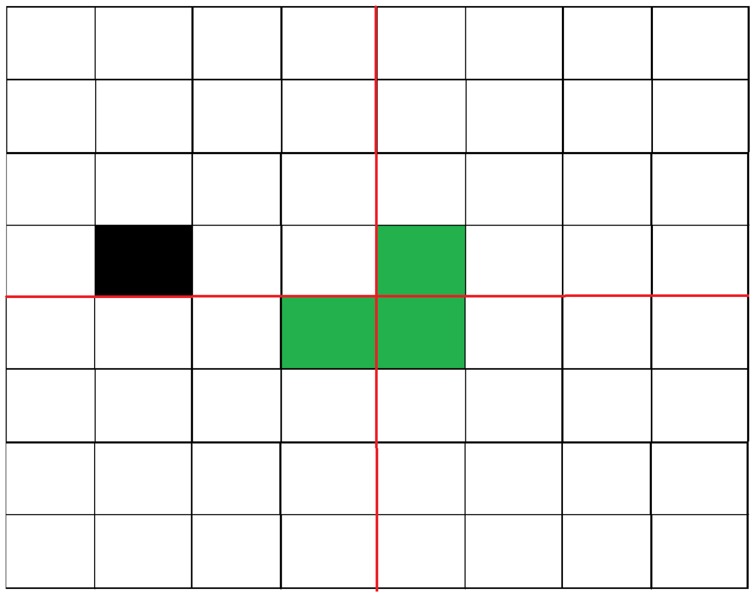


Figure 2: After placing first tile

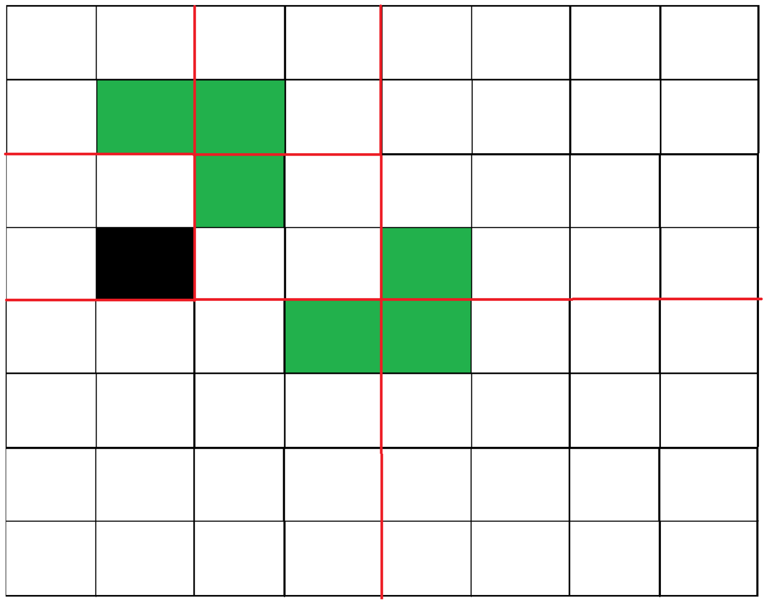


Figure 3: Recurring for first subsquare.

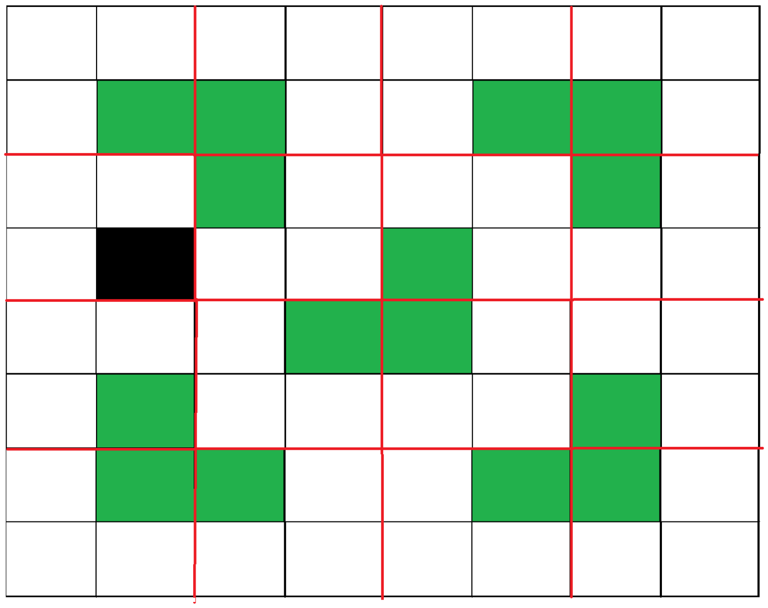


Figure 4: Shows first step in all four subsquares.

Time Complexity:

Recurrence relation for above recursive algorithm can be written as below. C is a constant.
T(n) = 4T(n/2) + C
The above recursion can be solved using Master Methodand time complexity is O(n^2)

How does this work?
The working of Divide and Conquer algorithm can be proved using Mathematical Induction.
Let the input square be of size 2^k x 2^k where k >=1.
Base Case: We know that the problem can be solved for k = 1. We have a 2 x 2 square with one cell missing.
Induction Hypothesis: Let the problem can be solved for k-1.
Now we need to prove to prove that the problem can be solved for k if it can be solved for k-1. For k, we put a L shaped tile in middle and we have four subsqures with dimension 2^{k-1} x 2^{k-1} as shown in figure 2 above. So if we can solve 4 subsquares, we can solve the complete square.

References:
http://www.comp.nus.edu.sg/~sanjay/cs3230/dandc.pdf

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

https://www.geeksforgeeks.org/tiling-problem-using-divide-and-conquer-algorithm/

Chapter 83

Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time)

Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time) - GeeksforGeeks

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;
}
```

Java

```
 // Java program for Binary Search
import java.util.*;

class Binary
{
    public static int f(int x)
    { return (x*x - 10*x - 20); }

    // Returns the value x where above
    // function f() becomes positive
    // first time.
    public static 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
    public static int binarySearch(int low, int high)
    {
        if (high >= low)
        {
            /* mid = (low + high)/2 */
            int mid = low + (high - low)/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 code
    public static void main(String[] args)
    {
        System.out.print ("The value n where f() "+
                          "becomes positive first is "+
                          findFirstPositive());
    }
}
```

```
// This code is contributed by rishabh_jain
```

Python3

```
 # Python3 program for Unbound Binary search.

# Let's take an example function as
# f(x) = x^2 - 10*x - 20
# Note that f(x) can be any monotonocally
# increasing function
def f(x):
    return (x * x - 10 * x - 20)

# Returns the value x where above function
# f() becomes positive first time.
def findFirstPositive() :

    # When first value itself is positive
    if (f(0) > 0):
        return 0

    # Find 'high' for binary search
    # by repeated doubling
    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
def binarySearch(low, high):
    if (high >= low) :

        # mid = (low + high)/2
        mid = low + (high - low)/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 and (mid == low or 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 Code
print ("The value n where f() becomes "+
      "positive first is ", findFirstPositive());

# This code is contributed by rishabh_jain
```

C#

```
 // C# program for Binary Search
using System;

class Binary
{
    public static int f(int x)
    {
        return (x*x - 10*x - 20);
    }

    // Returns the value x where above
    // function f() becomes positive
    // first time.
    public static 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
    public static int binarySearch(int low, int high)
    {
        if (high >= low)
        {
            /* mid = (low + high)/2 */
            int mid = low + (high - low)/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 code
    public static void Main()
    {
       Console.Write ("The value n where f() " +
                      "becomes positive first is " +
                       findFirstPositive());
    }
}

// This code is contributed by nitin mittal
```

PHP

```
 <?php
// PHP program for Binary Search

// Let's take an example function
// as f(x) = x^2 - 10*x - 20
// Note that f(x) can be any
// monotonocally increasing function
function f($x)
{
    return ($x * $x - 10 * $x - 20);
}

// Returns the value x where above
// function f() becomes positive
// first time.
function findFirstPositive()
{
    // When first value
    // itself is positive
    if (f(0) > 0)
        return 0;

    // Find 'high' for binary
    // search by repeated doubling
    $i = 1;
    while (f($i) <= 0)
        $i = $i * 2;

    // Call binary search
    return binarySearch(intval($i / 2), $i);
}

// Searches first positive value
// of f(i) where low <= i <= high
function binarySearch($low, $high)
{
    if ($high >= $low)
    {
        /* mid = (low + high)/2 */
        $mid = $low + intval(($high -
                              $low) / 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 Code
echo "The value n where f() becomes ".
                 "positive first is ".
                 findFirstPositive() ;

// This code is contributed by Sam007
?>
```

Output :

```
The value n where f() becomes positive first is 12
```

Related Article:
Exponential Search

Improved By : nitin mittal, Sam007

Source

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

Chapter 84

Write a program to calculate pow(x,n)

Write a program to calculate pow(x,n) - GeeksforGeeks

Given two integers x and n, write a function to compute x^n. We may assume that x and n are small and overflow doesn't happen.

Examples :

```
Input : x = 2, n = 3
Output : 8

Input : x = 7, n = 2
Output : 49
```

Below solution divides the problem into subproblems of size y/2 and call the subproblems recursively.

C

```
 #include<stdio.h>

/* Function to calculate x raised to the power y */
int power(int x, unsigned int y)
{
    if (y == 0)
        return 1;
    else if (y%2 == 0)
        return power(x, y/2)*power(x, y/2);
    else
        return x*power(x, y/2)*power(x, y/2);
}

/* Program to test function power */
int main()
{
    int x = 2;
    unsigned int y = 3;

    printf("%d", power(x, y));
    return 0;
}
```

Java

```
 class GFG {
    /* Function to calculate x raised to the power y */
    static int power(int x, int y)
    {
        if (y == 0)
            return 1;
        else if (y % 2 == 0)
            return power(x, y / 2) * power(x, y / 2);
        else
            return x * power(x, y / 2) * power(x, y / 2);
    }

    /* Program to test function power */
    public static void main(String[] args)
    {
        int x = 2;
        int y = 3;

        System.out.printf("%d", power(x, y));
    }
}

// This code is contributed by Smitha Dinesh Semwal
```

Python3

```
 # Python3 program to calculate pow(x,n)

# Function to calculate x
# raised to the power y
def power(x, y):
    if (y == 0): return 1
    elif (int(y % 2) == 0):
        return (power(x, int(y / 2)) *
               power(x, int(y / 2)))
    else:
        return (x * power(x, int(y / 2)) *
                   power(x, int(y / 2)))

# Driver Code
x = 2; y = 3
print(power(x, y))

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 using System;

public class GFG {

    // Function to calculate x raised to the power y
    static int power(int x, int y)
    {
        if (y == 0)
            return 1;
        else if (y % 2 == 0)
            return power(x, y / 2) * power(x, y / 2);
        else
            return x * power(x, y / 2) * power(x, y / 2);
    }

    // Program to test function power
    public static void Main()
    {
        int x = 2;
        int y = 3;

        Console.Write(power(x, y));
    }
}

// This code is contributed by shiv_bhakt.
```

PHP

```
 <?php
```

```
// Function to calculate x
// raised to the power y
function power($x, $y)
{
    if ($y == 0)
        return 1;
    else if ($y % 2 == 0)
        return power($x, (int)$y / 2) *
               power($x, (int)$y / 2);
    else
        return $x * power($x, (int)$y / 2) *
                    power($x, (int)$y / 2);
}

// Driver Code
$x = 2;
$y = 3;

echo power($x, $y);

// This code is contributed by ajit
?>
```

Output :

```
8
```

Time Complexity: O(n)
Space Complexity: O(1)
Algorithmic Paradigm: Divide and conquer.

Above function can be optimized to O(logn) by calculating power(x, y/2) only once and storing it.

```
 /* Function to calculate x raised to the power y in O(logn)*/
int power(int x, unsigned int y)
{
    int temp;
    if( y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

Time Complexity of optimized solution: O(logn)
Let us extend the pow function to work for negative y and float x.

C

```
 /* Extended version of power function that can work
 for float x and negative y*/
#include<stdio.h>

float power(float x, int y)
{
    float temp;
    if( y == 0)
       return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
    {
        if(y > 0)
            return x*temp*temp;
        else
            return (temp*temp)/x;
    }
}

/* Program to test function power */
int main()
{
    float x = 2;
    int y = -3;
    printf("%f", power(x, y));
    return 0;
}
```

Java

```
 /* Java code for extended version of power function
that can work for float x and negative y */
class GFG {

    static float power(float x, int y)
    {
        float temp;
        if( y == 0)
            return 1;
        temp = power(x, y/2);
        if (y%2 == 0)
            return temp*temp;
        else
        {
            if(y > 0)
                return x * temp * temp;
            else
                return (temp * temp) / x;
        }
    }

    /* Program to test function power */
    public static void main(String[] args)
    {
        float x = 2;
        int y = -3;
        System.out.printf("%f", power(x, y));
    }
}

// This code is contributed by  Smitha Dinesh Semwal.
```

Python3

```
 # Python3 code for extended version
# of power function that can work
# for float x and negative y

def power(x, y):

    if(y == 0): return 1
    temp = power(x, int(y / 2))

    if (y % 2 == 0):
        return temp * temp
    else:
        if(y > 0): return x * temp * temp
        else: return (temp * temp) / x

# Driver Code
x, y = 2, -3
print('%.6f' %(power(x, y)))

# This code is contributed by Smitha Dinesh Semwal.
```

C#

```
 // C# code for extended version of power function
```

```
// that can work for float x and negative y

using System;

public class GFG{

    static float power(float x, int y)
    {
        float temp;

        if( y == 0)
            return 1;
        temp = power(x, y/2);

        if (y % 2 == 0)
            return temp * temp;
        else
        {
            if(y > 0)
                return x * temp * temp;
            else
                return (temp * temp) / x;
        }
    }

    // Program to test function power
    public static void Main()
    {
        float x = 2;
        int y = -3;

        Console.Write(power(x, y));
    }
}

// This code is contributed by shiv_bhakt.
```

PHP

```
 <?php
// Extended version of power
// function that can work
// for float x and negative y

function power($x, $y)
{
    $temp;
    if( $y == 0)
    return 1;
    $temp = power($x, $y / 2);
    if ($y % 2 == 0)
        return $temp * $temp;
    else
    {
        if($y > 0)
            return $x *
                   $temp * $temp;
        else
            return ($temp *
                    $temp) / $x;
    }
}

// Driver Code
$x = 2;
$y = -3;
echo power($x, $y);

// This code is contributed by ajit
?>
```

Output :

```
0.125000
```

Write an iterative O(Log y) function for pow(x, y)
Modular Exponentiation (Power in Modular Arithmetic)

Improved By : jit_t

Source

https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/

Chapter 85

Write you own Power without using multiplication(*) and division(/) operators

Write you own Power without using multiplication(*) and division(/) operators - GeeksforGeeks

Method 1 (Using Nested Loops)

We can calculate power by using repeated addition.

For example to calculate 5^6.
1) First 5 times add 5, we get 25. (5^2)
2) Then 5 times add 25, we get 125. (5^3)
3) Then 5 time add 125, we get 625 (5^4)
4) Then 5 times add 625, we get 3125 (5^5)
5) Then 5 times add 3125, we get 15625 (5^6)

C

```
 #include<stdio.h>
/* Works only if a >= 0 and b >= 0  */
int pow(int a, int b)
{
  if (b == 0)
    return 1;
  int answer = a;
  int increment = a;
  int i, j;
  for(i = 1; i < b; i++)
  {
     for(j = 1; j < a; j++)
     {
        answer += increment;
     }
     increment = answer;
  }
  return answer;
}

/* driver program to test above function */
int main()
{
  printf("\n %d", pow(5, 3));
  getchar();
  return 0;
}
```

Java

```
 import java.io.*;

class GFG {

    /* Works only if a >= 0 and b >= 0 */
    static int pow(int a, int b)
    {
        if (b == 0)
            return 1;

        int answer = a;
        int increment = a;
        int i, j;

        for (i = 1; i < b; i++) {
            for (j = 1; j < a; j++) {
                answer += increment;
            }
            increment = answer;
        }

        return answer;
    }

    // driver program to test above function
    public static void main(String[] args)
    {
        System.out.println(pow(5, 3));
    }
}
```

```
// This code is contributed by vt_m.
```

Python

```
 # Python 3 code for power
# function

# Works only if a >= 0 and b >= 0
def pow(a,b):
    if(b==0):
        return 1

    answer=a
    increment=a

    for i in range(1,b):
        for j in range (1,a):
            answer+=increment
        increment=answer
    return answer

# drive code
print(pow(5,3))

# this code is contributed
# by Sam007
```

C#

```
 using System;

class GFG
{
    /* Works only if a >= 0 and b >= 0 */
    static int pow(int a, int b)
    {
        if (b == 0)
            return 1;

        int answer = a;
        int increment = a;
        int i, j;

        for (i = 1; i < b; i++) {
            for (j = 1; j < a; j++) {
                answer += increment;
            }
            increment = answer;
        }

        return answer;
    }

    // driver program to test
    // above function
    public static void Main()
    {
        Console.Write(pow(5, 3));
    }
}

// This code is contributed by Sam007
```

PHP

```
 <?php

// Works only if a >= 0
// and b >= 0
function poww($a, $b)
{
    if ($b == 0)
        return 1;
    $answer = $a;
    $increment = $a;
    $i;
    $j;
    for($i = 1; $i < $b; $i++)
    {
        for($j = 1; $j < $a; $j++)
        {
            $answer += $increment;
        }
        $increment = $answer;
    }
    return $answer;
}

    // Driver Code
    echo( poww(5, 3));

// This code is contributed by nitin mittal.
?>
```

Ouput :

```
125
```

Method 2 (Using Recursion)
Recursively add a to get the multiplication of two numbers. And recursively multiply to get *a* raise to the power *b*.

C

```
 #include<stdio.h>
/* A recursive function to get a^b
  Works only if a >= 0 and b >= 0  */
int pow(int a, int b)
{
   if(b)
     return multiply(a, pow(a, b-1));
   else
    return 1;
}

/* A recursive function to get x*y */
int multiply(int x, int y)
{
   if(y)
     return (x + multiply(x, y-1));
   else
     return 0;
}

/* driver program to test above functions */
int main()
{
  printf("\n %d", pow(5, 3));
  getchar();
  return 0;
}
```

Java

```
 import java.io.*;

class GFG {

    /* A recursive function to get a^b
    Works only if a >= 0 and b >= 0 */
    static int pow(int a, int b)
    {

        if (b > 0)
            return multiply(a, pow(a, b - 1));
        else
            return 1;
    }

    /* A recursive function to get x*y */
    static int multiply(int x, int y)
    {

        if (y > 0)
            return (x + multiply(x, y - 1));
        else
            return 0;
    }

    /* driver program to test above functions */
    public static void main(String[] args)
    {
        System.out.println(pow(5, 3));
    }
}

// This code is contributed by vt_m.
```

Python

```
 def pow(a,b):

    if(b):
        return multiply(a, pow(a, b-1));
    else:
        return 1;

# A recursive function to get x*y *
def multiply(x, y):

    if (y):
        return (x + multiply(x, y-1));
    else:
        return 0;

# driver program to test above functions *
print(pow(5, 3));
```

```
# This code is contributed
# by Sam007
```

C#

```
 using System;

class GFG
{
    /* A recursive function to get a^b
    Works only if a >= 0 and b >= 0 */
    static int pow(int a, int b)
    {

        if (b > 0)
            return multiply(a, pow(a, b - 1));
        else
            return 1;
    }

    /* A recursive function to get x*y */
    static int multiply(int x, int y)
    {

        if (y > 0)
            return (x + multiply(x, y - 1));
        else
            return 0;
    }

    /* driver program to test above functions */
    public static void Main()
    {
        Console.Write(pow(5, 3));
    }
}

// This code is contributed by Sam007
```

PHP

```
 <?php

/* A recursive function to get a^b
   Works only if a >= 0 and b >= 0 */
```

```
function p_ow( $a, $b)
{
    if($b)
        return multiply($a,
          p_ow($a, $b - 1));
    else
        return 1;
}

/* A recursive function
   to get x*y */
function multiply($x, $y)
{
    if($y)
        return ($x + multiply($x, $y - 1));
    else
        return 0;
}

// Driver Code
echo pow(5, 3);

// This code is contributed by anuj_67.
?>
```

Ouput :

```
125
```

Improved By : nitin mittal, vt_m

Source

https://www.geeksforgeeks.org/write-you-own-power-without-using-multiplication-and-division/