**Searching & Sorting**

# Find first and last positions of an element in a sorted array

Given a sorted array **arr** containing **n** elements with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element **x** in the given array.

**Example 1:**

**Input:**

n=9, x=5

arr[] = { 1, 3, 5, 5, 5, 5, 67, 123, 125 }

**Output:** 2 5

**Explanation**: First occurrence of 5 is at index 2 and last

  occurrence of 5 is at index 5.

**Example 2:**

**Input:**

n=9, x=7

arr[] = { 1, 3, 5, 5, 5, 5, 7, 123, 125 }

**Output:** 6 6

**Your Task:**  
Since, this is a function problem. You don't need to take any input, as it is already accomplished by the driver code. You just need to complete the function **find**() that takes **array arr, integer n and integer x** as parameters and returns the required answer.  
**Note:** If the number **x** is not found in the array just return both index as -1.

**Expected Time Complexity:** O(logN)  
**Expected Auxiliary Space:** O(1).

**Constraints:**  
1 ≤ N ≤ 107

## Solution:

**Using linear search:**

vector<int> find(int arr[], int n , int x )

{

vector<int> result;

result.push\_back(-1);

result.push\_back(-1);

for(int i=0;i<n;i++){

if(arr[i]==x){

if(result[0]<0)

result[0] = i; //first occurence

result[1] = i; //last occurence

}

}

return result;

}

**Time Complexity:** O(n) because only one traversal of array is needed.

**Space Complexity:** O(1).

**Using lower bound and upper bound in C++:**

vector<int> find(int arr[], int n , int x )

{

vector<int> res;

int ind = lower\_bound(arr,arr+n,x)-arr;

if(arr[ind]!=x){

res.push\_back(-1);

res.push\_back(-1);

return res;

}

res.push\_back(ind);

res.push\_back(upper\_bound(arr,arr+n,x)-arr-1);

return res;

}

**Time Complexity:** O(logn) because TC of lower\_bound and upper\_bound function of C++ is logn as they use modification of binary search.

**Space Complexity:** O(1).

**Using binary search:**

vector<int> find(int arr[], int n , int x )

{

vector<int> res;

int i=0, j=n-1, first=n, last;

while(i<=j){

int m = (i+j)/2;

if(arr[m]<x)

i = m+1;

else if(arr[m]>x)

j = m-1;

else{

first = m;

j = m-1;

}

}

if(first==n){

res.push\_back(-1);

res.push\_back(-1);

return res;

}

res.push\_back(first);

i=0,j=n-1;

while(i<=j){

int m = (i+j)/2;

if(arr[m]<x)

i = m+1;

else if(arr[m]>x)

j = m-1;

else{

last = m;

i = m+1;

}

}

res.push\_back(last);

return res;

}

**Time Complexity:** O(logn)

**Space Complexity:** O(1)

# Find a Fixed Point (Value equal to index) in a given array

Given an array **Arr** of **N** positive integers. Your task is to find the elements whose value is equal to that of its index value ( Consider 1-based indexing ).

**Example 1:**

**Input:**

N = 5

Arr[] = {15, 2, 45, 12, 7}

**Output:** 2

**Explanation:** Only Arr[2] = 2 exists here.

**Example 2:**

**Input:**

N = 1

Arr[] = {1}

**Output:** 1

**Explanation:** Here Arr[1] = 1 exists.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **valueEqualToIndex()** which takes the array of integers **arr[]**and**n**as parameters and returns an array of indices where the given conditions are satified. When there is not such element exists then return an empty array of length 0.  
  
**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)  
**Note**: There can be more than one element in the array which have same value as their index. You need to include every such element's index. Follows 1-based indexing of the array.

**Constraints:**  
1 ≤ N ≤ 105  
1 ≤ Arr[i] ≤ 106

## Solution:

Algorithm:

1. Create an auxiliary vector res, which stores all the elements

2. The idea is to traverse the array linearly and for each index i,

we need to check whether arr[i] = i + 1.

3. If the above condition is true, push the elements into the vector.

4. At the end, return the resultant vector.

vector<int> valueEqualToIndex(int arr[], int n) {

vector<int> result;

for(int i=0;i<n;i++){

if(arr[i]==i+1)

result.push\_back(arr[i]);

}

return result;

}

**Complexity Analysis:**

**Time complexity:** O(N).

One linear traversal, so time complexity is O(N).

**Space Complexity:** O(1).

As no extra space is required(we are not considering the auxiliary vector).

# Search in a rotated sorted array

Given a sorted and rotated array A of N distinct elements which is rotated at some point, and given an element key. The task is to find the index of the given element key in the array A.

**Example 1:**

**Input:**

N = 9

A[] = {5, 6, 7, 8, 9, 10, 1, 2, 3}

key = 10

**Output**:

5

**Explanation**: 10 is found at index 5.

**Example 2**:

**Input**:

N = 4

A[] = {3, 5, 1, 2}

key = 6

**Output**:

-1

**Explanation**: There is no element that has value 6.

**Your Task**:  
Complete the function search() which takes an array arr[] and start, end index of the array and the K as input parameters, and returns the answer.  
  
Can you solve it in expected time complexity?

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

**Constraints**:  
1 ≤ N ≤ 107  
0 ≤ A[i] ≤ 108  
1 ≤ key ≤ 108

## Solution:

**Basic Solution:**

**Approach:**

1. The idea is to find the pivot point, divide the array in two sub-arrays and perform binary search.
2. The main idea for finding pivot is - for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.
3. Using the above statement and binary search pivot can be found.
4. After the pivot is found out divide the array in two sub-arrays.
5. Now the individual sub - arrays are sorted so the element can be searched using Binary Search.

**Implementation:**

Input arr[] = {3, 4, 5, 1, 2}

Element to Search = 1

1) Find out pivot point and divide the array in two

sub-arrays. (pivot = 2) /\*Index of 5\*/

2) Now call binary search for one of the two sub-arrays.

(a) **If** element is greater than 0th element then

search in left array

(b) **Else** Search in right array

(1 will go in else as 1 < 0th element(3))

3) **If** element is found in selected sub-array then return index

**Else** return -1.

Below is the implementation of the above approach:

/\* C++ Program to search an element

in a sorted and pivoted array\*/

#include <bits/stdc++.h>

using namespace std;

/\* Standard Binary Search function\*/

int binarySearch(int arr[], int low,

int high, int key)

{

if (high < low)

return -1;

int mid = (low + high) / 2; /\*low + (high - low)/2;\*/

if (key == arr[mid])

return mid;

if (key > arr[mid])

return binarySearch(arr, (mid + 1), high, key);

// else

return binarySearch(arr, low, (mid - 1), key);

}

/\* Function to get pivot. For array 3, 4, 5, 6, 1, 2

it returns 3 (index of 6) \*/

int findPivot(int arr[], int low, int high)

{

// base cases

if (high < low)

return -1;

if (high == low)

return low;

int mid = (low + high) / 2; /\*low + (high - low)/2;\*/

if (mid < high && arr[mid] > arr[mid + 1])

return mid;

if (mid > low && arr[mid] < arr[mid - 1])

return (mid - 1);

if (arr[low] >= arr[mid])

return findPivot(arr, low, mid - 1);

return findPivot(arr, mid + 1, high);

}

/\* Searches an element key in a pivoted

sorted array arr[] of size n \*/

int pivotedBinarySearch(int arr[], int n, int key)

{

int pivot = findPivot(arr, 0, n - 1);

// If we didn't find a pivot,

// then array is not rotated at all

if (pivot == -1)

return binarySearch(arr, 0, n - 1, key);

// If we found a pivot, then first compare with pivot

// and then search in two subarrays around pivot

if (arr[pivot] == key)

return pivot;

if (arr[0] <= key)

return binarySearch(arr, 0, pivot - 1, key);

return binarySearch(arr, pivot + 1, n - 1, key);

}

/\* Driver program to check above functions \*/

int main()

{

// Let us search 3 in below array

int arr1[] = { 5, 6, 7, 8, 9, 10, 1, 2, 3 };

int n = sizeof(arr1) / sizeof(arr1[0]);

int key = 3;

// Function calling

cout << "Index of the element is : "

<< pivotedBinarySearch(arr1, n, key);

return 0;

}

**Output:**

Index of the element is : 8

**Complexity Analysis:**

* **Time Complexity:** O(log n).   
  Binary Search requires log n comparisons to find the element. So time complexity is O(log n).
* **Space Complexity:**O(1), No extra space is required

**Improved Solution:**   
**Approach:** Instead of two or more pass of binary search the result can be found in one pass of binary search. The binary search needs to be modified to perform the search. The idea is to create a recursive function that takes l and r as range in input and the key.

1) Find middle point mid = (l + h)/2

2) **If** key is present at middle point, return mid.

3) **Else If** arr[l..mid] is sorted

a) **If** key to be searched lies in range from arr[l]

to arr[mid], recur for arr[l..mid].

b) **Else** recur for arr[mid+1..h]

4) **Else** (arr[mid+1..h] must be sorted)

a) **If** key to be searched lies in range from arr[mid+1]

to arr[h], recur for arr[mid+1..h].

b) **Else** recur for arr[l..mid]

Below is the implementation of above idea:

// Search an element in sorted and rotated

// array using single pass of Binary Search

#include <bits/stdc++.h>

using namespace std;

// Returns index of key in arr[l..h] if

// key is present, otherwise returns -1

int search(int arr[], int l, int h, int key)

{

if (l > h)

return -1;

int mid = (l + h) / 2;

if (arr[mid] == key)

return mid;

/\* If arr[l...mid] is sorted \*/

if (arr[l] <= arr[mid]) {

/\* As this subarray is sorted, we can quickly

check if key lies in half or other half \*/

if (key >= arr[l] && key <= arr[mid])

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

/\*If key not lies in first half subarray,

Divide other half into two subarrays,

such that we can quickly check if key lies

in other half \*/

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

}

/\* If arr[l..mid] first subarray is not sorted, then arr[mid... h]

must be sorted subarray \*/

if (key >= arr[mid] && key <= arr[h])

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

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

}

// Driver program

int main()

{

int arr[] = { 4, 5, 6, 7, 8, 9, 1, 2, 3 };

int n = sizeof(arr) / sizeof(arr[0]);

int key = 6;

int i = search(arr, 0, n - 1, key);

if (i != -1)

cout << "Index: " << i << endl;

else

cout << "Key not found";

}

**Output:**

Index: 2

**Complexity Analysis:**

* **Time Complexity:** O(log n).   
  Binary Search requires log n comparisons to find the element. So time complexity is O(log n).
* **Space Complexity:** O(1).   
  As no extra space is required.

# Square root of an integer

Given an integer x, find it’s square root. If x is not a perfect square, then return floor(√x).

**Examples :**

**Input:** x = 4

**Output:** 2

**Explanation:**  The square root of 4 is 2.

**Input:** x = 11

**Output:** 3

**Explanation:**  The square root of 11 lies in between

3 and 4 so floor of the square root is 3.

## Solution:

**Simple Approach**: To find the floor of the square root, try with all-natural numbers starting from 1. Continue incrementing the number until the square of that number is greater than the given number.

* **Algorithm:**
  1. Create a variable (counter) *i* and take care of some base cases, i.e when the given number is 0 or 1.
  2. Run a loop until *i\*i <= n*, where n is the given number. Increment i by 1.
  3. The floor of the square root of the number is *i – 1*
* **Implementation:**

// 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;

// Starting 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;

}

**Output :**

3

* **Complexity Analysis:**
  + **Time Complexity:** O(√ n).   
    Only one traversal of the solution is needed, so the time complexity is O(√ n).
  + **Space Complexity:** O(1).   
    Constant extra space is needed.

**Better Approach:** The idea is to find the largest integer *i* whose square is less than or equal to the given number. The idea is to use [Binary Search](http://geeksquiz.com/binary-search/) to solve the problem. The values of i \* i is monotonically increasing, so the problem can be solved using binary search.

* **Algorithm:**
  1. Take care of some base cases, i.e when the given number is 0 or 1.
  2. Create some variables, lowerbound *l = 0*, upperbound *r = n*, where n is the given number, *mid* and *ans* to store the answer.
  3. Run a loop until *l <= r* , the search space vanishes
  4. Check if the square of mid (*mid = (l + r)/2*) is less than or equal to n, If yes then search for a larger value in second half of search space, i.e l = mid + 1, update ans = mid
  5. Else if the square of mid is more than n then search for a smaller value in first half of search space, i.e r = mid – 1
  6. Print the value of answer ( *ans*)
* **Implementation:**

// 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;

}

Here basically if we multiply mid with itself so

there will be integer overflow which will throw

tle for larger input so to overcome this

situation we can use long or we can just divide

the number by mid which is same as checking

mid\*mid < x

\*/

if (mid <= x / mid) {

start = mid + 1;

ans = mid;

}

else // If mid\*mid is greater than x

end = mid - 1;

}

return ans;

}

// Driver program

int main()

{

int x = 11;

cout << floorSqrt(x) << endl;

return 0;

}

**Output :**

3

* **Complexity Analysis:**
  + **Time complexity:** O(log n).   
    The time complexity of binary search is O(log n).
  + **Space Complexity:** O(1).   
    Constant extra space is needed.

**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.

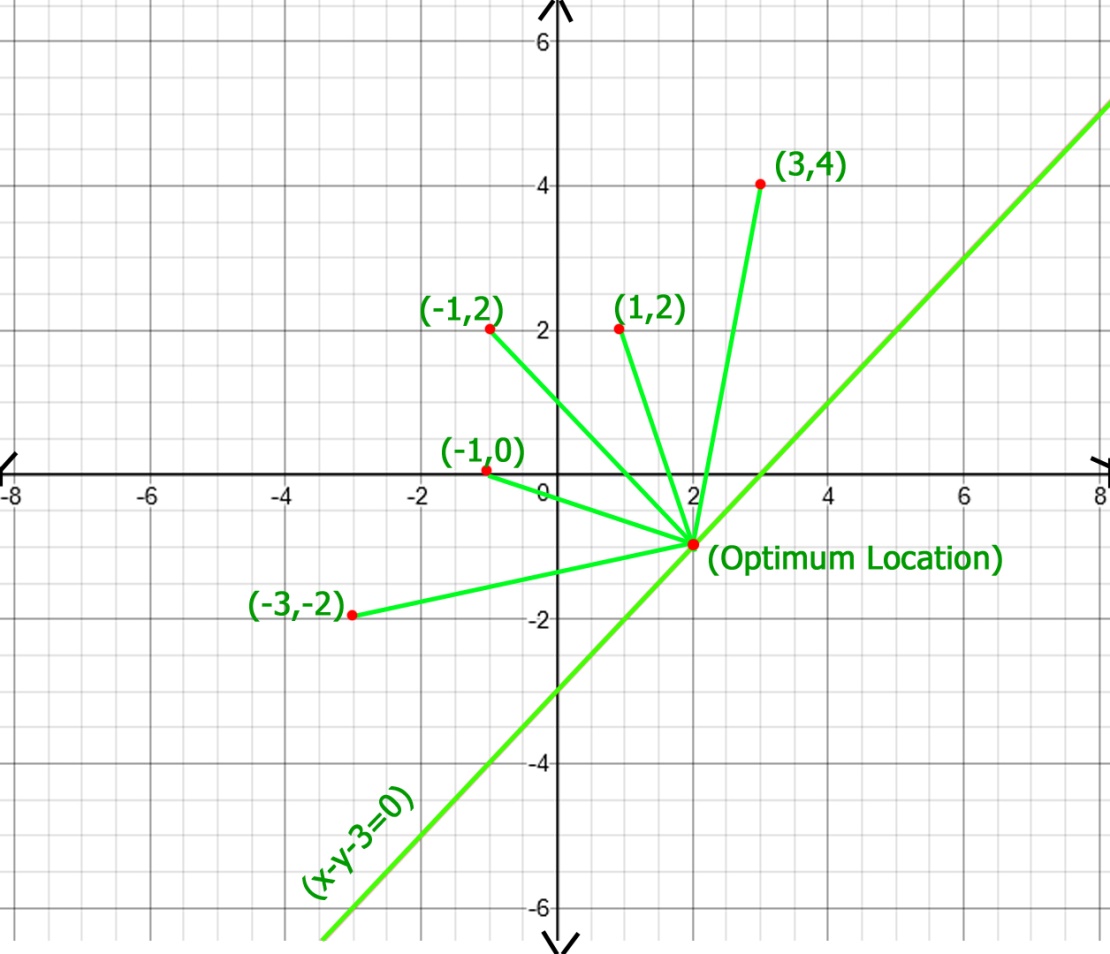
# Maximum and minimum of an array using minimum number of comparisons

## Same as question 2 of Array.

# Optimum location of point to minimize total distance

Given a set of points as and a line as ax+by+c = 0. We need to find a point on given line for which sum of distances from given set of points is minimum.

**Example:**



In above figure optimum location of point of x - y - 3 = 0 line

is (2, -1), whose total distance with other points is 20.77,

which is minimum obtainable total distance.

## Solution:

If we take one point on given line at infinite distance then total distance cost will be infinite, now when we move this point on line towards given points the total distance cost starts decreasing and after some time, it again starts increasing which reached to infinite on the other infinite end of line so distance cost curve looks like a U-curve and we have to find the bottom value of this U-curve.

As U-curve is not monotonically increasing or decreasing we can’t use binary search for finding bottom most point, here we will use ternary search for finding bottom most point, ternary search skips one third of search space at each iteration, you can read more about ternary search [here](https://en.wikipedia.org/wiki/Ternary_search).

So solution proceeds as follows, we start with low and high initialized as some smallest and largest values respectively, then we start iteration, in each iteration we calculate two mids, mid1 and mid2, which represent 1/3rd and 2/3rd position in search space, we calculate total distance of all points with mid1 and mid2 and update low or high by comparing these distance cost, this iteration continues until low and high become approximately equal.

// C/C++ program to find optimum location and total cost

#include <bits/stdc++.h>

using namespace std;

#define sq(x) ((x) \* (x))

#define EPS 1e-6

#define N 5

// structure defining a point

struct point {

int x, y;

point() {}

point(int x, int y)

: x(x)

, y(y)

{

}

};

// structure defining a line of ax + by + c = 0 form

struct line {

int a, b, c;

line(int a, int b, int c)

: a(a)

, b(b)

, c(c)

{

}

};

// method to get distance of point (x, y) from point p

double dist(double x, double y, point p)

{

return sqrt(sq(x - p.x) + sq(y - p.y));

}

/\* Utility method to compute total distance all points

when choose point on given line has x-coordinate

value as X \*/

double compute(point p[], int n, line l, double X)

{

double res = 0;

// calculating Y of chosen point by line equation

double Y = -1 \* (l.c + l.a \* X) / l.b;

for (int i = 0; i < n; i++)

res += dist(X, Y, p[i]);

return res;

}

// Utility method to find minimum total distance

double findOptimumCostUtil(point p[], int n, line l)

{

double low = -1e6;

double high = 1e6;

// loop until difference between low and high

// become less than EPS

while ((high - low) > EPS) {

// mid1 and mid2 are representative x co-ordiantes

// of search space

double mid1 = low + (high - low) / 3;

double mid2 = high - (high - low) / 3;

//

double dist1 = compute(p, n, l, mid1);

double dist2 = compute(p, n, l, mid2);

// if mid2 point gives more total distance,

// skip third part

if (dist1 < dist2)

high = mid2;

// if mid1 point gives more total distance,

// skip first part

else

low = mid1;

}

// compute optimum distance cost by sending average

// of low and high as X

return compute(p, n, l, (low + high) / 2);

}

// method to find optimum cost

double findOptimumCost(int points[N][2], line l)

{

point p[N];

// converting 2D array input to point array

for (int i = 0; i < N; i++)

p[i] = point(points[i][0], points[i][1]);

return findOptimumCostUtil(p, N, l);

}

// Driver code to test above method

int main()

{

line l(1, -1, -3);

int points[N][2] = {

{ -3, -2 }, { -1, 0 }, { -1, 2 }, { 1, 2 }, { 3, 4 }

};

cout << findOptimumCost(points, l) << endl;

return 0;

}

**Output**

20.7652

**Time Complexity:**

Run time order

T(n) = T(2n/3) + 1 = O(logn)

**Space Complexity:**

In above implementation, we have stored all points in struct array. But this can be implemented without it also so, its space complexity is O(1).

# Find the repeating and the missing

Given an unsorted array **Arr** of size **N** of positive integers. **One number 'A'** from set {1, 2, …N} is missing and **one number 'B'** occurs twice in array. Find these two numbers.

**Example 1:**

**Input:**

N = 2

Arr[] = {2, 2}

**Output:** 2 1

**Explanation:** Repeating number is 2 and

smallest positive missing number is 1.

**Example 2:**

**Input:**

N = 3

Arr[] = {1, 3, 3}

**Output:** 3 2

**Explanation:** Repeating number is 3 and

smallest positive missing number is 2.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findTwoElement()** which takes the array of integers **arr** and **n**as parameters and returns an array of integers of size 2 denoting the answer ( The first index contains **B**and second index contains **A.)**

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

**Constraints:**  
1 ≤ N ≤ 105  
1 ≤ Arr[i] ≤ N

## Solution:

**Below are various methods to solve the problems:**

**Method 1 (Use Sorting)**  
**Approach:**

* Sort the input array.
* Traverse the array and check for missing and repeating.

**Time Complexity:** O(nLogn)

Thanks to **LoneShadow** for suggesting this method.

**Method 2 (Use count array)**  
**Approach:**

* Create a temp array temp[] of size n with all initial values as 0.
* Traverse the input array arr[], and do following for each arr[i]
  + if(temp[arr[i]] == 0) temp[arr[i]] = 1;
  + if(temp[arr[i]] == 1) output “arr[i]” //repeating
* Traverse temp[] and output the array element having value as 0 (This is the missing element)

**Time Complexity:** O(n)

**Auxiliary Space:** O(n)

**Method 3 (Use elements as Index and mark the visited places)**  
**Approach:**   
Traverse the array. While traversing, use the absolute value of every element as an index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

// C++ program to Find the repeating

// and missing elements

#include <bits/stdc++.h>

using namespace std;

void printTwoElements(int arr[], int size)

{

int i;

cout << " The repeating element is ";

for (i = 0; i < size; i++) {

if (arr[abs(arr[i]) - 1] > 0)

arr[abs(arr[i]) - 1] = -arr[abs(arr[i]) - 1];

else

cout << abs(arr[i]) << "\n";

}

cout << "and the missing element is ";

for (i = 0; i < size; i++) {

if (arr[i] > 0)

cout << (i + 1);

}

}

/\* Driver code \*/

int main()

{

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

int n = sizeof(arr) / sizeof(arr[0]);

printTwoElements(arr, n);

}

**Output**

The repeating element is 5

and the missing element is 1

**Time Complexity:** O(n)  
Thanks to **Manish Mishra** for suggesting this method.

**Method 4 (Make two equations)**  
**Approach:**

* Let x be the missing and y be the repeating element.
* Get the sum of all numbers using formula **S = n(n+1)/2 – x + y**
* Get product of all numbers using formula **P = 1\*2\*3\*…\*n \* y / x**
* The above two steps give us two equations, we can solve the equations and get the values of x and y.

**Time Complexity:** O(n).

**Note:** This method can cause arithmetic overflow as we calculate product and sum of all array elements.

**Method 5 (Use XOR)**

**Approach:**

* Let x and y be the desired output elements.
* Calculate XOR of all the array elements.

***xor1 = arr[0]^arr[1]^arr[2]…..arr[n-1]***

* XOR the result with all numbers from 1 to n

***xor1 = xor1^1^2^…..^n***

* In the result *xor1*, all elements would nullify each other except x and y. All the bits that are set in *xor1* will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of *xor1*and divide the elements of the array in two sets – one set of elements with same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x, and by doing same in other set we will get y.

Below is the implementation of the above approach:

// C++ program to Find the repeating

// and missing elements

#include <bits/stdc++.h>

using namespace std;

/\* The output of this function is stored at

\*x and \*y \*/

void getTwoElements(int arr[], int n,

int\* x, int\* y)

{

/\* Will hold xor of all elements

and numbers from 1 to n \*/

int xor1;

/\* Will have only single set bit of xor1 \*/

int set\_bit\_no;

int i;

\*x = 0;

\*y = 0;

xor1 = arr[0];

/\* Get the xor of all array elements \*/

for (i = 1; i < n; i++)

xor1 = xor1 ^ arr[i];

/\* XOR the previous result with numbers

from 1 to n\*/

for (i = 1; i <= n; i++)

xor1 = xor1 ^ i;

/\* Get the rightmost set bit in set\_bit\_no \*/

set\_bit\_no = xor1 & ~(xor1 - 1);

/\* Now divide elements into two

sets by comparing a rightmost set

bit of xor1 with the bit at the same

position in each element. Also,

get XORs of two sets. The two

XORs are the output elements.

The following two for loops

serve the purpose \*/

for (i = 0; i < n; i++) {

if (arr[i] & set\_bit\_no)

/\* arr[i] belongs to first set \*/

\*x = \*x ^ arr[i];

else

/\* arr[i] belongs to second set\*/

\*y = \*y ^ arr[i];

}

for (i = 1; i <= n; i++) {

if (i & set\_bit\_no)

/\* i belongs to first set \*/

\*x = \*x ^ i;

else

/\* i belongs to second set\*/

\*y = \*y ^ i;

}

/\* \*x and \*y hold the desired

output elements \*/

}

/\* Driver code \*/

int main()

{

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

int\* x = (int\*)malloc(sizeof(int));

int\* y = (int\*)malloc(sizeof(int));

int n = sizeof(arr) / sizeof(arr[0]);

getTwoElements(arr, n, x, y);

cout << \*x <<” ”<< \*y;

getchar();

}

**Output**

**7 5**

**Time Complexity:** O(n)  
This method doesn’t cause overflow, but it doesn’t tell which one occurs twice and which one is missing. We can add one more step that checks which one is missing and which one is repeating. This can be easily done in O(n) time.

**Method 6 (Use a Map)**  
**Approach:**   
This method involves creating a Hashtable with the help of Map. In this, the elements are mapped to their natural index. In this process, if an element is mapped twice, then it is the repeating element. And if an element’s mapping is not there, then it is the missing element.

Below is the implementation of the above approach:

// C++ program to find the repeating

// and missing elements using Maps

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

int arr[] = { 4, 3, 6, 2, 1, 1 };

int n = 6;

unordered\_map<int, bool> numberMap;

for(int i : arr)

{

if (numberMap.find(i) ==

numberMap.end())

{

numberMap[i] = true;

}

else

{

cout << "Repeating = " << i;

}

}

cout << endl;

for(int i = 1; i <= n; i++)

{

if (numberMap.find(i) ==

numberMap.end())

{

cout << "Missing = " << i;

}

}

return 0;

}

**Output**

Repeating = 1

Missing = 5

**Method 7 (Make two equations using sum and sum of squares)**  
**Approach:**

* Let x be the missing and y be the repeating element.
* Let N is the size of array.
* Get the sum of all numbers using formula **S = N(N+1)/2**
* Get the sum of square of all numbers using formula **Sum\_Sq = N(N+1)(2N+1)/6**
* Iterate through a loop from i=1….N
* **S -= A[i]**
* **Sum\_Sq -= (A[i]\*A[i])**
* It will give two equations   
  x-y = S – (1)   
  x^2 – y^2 = Sum\_sq   
  x+ y = (Sum\_sq/S) – (2)

**Time Complexity:** O(n)

#include <bits/stdc++.h>

using namespace std;

vector<int>repeatedNumber(const vector<int> &A) {

long long int len = A.size();

long long int Sum\_N = (len \* (len+1) ) /2, Sum\_NSq = (len \* (len +1) \*(2\*len +1) )/6;

long long int missingNumber=0, repeating=0;

for(int i=0;i<A.size(); i++){

Sum\_N -= (long long int)A[i];

Sum\_NSq -= (long long int)A[i]\*(long long int)A[i];

}

missingNumber = (Sum\_N + Sum\_NSq/Sum\_N)/2;

repeating= missingNumber - Sum\_N;

vector <int> ans;

ans.push\_back(repeating);

ans.push\_back(missingNumber);

return ans;

}

int main(void){

std::vector<int> v = {4, 3, 6, 2, 1, 6,7};

vector<int> res = repeatedNumber(v);

for(int x: res){

cout<< x<<" ";

}

cout<<endl;

}

**Output**

6 5

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

**Method 8 (Using OR Operator):**

**Approach:**

*Given an input array*

1. *Performing OR operation on input array.*
2. *At the same time checking if that number has occurred before, by determining if the position is already set or not. We will get the repeating number in this step.*
3. *To find missing value we have to check the bit containing 0 using OR again.*

#include <bits/stdc++.h>

using namespace std;

int main()

{

// Input:

vector<int> arr = {4, 3, 6, 2, 1, 1};

int n = arr.size();

// Declaring output variables

// Note : arr[i]-1 is used instead of arr[i] as we want to use all 64 bits

int bitOr = (1 << (arr[0]-1));

int repeating, missing;

// Performing XOR as well as Checking repeating number

for(int i=1; i<n; i++){

// If OR operation with 1 gives same output that means, we already have 1 at that position

if((bitOr | (1 << (arr[i]-1))) == bitOr) {

repeating = arr[i];

continue;

}

bitOr = (bitOr | (1 << (arr[i]-1)));

}

// Checking missing number

for(int i=0; i<n; i++){

// property: OR with 0 yield 1 hence value of bitOr changes

if((bitOr | (1 << i)) != bitOr) {

missing = i+1;

break;

}

}

cout << "Repeating : " << repeating << "\nMissing : " << missing;

return 0;

}

**Output**

Repeating : 1

Missing : 5

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

**Limitations of the approach**: it only works on *size of array <= 64* if we use long and *size of array <= 32*

{\displaystyle T(n)=T(2n/3)+1=\Theta (\log n)}

# find majority element

Given an array **A** of **N** elements. Find the majority element in the array. A majority element in an array A of size N is an **element that appears more than N/2 times in the array**.

**Example 1:**

**Input:**

N = 3

A[] = {1,2,3}

**Output:**

-1

**Explanation:**

Since, each element in

{1,2,3} appears only once so there

is no majority element.

**Example 2:**

**Input:**

N = 5

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

**Output:**

3

**Explanation:**

Since, 3 is present more

than N/2 times, so it is

the majority element.

**Your Task:**  
The task is to complete the function **majorityElement**() which returns the majority element in the array. If no majority exists, return -1.

**Expected Time Complexity:** O(N).  
**Expected Auxiliary Space:** O(1).

**Constraints:**  
1 ≤ N ≤ 107  
0 ≤ Ai ≤ 106

## Solution:

**METHOD 1**

* **Approach:** The basic solution is to have two loops and keep track of the maximum count for all different elements. If maximum count becomes greater than n/2 then break the loops and return the element having maximum count. If the maximum count doesn’t become more than n/2 then the majority element doesn’t exist.
* **Algorithm:**
  1. Create a variable to store the max count, *count = 0*
  2. Traverse through the array from start to end.
  3. For every element in the array run another loop to find the count of similar elements in the given array.
  4. If the count is greater than the max count update the max count and store the index in another variable.
  5. If the maximum count is greater than the half the size of the array, print the element. Else print there is no majority element.

Below is the implementation of the above idea:

// C++ program to find Majority

// element in an array

#include <bits/stdc++.h>

using namespace std;

// Function to find Majority element

// in an array

void findMajority(int arr[], int n)

{

int maxCount = 0;

int index = -1; // sentinels

for (int i = 0; i < n; i++) {

int count = 0;

for (int j = 0; j < n; j++) {

if (arr[i] == arr[j])

count++;

}

// update maxCount if count of

// current element is greater

if (count > maxCount) {

maxCount = count;

index = i;

}

}

// if maxCount is greater than n/2

// return the corresponding element

if (maxCount > n / 2)

cout << arr[index] << endl;

else

cout << "No Majority Element" << endl;

}

// Driver code

int main()

{

int arr[] = { 1, 1, 2, 1, 3, 5, 1 };

int n = sizeof(arr) / sizeof(arr[0]);

// Function calling

findMajority(arr, n);

return 0;

}

**Output**

1

**Complexity Analysis:**

* **Time Complexity:** O(n\*n).   
  A nested loop is needed where both the loops traverse the array from start to end, so the time complexity is O(n^2).
* **Auxiliary Space:** O(1).   
  As no extra space is required for any operation so the space complexity is constant.

**METHOD 2 (Using**[**Binary Search Tree**](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)**)**

* **Approach:** Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if the count of a node becomes more than n/2 then return.
* **Algorithm:**
  1. Create a binary search tree, if same element is entered in the binary search tree the frequency of the node is increased.
  2. traverse the array and insert the element in the binary search tree.
  3. If the maximum frequency of any node is greater than the half the size of the array, then perform a inorder traversal and find the node with frequency greater than half
  4. Else print No majority Element.

Below is the implementation of the above idea:

// C++ program to demonstrate insert operation in binary

// search tree.

#include <bits/stdc++.h>

using namespace std;

struct node {

int key;

int c = 0;

struct node \*left, \*right;

};

// A utility function to create a new BST node

struct node\* newNode(int item)

{

struct node\* temp

= (struct node\*)malloc(sizeof(struct node));

temp->key = item;

temp->c = 1;

temp->left = temp->right = NULL;

return temp;

}

// A utility function to insert a new node with given key in

// BST

struct node\* insert(struct node\* node, int key, int& ma)

{

// If the tree is empty, return a new node

if (node == NULL) {

if (ma == 0)

ma = 1;

return newNode(key);

}

// Otherwise, recur down the tree

if (key < node->key)

node->left = insert(node->left, key, ma);

else if (key > node->key)

node->right = insert(node->right, key, ma);

else

node->c++;

// find the max count

ma = max(ma, node->c);

// return the (unchanged) node pointer

return node;

}

// A utility function to do inorder traversal of BST

void inorder(struct node\* root, int s)

{

if (root != NULL) {

inorder(root->left, s);

if (root->c > (s / 2))

printf("%d \n", root->key);

inorder(root->right, s);

}

}

// Driver Code

int main()

{

int a[] = { 1, 3, 3, 3, 2 };

int size = (sizeof(a)) / sizeof(a[0]);

struct node\* root = NULL;

int ma = 0;

for (int i = 0; i < size; i++) {

root = insert(root, a[i], ma);

}

// Function call

if (ma > (size / 2))

inorder(root, size);

else

cout << "No majority element\n";

return 0;

}

**Output**

3

**Complexity Analysis:**

* **Time Complexity:** If a [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) is used then time complexity will be O(n^2). If a [self-balancing-binary-search](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) tree is used then it will be O(nlogn)
* **Auxiliary Space:**O(n).   
  As extra space is needed to store the array in tree.

**METHOD 3 (Using Moore’s Voting Algorithm):**

* **Approach:** This is a two-step process.
  + The first step gives the element that maybe the majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise, it will return candidate for majority element.
  + Check if the element obtained from the above step is majority element. This step is necessary as there might be no majority element.
* **Algorithm:**
  + Loop through each element and maintains a count of majority element, and a majority index, *maj\_index*
  + If the next element is same then increment the count if the next element is not same then decrement the count.
  + if the count reaches 0 then changes the maj\_index to the current element and set the count again to 1.
  + Now again traverse through the array and find the count of majority element found.
  + If the count is greater than half the size of the array, print the element
  + Else print that there is no majority element

Below is the implementation of above idea:

// C++ Program for finding out

// majority element in an array

#include <bits/stdc++.h>

using namespace std;

/\* Function to find the candidate for Majority \*/

int findCandidate(int a[], int size)

{

int maj\_index = 0, count = 1;

for (int i = 1; i < size; i++) {

if (a[maj\_index] == a[i])

count++;

else

count--;

if (count == 0) {

maj\_index = i;

count = 1;

}

}

return a[maj\_index];

}

/\* Function to check if the candidate

occurs more than n/2 times \*/

bool isMajority(int a[], int size, int cand)

{

int count = 0;

for (int i = 0; i < size; i++)

if (a[i] == cand)

count++;

if (count > size / 2)

return 1;

else

return 0;

}

/\* Function to print Majority Element \*/

void printMajority(int a[], int size)

{

/\* Find the candidate for Majority\*/

int cand = findCandidate(a, size);

/\* Print the candidate if it is Majority\*/

if (isMajority(a, size, cand))

cout << " " << cand << " ";

else

cout << "No Majority Element";

}

/\* Driver code \*/

int main()

{

int a[] = { 1, 3, 3, 1, 2 };

int size = (sizeof(a)) / sizeof(a[0]);

// Function calling

printMajority(a, size);

return 0;

}

**Output**

No Majority Element

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  As two traversal of the array is needed, so the time complexity is linear.
* **Auxiliary Space:** O(1).   
  As no extra space is required.

**METHOD 4 (Using Hashmap):**

* **Approach:** This method is somewhat similar to Moore voting algorithm in terms of time complexity, but in this case, there is no need for the second step of Moore voting algorithm. But as usual, here space complexity becomes O(n).   
  In Hashmap(key-value pair), at value, maintain a count for each element(key) and whenever the count is greater than half of the array length, return that key(majority element).
* **Algorithm:**
  1. Create a hashmap to store a key-value pair, i.e. element-frequency pair.
  2. Traverse the array from start to end.
  3. For every element in the array, insert the element in the hashmap if the element does not exist as key, else fetch the value of the key ( array[i] ), and increase the value by 1
  4. If the count is greater than half then print the majority element and break.
  5. If no majority element is found print “No Majority element”

Below is the implementation of above idea:

/\* C++ program for finding out majority

element in an array \*/

#include <bits/stdc++.h>

using namespace std;

void findMajority(int arr[], int size)

{

unordered\_map<int, int> m;

for(int i = 0; i < size; i++)

m[arr[i]]++;

int count = 0;

for(auto i : m)

{

if(i.second > size / 2)

{

count =1;

cout << "Majority found :- " << i.first<<endl;

break;

}

}

if(count == 0)

cout << "No Majority element" << endl;

}

// Driver code

int main()

{

int arr[] = {2, 2, 2, 2, 5, 5, 2, 3, 3};

int n = sizeof(arr) / sizeof(arr[0]);

// Function calling

findMajority(arr, n);

return 0;

}

**Output**

Majority found :- 2

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  One traversal of the array is needed, so the time complexity is linear.
* **Auxiliary Space:** O(n).   
  Since a hashmap requires linear space.

**METHOD 5**

* **Approach:**The idea is to sort the array. Sorting makes similar elements in the array adjacent, so traverse the array and update the count until the present element is similar to the previous one. If the frequency is more than half the size of the array, print the majority element.
* **Algorithm:**
  1. Sort the array and create a variable count and previous, *prev = INT\_MIN*.
  2. Traverse the element from start to end.
  3. If the current element is equal to the previous element increase the count.
  4. Else set the count to 1.
  5. If the count is greater than half the size of array, print the element as majority element and break.
  6. If no majority element found, print “No majority element”

Below is the implementation of the above idea:

// C++ program to find Majority

// element in an array

#include <bits/stdc++.h>

using namespace std;

// Function to find Majority element

// in an array

// it returns -1 if there is no majority element

int majorityElement(int \*arr, int n)

{

// sort the array in O(nlogn)

sort(arr, arr+n);

int count = 1, max\_ele = -1, temp = arr[0], ele, f=0;

for(int i=1;i<n;i++)

{

// increases the count if the same element occurs

// otherwise starts counting new element

if(temp==arr[i])

{

count++;

}

else

{

count = 1;

temp = arr[i];

}

// sets maximum count

// and stores maximum occured element so far

// if maximum count becomes greater than n/2

// it breaks out setting the flag

if(max\_ele<count)

{

max\_ele = count;

ele = arr[i];

if(max\_ele>(n/2))

{

f = 1;

break;

}

}

}

// returns maximum occured element

// if there is no such element, returns -1

return (f==1 ? ele : -1);

}

// Driver code

int main()

{

int arr[] = {1, 1, 2, 1, 3, 5, 1};

int n = sizeof(arr) / sizeof(arr[0]);

// Function calling

cout<<majorityElement(arr, n);

return 0;

}

**Output**

1

**Complexity Analysis:**

* **Time Complexity:**O(nlogn).   
  Sorting requires O(n log n) time complexity.
* **Auxiliary Space:** O(1).   
  As no extra space is required.

# Searching in an array where adjacent differ by at most k

A step array is an array of integers where each element has a difference of at most k with its neighbor. Given a key x, we need to find the index value of x if multiple-element exist to return the first occurrence of the key.  
Examples: 

Input : arr[] = {4, 5, 6, 7, 6}

k = 1

x = 6

Output : 2

The first index of 6 is 2.

Input : arr[] = {20, 40, 50, 70, 70, 60}

k = 20

x = 60

Output : 5

The index of 60 is 5

## Solution:

A **Simple Approach** is to traverse the given array one by one and compare every element with the given element ‘x’. If matches, then return index.  
The above solution can be **Optimized**using the fact that the difference between all adjacent elements is at most k. The idea is to start comparing from the leftmost element and find the difference between the current array element and x. Let this difference be ‘diff’. From the given property of the array, we always know that x must be at least ‘diff/k’ away, so instead of searching one by one, we jump ‘diff/k’.   
Below is the implementation of the above idea.

// C++ program to search an element in an array

// where difference between adjacent elements is atmost k

#include<bits/stdc++.h>

using namespace std;

// x is the element to be searched in arr[0..n-1]

// such that all elements differ by at-most k.

int search(int arr[], int n, int x, int k)

{

// Traverse the given array starting from

// leftmost element

int i = 0;

while (i < n)

{

// If x is found at index i

if (arr[i] == x)

return i;

// Jump the difference between current

// array element and x divided by k

// We use max here to make sure that i

// moves at-least one step ahead.

i = i + max(1, abs(arr[i]-x)/k);

}

cout << "number is not present!";

return -1;

}

// Driver program to test above function

int main()

{

int arr[] = {2, 4, 5, 7, 7, 6};

int x = 6;

int k = 2;

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Element " << x << " is present at index "

<< search(arr, n, x, k);

return 0;

}

**Output:**

Element 6 is present at index

# find a pair with a given difference

Given an array **Arr[]** of size **L** and a number **N,** you need to write a program to find if there exists a pair of elements in the array whose difference is **N**.

**Example 1:**

**Input:**

L = 6, N = 78

arr[] = {5, 20, 3, 2, 5, 80}

**Output:** 1

**Explanation:** (2, 80) have difference of 78.

**Example 2:**

**Input:**

L = 5, N = 45

arr[] = {90, 70, 20, 80, 50}

**Output:** -1

**Explanation:** There is no pair with difference of 45.

**Your Task:**  
You need not take input or print anything. Your task is to complete the function **findPair()**which takes array arr, size of the array L and N as input parameters and returns True if required pair exists, else return False.

**Expected Time Complexity:**O(L\* Log(L)).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 ≤ L ≤ 104  
1 ≤ Arr[i], N ≤ 105

## Solution:

The simplest method is to run two loops, the outer loop picks the first element (smaller element) and the inner loop looks for the element picked by outer loop plus n. Time complexity of this method is O(n^2).  
We can use sorting and Binary Search to improve time complexity to O(nLogn). The first step is to sort the array in ascending order. Once the array is sorted, traverse the array from left to right, and for each element arr[i], binary search for arr[i] + n in arr[i+1..n-1]. If the element is found, return the pair.   
Both first and second steps take O(nLogn). So overall complexity is O(nLogn).   
The second step of the above algorithm can be improved to O(n). The first step remain same. The idea for second step is take two index variables i and j, initialize them as 0 and 1 respectively. Now run a linear loop. If arr[j] – arr[i] is smaller than n, we need to look for greater arr[j], so increment j. If arr[j] – arr[i] is greater than n, we need to look for greater arr[i], so increment i. Thanks to Aashish Barnwal for suggesting this approach.   
The following code is only for the second step of the algorithm, it assumes that the array is already sorted.

// C++ program to find a pair with the given difference

#include <bits/stdc++.h>

using namespace std;

// The function assumes that the array is sorted

bool findPair(int arr[], int size, int n)

{

// Initialize positions of two elements

int i = 0;

int j = 1;

// Search for a pair

while (i < size && j < size)

{

if (i != j && (arr[j] - arr[i] == n || arr[i] - arr[j] == n) )

{

cout << "Pair Found: (" << arr[i] <<

", " << arr[j] << ")";

return true;

}

else if (arr[j]-arr[i] < n)

j++;

else

i++;

}

cout << "No such pair";

return false;

}

// Driver program to test above function

int main()

{

int arr[] = {1, 8, 30, 40, 100};

int size = sizeof(arr)/sizeof(arr[0]);

int n = -60;

findPair(arr, size, n);

return 0;

}

**Output**

Pair Found: (100, 40)

Hashing can also be used to solve this problem. Create an empty hash table HT. Traverse the array, use array elements as hash keys and enter them in HT. Traverse the array again look for value n + arr[i] in HT.

# find four elements that sum to a given value

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X.   
For example, if the given array is {10, 2, 3, 4, 5, 9, 7, 8} and X = 23, then your function should print “3 5 7 8” (3 + 5 + 7 + 8 = 23).

## Solution:

A **Naive Solution**is to generate all possible quadruples and compare the sum of every quadruple with X. The following code implements this simple method using four nested loops

// C++ program for naive solution to

// print all combination of 4 elements

// in A[] with sum equal to X

#include <bits/stdc++.h>

using namespace std;

/\* A naive solution to print all combination

of 4 elements in A[]with sum equal to X \*/

void findFourElements(int A[], int n, int X)

{

// Fix the first element and find other three

for (int i = 0; i < n - 3; i++)

{

// Fix the second element and find other two

for (int j = i + 1; j < n - 2; j++)

{

// Fix the third element and find the fourth

for (int k = j + 1; k < n - 1; k++)

{

// find the fourth

for (int l = k + 1; l < n; l++)

if (A[i] + A[j] + A[k] + A[l] == X)

cout << A[i] <<", " << A[j]

<< ", " << A[k] << ", " << A[l];

}

}

}

}

// Driver Code

int main()

{

int A[] = {10, 20, 30, 40, 1, 2};

int n = sizeof(A) / sizeof(A[0]);

int X = 91;

findFourElements (A, n, X);

return 0;

}

**Output:**

20, 30, 40, 1

Time Complexity: O(n^4)  
*The time complexity can be improved to O(n^3) with the****use of sorting****as a preprocessing step, and then using method 1 of*[*this*](https://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-two-elements-in-s-whose-sum-is-exactly-x/)*post to reduce a loop.*  
Following are the detailed steps.   
1) Sort the input array.   
2) Fix the first element as A[i] where i is from 0 to n–3. After fixing the first element of quadruple, fix the second element as A[j] where j varies from i+1 to n-2. Find remaining two elements in O(n) time, using the method 1 of [this](https://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-two-elements-in-s-whose-sum-is-exactly-x/)post   
Following is the implementation of O(n^3) solution.

// C++ program for to print all combination

// of 4 elements in A[] with sum equal to X

#include<bits/stdc++.h>

using namespace std;

/\* Following function is needed

for library function qsort(). \*/

int compare (const void \*a, const void \* b)

{

return ( \*(int \*)a - \*(int \*)b );

}

/\* A sorting based solution to print

all combination of 4 elements in A[]

with sum equal to X \*/

void find4Numbers(int A[], int n, int X)

{

int l, r;

// Sort the array in increasing

// order, using library function

// for quick sort

qsort (A, n, sizeof(A[0]), compare);

/\* Now fix the first 2 elements

one by one and find

the other two elements \*/

for (int i = 0; i < n - 3; i++)

{

for (int j = i+1; j < n - 2; j++)

{

// Initialize two variables as

// indexes of the first and last

// elements in the remaining elements

l = j + 1;

r = n-1;

// To find the remaining two

// elements, move the index

// variables (l & r) toward each other.

while (l < r)

{

if( A[i] + A[j] + A[l] + A[r] == X)

{

cout << A[i]<<", " << A[j] <<

", " << A[l] << ", " << A[r];

l++; r--;

}

else if (A[i] + A[j] + A[l] + A[r] < X)

l++;

else // A[i] + A[j] + A[l] + A[r] > X

r--;

} // end of while

} // end of inner for loop

} // end of outer for loop

}

/\* Driver code \*/

int main()

{

int A[] = {1, 4, 45, 6, 10, 12};

int X = 21;

int n = sizeof(A) / sizeof(A[0]);

find4Numbers(A, n, X);

return 0;

}

**Output :**

1, 4, 6, 10

**Time Complexity :** O(n^3)

The problem can be solved in **O(n2 Logn)** time with the help of auxiliary space.   
*Thanks to itsnimish for suggesting this method. Following is the detailed process.*

**Method 1:** Two Pointers Algorithm.   
**Approach:** Let the input array be A[].

1. Create an auxiliary array aux[] and store sum of all possible pairs in aux[]. The size of aux[] will be n\*(n-1)/2 where n is the size of A[].
2. Sort the auxiliary array aux[].
3. Now the problem reduces to find two elements in aux[] with sum equal to X. We can use method 1 of this post to find the two elements efficiently. There is following important point to note though:   
   An element of aux[] represents a pair from A[]. While picking two elements from aux[], we must check whether the two elements have an element of A[] in common. For example, if first element sum of A[1] and A[2], and second element is sum of A[2] and A[4], then these two elements of aux[] don’t represent four distinct elements of input array A[].

Below is the implementation of the above approach:

// C++ program to find 4 elements

// with given sum

#include <bits/stdc++.h>

using namespace std;

// The following structure is needed

// to store pair sums in aux[]

class pairSum {

public:

// index (int A[]) of first element in pair

int first;

// index of second element in pair

int sec;

// sum of the pair

int sum;

};

// Following function is needed

// for library function qsort()

int compare(const void\* a, const void\* b)

{

return ((\*(pairSum\*)a).sum - (\*(pairSum\*)b).sum);

}

// Function to check if two given pairs

// have any common element or not

bool noCommon(pairSum a, pairSum b)

{

if (a.first == b.first || a.first == b.sec

|| a.sec == b.first || a.sec == b.sec)

return false;

return true;

}

// The function finds four

// elements with given sum X

void findFourElements(int arr[], int n, int X)

{

int i, j;

// Create an auxiliary array

// to store all pair sums

int size = (n \* (n - 1)) / 2;

pairSum aux[size];

// Generate all possible pairs

// from A[] and store sums

// of all possible pairs in aux[]

int k = 0;

for (i = 0; i < n - 1; i++) {

for (j = i + 1; j < n; j++) {

aux[k].sum = arr[i] + arr[j];

aux[k].first = i;

aux[k].sec = j;

k++;

}

}

// Sort the aux[] array using

// library function for sorting

qsort(aux, size, sizeof(aux[0]), compare);

// Now start two index variables

// from two corners of array

// and move them toward each other.

i = 0;

j = size - 1;

while (i < size && j >= 0) {

if ((aux[i].sum + aux[j].sum == X)

&& noCommon(aux[i], aux[j])) {

cout << arr[aux[i].first] << ", "

<< arr[aux[i].sec] << ", "

<< arr[aux[j].first] << ", "

<< arr[aux[j].sec] << endl;

return;

}

else if (aux[i].sum + aux[j].sum < X)

i++;

else

j--;

}

}

// Driver code

int main()

{

int arr[] = { 10, 20, 30, 40, 1, 2 };

int n = sizeof(arr) / sizeof(arr[0]);

int X = 91;

// Function Call

findFourElements(arr, n, X);

return 0;

}

**Output**

20, 1, 30, 40

Please note that the above code prints only one quadruple. If we remove the return statement and add statements “i++; j–;”, then it prints same quadruple five times. The code can modified to print all quadruples only once. It has been kept this way to keep it simple.   
**Complexity Analysis:**

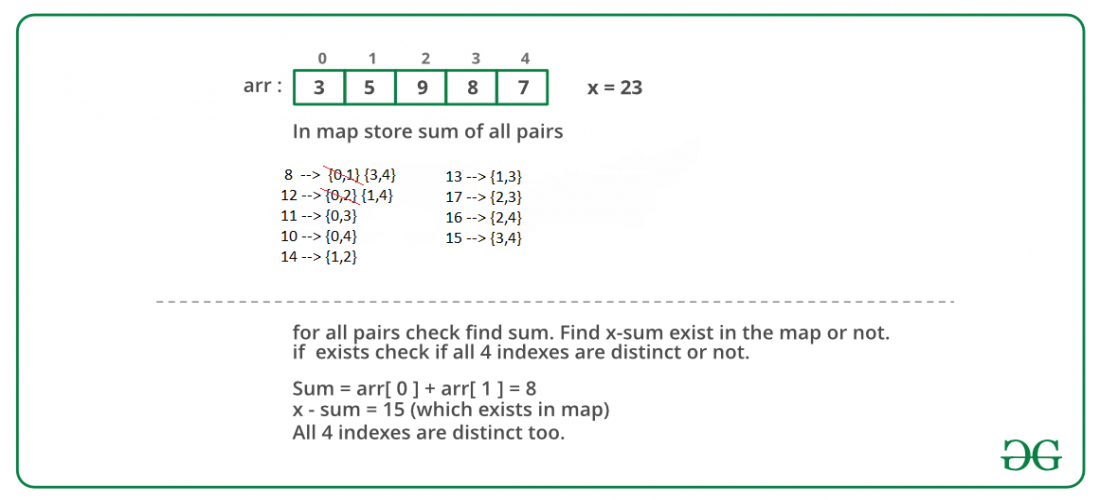
* **Time complexity:** O(n^2Logn).   
  The step 1 takes O(n^2) time. The second step is sorting an array of size O(n^2). Sorting can be done in O(n^2Logn) time using merge sort or heap sort or any other O(nLogn) algorithm. The third step takes O(n^2) time. So overall complexity is O(n^2Logn).
* **Auxiliary Space:** O(n^2).   
  The size of the auxiliary array is O(n^2). The big size of the auxiliary array can be a concern in this method.

**Method 2:** Hashing Based Solution[O(n2)]

**Approach:**

1. Store sums of all pairs in a hash table
2. Traverse through all pairs again and search for **X – (current pair sum)** in the hash table.
3. If a pair is found with the required sum, then make sure that all elements are distinct array elements and an element is not considered more than once.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// A hashing based CPP program

// to find if there are

// four elements with given sum.

#include <bits/stdc++.h>

using namespace std;

// The function finds four

// elements with given sum X

void findFourElements(int arr[], int n, int X)

{

// Store sums of all pairs

// in a hash table

unordered\_map<int, pair<int, int> > mp;

for (int i = 0; i < n - 1; i++)

for (int j = i + 1; j < n; j++)

mp[arr[i] + arr[j]] = { i, j };

// Traverse through all pairs and search

// for X - (current pair sum).

for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

int sum = arr[i] + arr[j];

// If X - sum is present in hash table,

if (mp.find(X - sum) != mp.end()) {

// Making sure that all elements are

// distinct array elements and an element

// is not considered more than once.

pair<int, int> p = mp[X - sum];

if (p.first != i && p.first != j

&& p.second != i && p.second != j) {

cout << arr[i] << ", " << arr[j] << ", "

<< arr[p.first] << ", "

<< arr[p.second];

return;

}

}

}

}

}

// Driver code

int main()

{

int arr[] = { 10, 20, 30, 40, 1, 2 };

int n = sizeof(arr) / sizeof(arr[0]);

int X = 91;

// Function call

findFourElements(arr, n, X);

return 0;

}

**Output**

20, 30, 40, 1

**Complexity Analysis:**

* **Time complexity:** O(n^2).   
  Nested traversal is needed to store all pairs in the hash Map.
* **Auxiliary Space:** O(n^2).   
  All n\*(n-1) pairs are stored in hash Map so the space required is O(n^2)  
  Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

**Method 3:** Solution having no duplicate elements

**Approach:**

1. Store sums of all pairs in a hash table
2. Traverse through all pairs again and search for X – (current pair sum) in the hash table.
3. Consider a temp array that is initially stored with zeroes. It is changed to 1 when we get 4 elements that sum up to the required value.
4. If a pair is found with the required sum, then make sure that all elements are distinct array elements and check if the value in temp array is 0 so that duplicates are not considered.

Below is the implementation of the code:

// C++ program to find four

// elements with the given sum

#include <bits/stdc++.h>

using namespace std;

// Function to find 4 elements that add up to

// given sum

void fourSum(int X, int arr[], map<int,

pair<int, int>> Map, int N)

{

int temp[N];

// Iterate from 0 to temp.length

for (int i = 0; i < N; i++)

temp[i] = 0;

// Iterate from 0 to arr.length

for (int i = 0; i < N - 1; i++)

{

// Iterate from i + 1 to arr.length

for (int j = i + 1; j < N; j++)

{

// Store curr\_sum = arr[i] + arr[j]

int curr\_sum = arr[i] + arr[j];

// Check if X - curr\_sum if present

// in map

if (Map.find(X - curr\_sum) != Map.end())

{

// Store pair having map value

// X - curr\_sum

pair<int, int> p = Map[X - curr\_sum];

if (p.first != i && p.second != i

&& p.first != j && p.second != j

&& temp[p.first] == 0

&& temp[p.second] == 0 && temp[i] == 0

&& temp[j] == 0)

{

// Print the output

cout << arr[i] << "," << arr[j] <<

"," << arr[p.first] << "," << arr[p.second];

temp[p.second] = 1;

temp[i] = 1;

temp[j] = 1;

break;

}

}

}

}

}

// Program for two Sum

map<int, pair<int, int>> twoSum(int nums[], int N)

{

map<int, pair<int, int>> Map;

for (int i = 0; i < N - 1; i++)

{

for (int j = i + 1; j < N; j++)

{

Map[nums[i] + nums[j]].first = i;

Map[nums[i] + nums[j]].second = j;

}

}

return Map;

}

// Driver code

int main()

{

int arr[] = { 10, 20, 30, 40, 1, 2 };

int n = sizeof(arr) / sizeof(arr[0]);

int X = 91;

map<int, pair<int, int>> Map = twoSum(arr, n);

// Function call

fourSum(X, arr, Map, n);

return 0;

}

**Output**

20,30,40,1

**Complexity Analysis:**

* **Time complexity:**O(n^2).  
  Nested traversal is needed to store all pairs in the hash Map.
* **Auxiliary Space:** O(n^2).  
  All n\*(n-1) pairs are stored in hash Map so the space required is O(n^2) and the temp array takes O(n) so space comes to O(n^2).

**Question:**

Given an array of integers and another number. Find all the **unique**quadruple from the given array that sums up to the given number.

**Example 1:**

**Input:**

N = 5, K = 3

A[] = {0,0,2,1,1}

**Output:** 0 0 1 2 $

**Explanation:** Sum of 0, 0, 1, 2 is equal

to K.

**Example 2:**

**Input:**

N = 7, K = 23

A[] = {10,2,3,4,5,7,8}

**Output:** 2 3 8 10 $2 4 7 10 $3 5 7 8 $

**Explanation:** Sum of 2, 3, 8, 10 = 23,

sum of 2, 4, 7, 10 = 23 and sum of 3,

5, 7, 8 = 23.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **fourSum()** which takes the array arr[] and the integer k as its input and returns an array containing all the quadruples in a lexicographical manner. Also note that all the quadruples should be internally sorted, ie for any quadruple [q1, q2, q3, q4] the following should follow: q1 <= q2 <= q3 <= q4.  (In the output each quadruple is separate by $. The printing is done by the driver's code)

**Expected Time Complexity:** O(N3).  
**Expected Auxiliary Space:** O(N2).

**Constraints:**  
1 <= N <= 100  
-1000 <= K <= 1000  
-100 <= A[] <= 100

**Solution:**

vector<vector<int> > fourSum(vector<int> &arr, int k) {

int n = arr.size();

sort(arr.begin(),arr.end());

vector<vector<int>> res;

set<set<int>> st;

for(int i=0;i<n-3;i++){

for(int j=i+1;j<n-2;j++){

int req = k-arr[i]-arr[j];

int l=j+1, m=n-1;

while(l<m){

if(arr[l]+arr[m]<req)

l++;

else if(arr[l]+arr[m]>req)

m--;

else{

set<int> t;

t.insert(arr[i]);

t.insert(arr[j]);

t.insert(arr[l]);

t.insert(arr[m]);

if(st.find(t)==st.end()){

vector<int> temp;

temp.push\_back(arr[i]);

temp.push\_back(arr[j]);

temp.push\_back(arr[l]);

temp.push\_back(arr[m]);

res.push\_back(temp);

st.insert(t);

}

l++;

m--;

}

}

}

}

return res;

}

# Maximum sum such that no 2 elements are adjacent

Stickler the thief wants to loot money from a societyhaving**n** houses in a single line. He is a weird person and follows a certain **rule**when looting the houses. According to the rule, he will **never loot two consecutive houses**. At the same time, he wants to **maximize**the amount he **loots**. The thief knows which house has what amount of money but is unable to come up with an optimal looting strategy. He asks for your help to **find the maximum money he can get** if he strictly **follows**the **rule**. Each house has **a[i] amount of money** present in it.

**Example 1:**

**Input:**

n = 6

a[] = {5,5,10,100,10,5}

**Output:** 110

**Explanation:** 5+100+5=110

**Example 2:**

**Input:**

n = 3

a[] = {1,2,3}

**Output:** 4

**Explanation:** 1+3=4

**Your Task:**  
Complete the function **FindMaxSum()**which takes an array **arr[]** and **n** as input which returns the maximum money he can get following the rules

**Expected Time Complexity:** O(N).  
**Expected Space Complexity:** O(N).

**Constraints:**  
1 ≤ n ≤ 104  
1 ≤ a[i] ≤ 104

## Solution:

we can also solve this problem using the *Dynamic Programming* approach.  
**Dynamic Programming Approach:** Let’s decide the states of ‘dp’. Let dp[i] be the largest possible sum for the sub-array staring from index ‘i’ and ending at index ‘N-1’. Now, we have to find a recurrence relation between this state and a lower-order state.  
In this case for an index ‘i’, we will have two choices.

1) Choose the current index:

In this case, the relation will be dp[i] = arr[i] + dp[i+2]

2) Skip the current index:

Relation will be dp[i] = dp[i+1]

We will choose the path that maximizes our result.   
Thus, the final relation will be:

dp[i] = max(dp[i+2]+arr[i], dp[i+1])

Below is the implementation of the above approach:

// C++ program to implement above approach

#include <bits/stdc++.h>

#define maxLen 10

using namespace std;

// variable to store states of dp

int dp[maxLen];

// variable to check if a given state

// has been solved

bool v[maxLen];

// Function to find the maximum sum subsequence

// such that no two elements are adjacent

int maxSum(int arr[], int i, int n)

{

// Base case

if (i >= n)

return 0;

// To check if a state has

// been solved

if (v[i])

return dp[i];

v[i] = 1;

// Required recurrence relation

dp[i] = max(maxSum(arr, i + 1, n),

arr[i] + maxSum(arr, i + 2, n));

// Returning the value

return dp[i];

}

// Driver code

int main()

{

int arr[] = { 12, 9, 7, 33 };

int n = sizeof(arr) / sizeof(int);

cout << maxSum(arr, 0, n);

return 0;

}

**Output:**

45

**My code using similar approach:**

int FindMaxSum(int arr[], int n)

{

int dp[n], maxi[n];

dp[0] = maxi[0] = arr[0];

dp[1] = arr[1];

maxi[1] = max(arr[0],arr[1]);

for(int i=2;i<n;i++){

dp[i] = arr[i] + maxi[i-2];

maxi[i] = max(maxi[i-1],dp[i]);

}

return maxi[n-1];

}

**Time Complexity:** O(n)

**Space Complexity:** O(n)

**O(1) approach:**

**Algorithm:**   
Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.  
Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).  
At the end of the loop return max of incl and excl.

**Example:**

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

incl = 5

excl = 0

For i = 1 (current element is 5)

incl = (excl + arr[i]) = 5

excl = max(5, 0) = 5

For i = 2 (current element is 10)

incl = (excl + arr[i]) = 15

excl = max(5, 5) = 5

For i = 3 (current element is 40)

incl = (excl + arr[i]) = 45

excl = max(5, 15) = 15

For i = 4 (current element is 50)

incl = (excl + arr[i]) = 65

excl = max(45, 15) = 45

For i = 5 (current element is 35)

incl = (excl + arr[i]) = 80

excl = max(65, 45) = 65

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

**Implementation:**

//c++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

/\*Function to return max sum such that no two elements

are adjacent \*/

int FindMaxSum(vector<int> arr, int n)

{

int incl = arr[0];

int excl = 0;

int excl\_new;

int i;

for (i = 1; i < n; i++)

{

/\* current max excluding i \*/

excl\_new = (incl > excl) ? incl : excl;

/\* current max including i \*/

incl = excl + arr[i];

excl = excl\_new;

}

/\* return max of incl and excl \*/

return ((incl > excl) ? incl : excl);

}

// Driver program to test above functions

int main()

{

vector<int> arr = {5, 5, 10, 100, 10, 5};

cout<<FindMaxSum(arr, arr.size());

}

Output:

110

**Time Complexity:**O(n)

**Space Complexity:** O(1)

# Count triplet with sum smaller than a given value

Given an array **arr[]** of distinct integers of size **N** and a value **sum**, the task is to find the count of triplets **(i, j, k)**, having **(i<j<k)**with the sum of**(arr[i] + arr[j] + arr[k])**smaller than the given value sum.

**Example 1:**

**Input:** N = 4, sum = 2

arr[] = {-2, 0, 1, 3}

**Output:** 2

**Explanation:** Below are triplets with

sum less than 2 (-2, 0, 1) and (-2, 0, 3).

**Example 2:**

**Input:** N = 5, sum = 12

arr[] = {5, 1, 3, 4, 7}

**Output:** 4

**Explanation:** Below are triplets with

sum less than 12 (1, 3, 4), (1, 3, 5),

(1, 3, 7) and (1, 4, 5).

**Your Task:**  
This is a function problem. You don't need to take any input, as it is already accomplished by the driver code. You just need to complete the function countTriplets() that take array arr[], integer N  and integer sum as parameters and returns the count of triplets.

**Expected Time Complexity:**O(N2).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
3 ≤ N ≤ 103

-103 ≤ arr[i] ≤ 103

## Solution:

A **Simple Solution** is to run three loops to consider all triplets one by one. For every triplet, compare the sums and increment count if the triplet sum is smaller than the given sum.

// A Simple C++ program to count triplets with sum smaller

// than a given value

#include<bits/stdc++.h>

using namespace std;

int countTriplets(int arr[], int n, int sum)

{

// Initialize result

int ans = 0;

// Fix the first element as A[i]

for (int i = 0; i < n-2; i++)

{

// Fix the second element as A[j]

for (int j = i+1; j < n-1; j++)

{

// Now look for the third number

for (int k = j+1; k < n; k++)

if (arr[i] + arr[j] + arr[k] < sum)

ans++;

}

}

return ans;

}

// Driver program

int main()

{

int arr[] = {5, 1, 3, 4, 7};

int n = sizeof arr / sizeof arr[0];

int sum = 12;

cout << countTriplets(arr, n, sum) << endl;

return 0;

}

Output:

4

The time complexity of the above solution is O(n3).

An **Efficient Solution** can count triplets in O(n2) by sorting the array first, and then using method 1 of [this](https://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-two-elements-in-s-whose-sum-is-exactly-x/) post in a loop.

1) Sort the input array in increasing order.

2) Initialize result as 0.

3) Run a loop from i = 0 to n-2. An iteration of this loop finds all

triplets with arr[i] as first element.

a) Initialize other two elements as corner elements of subarray

arr[i+1..n-1], i.e., j = i+1 and k = n-1

b) Move j and k toward each other until they meet, i.e., while (j<k),

(i) If arr[i] + arr[j] + arr[k] >= sum

then k--

// Else for current i and j, there can (k-j) possible third elements

// that satisfy the constraint.

(ii) Else Do ans += (k - j) followed by j++

Below is the implementation of the above idea.

// C++ program to count triplets with sum smaller than a given value

#include<bits/stdc++.h>

using namespace std;

int countTriplets(int arr[], int n, int sum)

{

// Sort input array

sort(arr, arr+n);

// Initialize result

int ans = 0;

// Every iteration of loop counts triplet with

// first element as arr[i].

for (int i = 0; i < n - 2; i++)

{

// Initialize other two elements as corner elements

// of subarray arr[j+1..k]

int j = i + 1, k = n - 1;

// Use Meet in the Middle concept

while (j < k)

{

// If sum of current triplet is more or equal,

// move right corner to look for smaller values

if (arr[i] + arr[j] + arr[k] >= sum)

k--;

// Else move left corner

else

{

// This is important. For current i and j, there

// can be total k-j third elements.

ans += (k - j);

j++;

}

}

}

return ans;

}

// Driver program

int main()

{

int arr[] = {5, 1, 3, 4, 7};

int n = sizeof arr / sizeof arr[0];

int sum = 12;

cout << countTriplets(arr, n, sum) << endl;

return 0;

}

**Output:**

4

# merge 2 sorted arrays

## Same as question 12 of arrays.

# print all subarrays with 0 sum

Given an array, print all subarrays in the array which has sum 0.  
**Examples:** 

**Input:**  arr = [6, 3, -1, -3, 4, -2, 2,

4, 6, -12, -7]

**Output:**

Subarray found from Index 2 to 4

Subarray found from Index 2 to 6

Subarray found from Index 5 to 6

Subarray found from Index 6 to 9

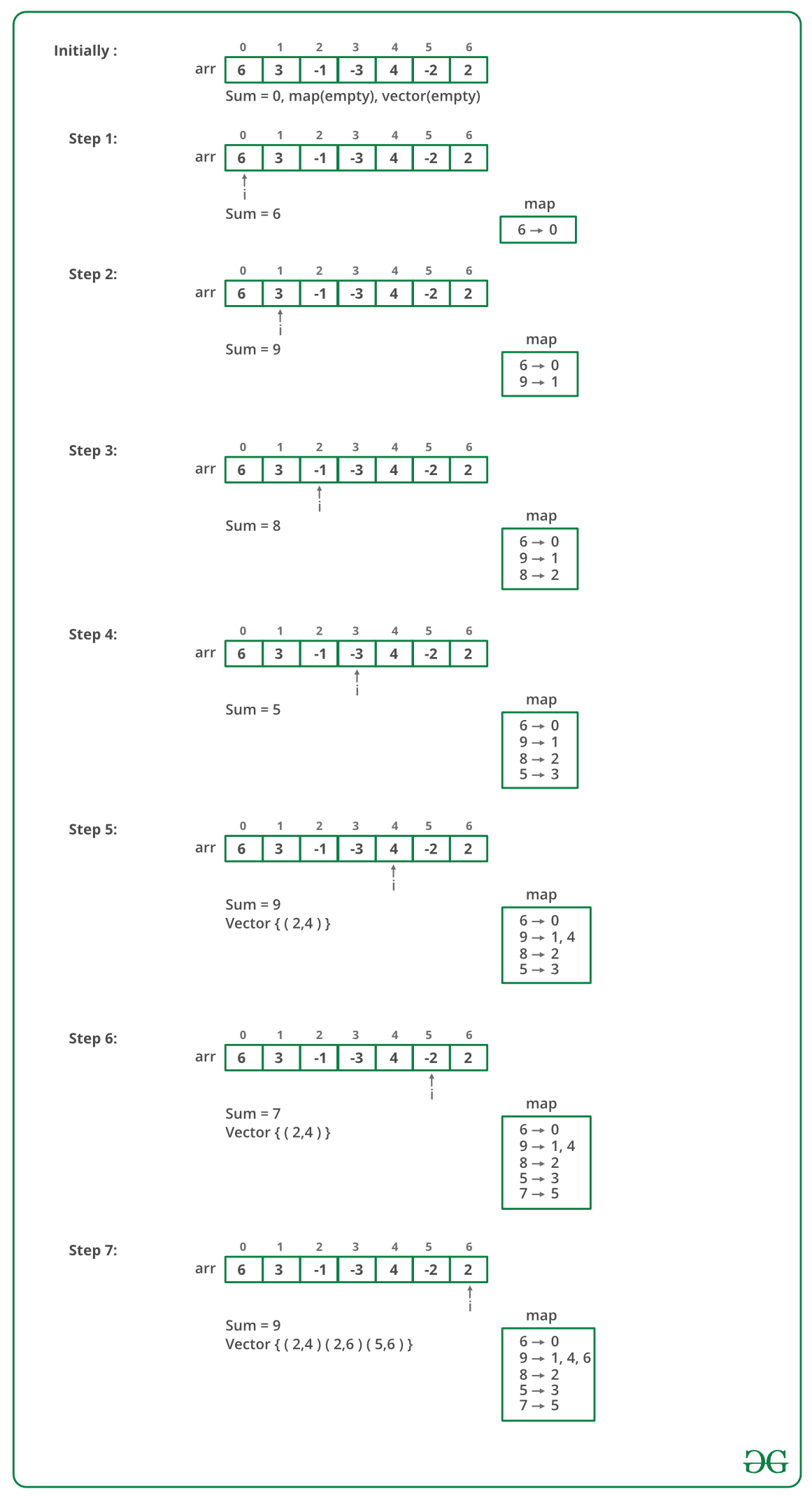
Subarray found from Index 0 to 1

## Solution:

A simple solution is to consider all subarrays one by one and check if sum of every subarray is equal to 0 or not. The complexity of this solution would be O(n^2).  
A better approach is to use Hashing.  
Do following for each element in the array 

1. Maintain sum of elements encountered so far in a variable (say sum).
2. If current sum is 0, we found a subarray starting from index 0 and ending at index current index
3. Check if current sum exists in the hash table or not.
4. If current sum already exists in the hash table then it indicates that this sum was the sum of some sub-array elements arr[0]…arr[i] and now the same sum is obtained for the current sub-array arr[0]…arr[j] which means that the sum of the sub-array arr[i+1]…arr[j] must be 0.
5. Insert current sum into the hash table

Below is a dry run of the above approach:



Below is the implementation of the above approach:

// C++ program to print all subarrays

// in the array which has sum 0

#include <bits/stdc++.h>

using namespace std;

// Function to print all subarrays in the array which

// has sum 0

vector< pair<int, int> > findSubArrays(int arr[], int n)

{

// create an empty map

unordered\_map<int, vector<int> > map;

// create an empty vector of pairs to store

// subarray starting and ending index

vector <pair<int, int>> out;

// Maintains sum of elements so far

int sum = 0;

for (int i = 0; i < n; i++)

{

// add current element to sum

sum += arr[i];

// if sum is 0, we found a subarray starting

// from index 0 and ending at index i

if (sum == 0)

out.push\_back(make\_pair(0, i));

// If sum already exists in the map there exists

// at-least one subarray ending at index i with

// 0 sum

if (map.find(sum) != map.end())

{

// map[sum] stores starting index of all subarrays

vector<int> vc = map[sum];

for (auto it = vc.begin(); it != vc.end(); it++)

out.push\_back(make\_pair(\*it + 1, i));

}

// Important - no else

map[sum].push\_back(i);

}

// return output vector

return out;

}

// Utility function to print all subarrays with sum 0

void print(vector<pair<int, int>> out)

{

for (auto it = out.begin(); it != out.end(); it++)

cout << "Subarray found from Index " <<

it->first << " to " << it->second << endl;

}

// Driver code

int main()

{

int arr[] = {6, 3, -1, -3, 4, -2, 2, 4, 6, -12, -7};

int n = sizeof(arr)/sizeof(arr[0]);

vector<pair<int, int> > out = findSubArrays(arr, n);

// if we didn’t find any subarray with 0 sum,

// then subarray doesn’t exists

if (out.size() == 0)

cout << "No subarray exists";

else

print(out);

return 0;

}

**Output:** 

Subarray found from Index 2 to 4

Subarray found from Index 2 to 6

Subarray found from Index 5 to 6

Subarray found from Index 6 to 9

Subarray found from Index 0 to 10

***Time Complexity:****O(N^2)*  
***Auxiliary Space:****O(N)*

**Question:**

You are given an array arr[] of size n. Find the total count of sub-arrays having their sum equal to 0.

**Example 1:**

**Input:**

n = 6

arr[] = {0,0,5,5,0,0}

**Output:** 6

**Explanation:** The 6 subarrays are

[0], [0], [0], [0], [0,0], and [0,0].

**Example 2:**

**Input:**

n = 10

arr[] = {6,-1,-3,4,-2,2,4,6,-12,-7}

**Output:** 4

**Explanation:** The 4 subarrays are [-1 -3 4]

[-2 2], [2 4 6 -12], and [-1 -3 4 -2 2]

**Your Task:**  
You don't need to read input or print anything. Complete thefunction **findSubarray()** that takes the array arr and its size n as input parametersand returns the total number of sub-arrays with 0 sum. 

**Expected Time Complexity** : O(n)  
**Expected Auxilliary Space** : O(n)

**Constraints:**  
1<= n <= 107  
-1010 <= arri <= 1010

**Solution:**

ll findSubarray(vector<ll> arr, int n ) {

//ll is already defined as long long

unordered\_map<ll,ll> mp;

ll sum=0,res=0;

for(int i=0;i<n;i++){

sum += arr[i];

if(sum==0)

res++;

if(mp.find(sum)!=mp.end())

res += mp[sum];

mp[sum]++;

}

return res;

}

# Product array Puzzle

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

**Example :**

**Input:** arr[] = {10, 3, 5, 6, 2}

**Output:** prod[] = {180, 600, 360, 300, 900}

3 \* 5 \* 6 \* 2 product of other array

elements except 10 is 180

10 \* 5 \* 6 \* 2 product of other array

elements except 3 is 600

10 \* 3 \* 6 \* 2 product of other array

elements except 5 is 360

10 \* 3 \* 5 \* 2 product of other array

elements except 6 is 300

10 \* 3 \* 6 \* 5 product of other array

elements except 2 is 900

**Input:** arr[] = {1, 2, 3, 4, 5}

**Output:** prod[] = {120, 60, 40, 30, 24 }

2 \* 3 \* 4 \* 5 product of other array

elements except 1 is 120

1 \* 3 \* 4 \* 5 product of other array

elements except 2 is 60

1 \* 2 \* 4 \* 5 product of other array

elements except 3 is 40

1 \* 2 \* 3 \* 5 product of other array

elements except 4 is 30

1 \* 2 \* 3 \* 4 product of other array

elements except 5 is 24

## Solution:

**Naive Solution:**  
**Approach:** Create two extra space, i.e. two extra arrays to store the product of all the array elements from start, up to that index and another array to store the product of all the array elements from the end of the array to that index.   
To get the product excluding that index, multiply the prefix product up to index i-1 with the suffix product up to index i+1.

**Algorithm:**

1. Create two array *prefix* and *suffix* of length *n*, i.e length of the original array, initialize *prefix[0] = 1* and *suffix[n-1] = 1* and also another array to store the product.
2. Traverse the array from second index to end.
3. For every index *i* update *prefix[i]* as *prefix[i] = prefix[i-1] \* array[i-1]*, i.e store the product upto *i-1* index from the start of array.
4. Traverse the array from second last index to start.
5. For every index *i* update *suffix[i]* as *suffix[i] = suffix[i+1] \* array[i+1]*, i.e store the product upto *i+1* index from the end of array
6. Traverse the array from start to end.
7. For every index *i* the output will be *prefix[i] \* suffix[i]*, the product of the array element except that element.

// C++ implementation of above approach

#include <bits/stdc++.h>

using namespace std;

/\* Function to print product array

for a given array arr[] of size n \*/

void productArray(int arr[], int n)

{

// Base case

if (n == 1) {

cout << 0;

return;

}

/\* Allocate memory for temporary

arrays left[] and right[] \*/

int\* left = new int[sizeof(int) \* n];

int\* right = new int[sizeof(int) \* n];

/\* Allocate memory for the product array \*/

int\* prod = new int[sizeof(int) \* n];

int i, j;

/\* Left most element of left

array is always 1 \*/

left[0] = 1;

/\* Rightmost most element of right

array is always 1 \*/

right[n - 1] = 1;

/\* Construct the left array \*/

for (i = 1; i < n; i++)

left[i] = arr[i - 1] \* left[i - 1];

/\* Construct the right array \*/

for (j = n - 2; j >= 0; j--)

right[j] = arr[j + 1] \* right[j + 1];

/\* Construct the product array using

left[] and right[] \*/

for (i = 0; i < n; i++)

prod[i] = left[i] \* right[i];

/\* print the constructed prod array \*/

for (i = 0; i < n; i++)

cout << prod[i] << " ";

return;

}

/\* Driver code\*/

int main()

{

int arr[] = { 10, 3, 5, 6, 2 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "The product array is: \n";

productArray(arr, n);

}

**Output**

The product array is:

180 600 360 300 900

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  The array needs to be traversed three times, so the time complexity is O(n).
* **Space Complexity:** O(n).   
  Two extra arrays and one array to store the output is needed so the space complexity is O(n)

**Efficient solution:**  
**Approach:** In the previous solution, two extra arrays were created to store the prefix and suffix, in this solution store the prefix and suffix product in the output array (or product array) itself. Thus reducing the space required.

**Algorithm:**

1. Create an array *product* and initialize its value to 1 and a variable *temp* = 1.
2. Traverse the array from start to end.
3. For every index *i* update *product[i]* as *product[i] = temp* and *temp = temp \* array[i]*, i.e store the product upto *i-1* index from the start of array.
4. initialize temp = 1 and traverse the array from last index to start.
5. For every index *i* update *product[i]* as *product[i] = product[i] \* temp* and *temp = temp \* array[i]*, i.e multiply with the product upto *i+1* index from the end of array.
6. Print the product array.

// C++ implementation of above approach

#include <bits/stdc++.h>

using namespace std;

/\* Function to print product array

for a given array arr[] of size n \*/

void productArray(int arr[], int n)

{

// Base case

if (n == 1) {

cout << 0;

return;

}

int i, temp = 1;

/\* Allocate memory for the product array \*/

int\* prod = new int[(sizeof(int) \* n)];

/\* Initialize the product array as 1 \*/

memset(prod, 1, n);

/\* In this loop, temp variable contains product of

elements on left side excluding arr[i] \*/

for (i = 0; i < n; i++) {

prod[i] = temp;

temp \*= arr[i];

}

/\* Initialize temp to 1

for product on right side \*/

temp = 1;

/\* In this loop, temp variable contains product of

elements on right side excluding arr[i] \*/

for (i = n - 1; i >= 0; i--) {

prod[i] \*= temp;

temp \*= arr[i];

}

/\* print the constructed prod array \*/

for (i = 0; i < n; i++)

cout << prod[i] << " ";

return;

}

// Driver Code

int main()

{

int arr[] = { 10, 3, 5, 6, 2 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "The product array is: \n";

productArray(arr, n);

}

**Output**

The product array is:

180 600 360 300 900

**Complexity Analysis:**

* **Time Complexity:**O(n).   
  The original array needs to be traversed only once, so the time complexity is constant.
* **Space Complexity:**O(n).   
  Even though the extra arrays are removed, the space complexity remains O(n), as the product array is still needed.

**Another Approach:**

Store the product of all the elements is a variable and then iterate the array and add product/current\_index\_value in a new array. and then return this new array.

Below is the implementation of the above approach:

// C++ program for the above approach

#include <iostream>

using namespace std;

long\* productExceptSelf(int a[], int n)

{

long prod = 1;

long flag = 0;

// product of all elements

for (int i = 0; i < n; i++) {

// counting number of elements

// which have value

// 0

if (a[i] == 0)

flag++;

else

prod \*= a[i];

}

// creating a new array of size n

long\* arr = new long[n];

for (int i = 0; i < n; i++) {

// if number of elements in

// array with value 0

// is more than 1 than each

// value in new array

// will be equal to 0

if (flag > 1) {

arr[i] = 0;

}

// if no element having value

// 0 than we will

// insert product/a[i] in new array

else if (flag == 0)

arr[i] = (prod / a[i]);

// if 1 element of array having

// value 0 than all

// the elements except that index

// value , will be

// equal to 0

else if (flag == 1 && a[i] != 0) {

arr[i] = 0;

}

// if(flag == 1 && a[i] == 0)

else

arr[i] = prod;

}

return arr;

}

// Driver Code

int main()

{

int n = 5;

int array[] = { 10, 3, 5, 6, 2 };

long\* ans;

ans = productExceptSelf(array, n);

for (int i = 0; i < n; i++) {

cout << ans[i] << " ";

}

// cout<<"GFG!";

return 0;

}

**Output**

180 600 360 300 900

**Time Complexity: O(n)**

**Space Complexity: O(1)**

# Sort array according to count of set bits

Given an array of positive integers, sort the array in decreasing order of count of set bits in binary representations of array elements. For integers having the same number of set bits in their binary representation, sort according to their position in the original array i.e., a stable sort. For example, if the input array is {3, 5}, then the output array should also be {3, 5}. Note that both 3 and 5 have the same number set bits.

**Examples:**

**Input:** arr[] = {5, 2, 3, 9, 4, 6, 7, 15, 32};

**Output:** 15 7 5 3 9 6 2 4 32

**Explanation:**

The integers in their binary representation are:

15 -1111

7 -0111

5 -0101

3 -0011

9 -1001

6 -0110

2 -0010

4- -0100

32 -10000

hence the non-increasing sorted order is:

{15}, {7}, {5, 3, 9, 6}, {2, 4, 32}

**Input:** arr[] = {1, 2, 3, 4, 5, 6};

**Output:** 3 5 6 1 2 4

**Explanation:**

3 - 0011

5 - 0101

6 - 0110

1 - 0001

2 - 0010

4 - 0100

hence the non-increasing sorted order is

{3, 5, 6}, {1, 2, 4}

## Solution:

**Method 1: Simple**

1. Create an auxiliary array and store the set-bit counts of all integers in the aux array
2. Simultaneously sort both arrays according to the non-increasing order of auxiliary array. (Note that we need to use a stable sort algorithm)

**Before sort:**

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

int aux[] = {1, 1, 2, 1, 2, 2}

**After sort:**

arr = {3, 5, 6, 1, 2, 4}

aux = {2, 2, 2, 1, 1, 1}

**Implementation:**

// C++ program to implement simple approach to sort

// an array according to count of set bits.

#include <iostream>

using namespace std;

// a utility function that returns total set bits

// count in an integer

int countBits(int a)

{

int count = 0;

while (a)

{

if (a & 1 )

count+= 1;

a = a>>1;

}

return count;

}

// Function to simultaneously sort both arrays

// using insertion sort

// ( https://www.geeksforgeeks.org/insertion-sort/ )

void insertionSort(int arr[],int aux[], int n)

{

for (int i = 1; i < n; i++)

{

// use 2 keys because we need to sort both

// arrays simultaneously

int key1 = aux[i];

int key2 = arr[i];

int j = i-1;

/\* Move elements of arr[0..i-1] and aux[0..i-1],

such that elements of aux[0..i-1] are

greater than key1, to one position ahead

of their current position \*/

while (j >= 0 && aux[j] < key1)

{

aux[j+1] = aux[j];

arr[j+1] = arr[j];

j = j-1;

}

aux[j+1] = key1;

arr[j+1] = key2;

}

}

// Function to sort according to bit count using

// an auxiliary array

void sortBySetBitCount(int arr[],int n)

{

// Create an array and store count of

// set bits in it.

int aux[n];

for (int i=0; i<n; i++)

aux[i] = countBits(arr[i]);

// Sort arr[] according to values in aux[]

insertionSort(arr, aux, n);

}

// Utility function to print an array

void printArr(int arr[], int n)

{

for (int i=0; i<n; i++)

cout << arr[i] << " ";

}

// Driver Code

int main()

{

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

int n = sizeof(arr)/sizeof(arr[0]);

sortBySetBitCount(arr, n);

printArr(arr, n);

return 0;

}

**Output:**

3 5 6 1 2 4

**Auxiliary Space:** O(n)  
**Time complexity:** O(n2)  
**Note:** Time complexity can be improved to O(nLogn) by using a stable O(nlogn) sorting algorithm.

**Method 2: Using**[**std::sort()**](https://www.geeksforgeeks.org/sort-c-stl/)

Using custom comparator of std::sort to sort the array according to set-bit count

// C++ program to sort an array according to

// count of set bits using std::sort()

#include <bits/stdc++.h>

using namespace std;

// a utility function that returns total set bits

// count in an integer

int countBits(int a)

{

int count = 0;

while (a) {

if (a & 1)

count += 1;

a = a >> 1;

}

return count;

}

// custom comparator of std::sort

int cmp(int a, int b)

{

int count1 = countBits(a);

int count2 = countBits(b);

// this takes care of the stability of

// sorting algorithm too

if (count1 <= count2)

return false;

return true;

}

// Function to sort according to bit count using

// std::sort

void sortBySetBitCount(int arr[], int n)

{

stable\_sort(arr, arr + n, cmp);

}

// Utility function to print an array

void printArr(int arr[], int n)

{

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

}

// Driver Code

int main()

{

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

int n = sizeof(arr) / sizeof(arr[0]);

sortBySetBitCount(arr, n);

printArr(arr, n);

return 0;

}

**Output:**

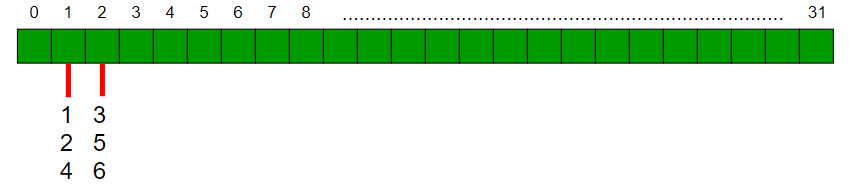
3 5 6 1 2 4

**Auxiliary Space:** O(1)  
**Time complexity:** O(n log n)

**Method 3:**[**Counting Sort**](https://www.geeksforgeeks.org/counting-sort/)**based**

This problem can be solved in O(n) time. The idea is similar to counting sort.  
**Note:** There can be a minimum 1 set-bit and only a maximum of 31set-bits in an integer.  
**Steps (assuming that an integer takes 32 bits):**

1. Create a vector “count” of size 32. Each cell of count i.e., count[i] is another vector that stores all the elements whose set-bit-count is i
2. Traverse the array and do the following for each element:
   1. Count the number set-bits of this element. Let it be ‘setbitcount’
   2. count[setbitcount].push\_back(element)
3. Traverse ‘count’ in reverse fashion(as we need to sort in non-increasing order) and modify the array.



// C++ program to sort an array according to

// count of set bits using std::sort()

#include <bits/stdc++.h>

using namespace std;

// a utility function that returns total set bits

// count in an integer

int countBits(int a)

{

int count = 0;

while (a)

{

if (a & 1 )

count+= 1;

a = a>>1;

}

return count;

}

// Function to sort according to bit count

// This function assumes that there are 32

// bits in an integer.

void sortBySetBitCount(int arr[],int n)

{

vector<vector<int> > count(32);

int setbitcount = 0;

for (int i=0; i<n; i++)

{

setbitcount = countBits(arr[i]);

count[setbitcount].push\_back(arr[i]);

}

int j = 0; // Used as an index in final sorted array

// Traverse through all bit counts (Note that we

// sort array in decreasing order)

for (int i=31; i>=0; i--)

{

vector<int> v1 = count[i];

for (int i=0; i<v1.size(); i++)

arr[j++] = v1[i];

}

}

// Utility function to print an array

void printArr(int arr[], int n)

{

for (int i=0; i<n; i++)

cout << arr[i] << " ";

}

// Driver Code

int main()

{

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

int n = sizeof(arr)/sizeof(arr[0]);

sortBySetBitCount(arr, n);

printArr(arr, n);

return 0;

}

**Output:**

3 5 6 1 2 4

**Method 4: Using MultiMap**  
**Steps:**

* Create a MultiMap whose key values will be the negative of the number of set-bits of the element.
* Traverse the array and do following for each element:
  + Count the number set-bits of this element. Let it be ‘setBitCount’
  + count.insert({(-1) \* setBitCount, element})
* Traverse ‘count’ and print the second elements.

Below is the implementation of the above approach:

// C++ program to implement

// simple approach to sort

// an array according to

// count of set bits.

#include<bits/stdc++.h>

using namespace std;

// Function to count setbits

int setBitCount(int num){

int count = 0;

while ( num )

{

if ( num & 1)

count++;

num >>= 1;

}

return count;

}

// Function to sort By SetBitCount

void sortBySetBitCount(int arr[], int n)

{

multimap< int, int > count;

// Iterate over all values and

// insert into multimap

for( int i = 0 ; i < n ; ++i )

{

count.insert({(-1) \*

setBitCount(arr[i]), arr[i]});

}

for(auto i : count)

cout << i.second << " " ;

cout << "\n" ;

}

// Driver Code

int main()

{

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

int n = sizeof(arr)/sizeof(arr[0]);

sortBySetBitCount(arr, n);

}

**Output:**

3 5 6 1 2 4

**Time complexity:** O(n log n)

**Auxiliary Space:**O(n)

# Minimum no. of swaps required to sort the array

Given an array of n distinct elements. Find the minimum number of swaps required to sort the array in strictly increasing order.

**Example 1:**

**Input:**

nums = {2, 8, 5, 4}

**Output:**

1

**Explaination:**

swap 8 with 4.

**Example 2:**

**Input:**

nums = {10, 19, 6, 3, 5}

**Output:**

2

**Explaination:**

swap 10 with 3 and swap 19 with 5.

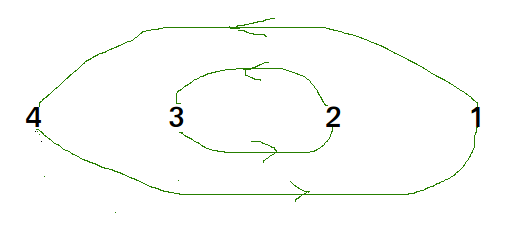
**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **minSwaps()**which takes the **nums**as input parameter and returns an integer denoting the minimum number of swaps required to sort the array. If the array is already sorted, return 0.

**Expected Time Complexity:** O(nlogn)  
**Expected Auxiliary Space:**O(n)

**Constraints:**  
1 ≤ n ≤ 105  
1 ≤ numsi ≤ 106

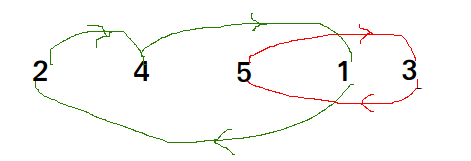
## Solution:

This can be easily done by visualizing the problem as a graph. We will have **n** nodes and an edge directed from node **i** to node **j** if the element at i’th index must be present at j’th index in the sorted array. 



Graph for {4, 3, 2, 1}

The graph will now contain many non-intersecting cycles. Now a cycle with 2 nodes will only require 1 swap to reach the correct ordering, similarly, a cycle with 3 nodes will only require 2 swaps to do so.



Graph for {4, 5, 2, 1, 5}

Hence,

* ans = **Σi = 1k**(cycle\_size – 1)

where, **k** is the number of cycles

Below is the implementation of the idea.

// C++ program to find

// minimum number of swaps

// required to sort an array

#include<bits/stdc++.h>

using namespace std;

// Function returns the

// minimum number of swaps

// required to sort the array

int minSwaps(int arr[], int n)

{

// Create an array of

// pairs where first

// element is array element

// and second element

// is position of first element

pair<int, int> arrPos[n];

for (int i = 0; i < n; i++)

{

arrPos[i].first = arr[i];

arrPos[i].second = i;

}

// Sort the array by array

// element values to

// get right position of

// every element as second

// element of pair.

sort(arrPos, arrPos + n);

// To keep track of visited elements.

// Initialize

// all elements as not visited or false.

vector<bool> vis(n, false);

// Initialize result

int ans = 0;

// Traverse array elements

for (int i = 0; i < n; i++)

{

// already swapped and corrected or

// already present at correct pos

if (vis[i] || arrPos[i].second == i)

continue;

// find out the number of node in

// this cycle and add in ans

int cycle\_size = 0;

int j = i;

while (!vis[j])

{

vis[j] = 1;

// move to next node

j = arrPos[j].second;

cycle\_size++;

}

// Update answer by adding current cycle.

if (cycle\_size > 0)

{

ans += (cycle\_size - 1);

}

}

// Return result

return ans;

}

// Driver program to test the above function

int main()

{

int arr[] = {1, 5, 4, 3, 2};

int n = (sizeof(arr) / sizeof(int));

cout << minSwaps(arr, n);

return 0;

}

**Output:**

2

**Time Complexity:** O(n Log n)   
**Auxiliary Space:** O(n)

**My implementation using similar approach:**

int minSwaps(vector<int>&nums)

{

int n = nums.size();

vector<int> sorted(nums);

sort(sorted.begin(),sorted.end());

vector<int> target\_loc(n);

for(int i=0;i<n;i++){

int p=0, q=n-1;

while(p<=q){

int m = p+(q-p)/2;

if(sorted[m]==nums[i]){

target\_loc[i] = m;

break;

}

else if(sorted[m]<nums[i])

p = m+1;

else

q = m-1;

}

}

int swaps=0;

for(int i=0;i<n;i++){

if(target\_loc[i]>i){

int t = target\_loc[i], len=1;

while(t>i){

t = target\_loc[t];

len++;

}

if(t==i)

swaps += (len-1);

}

}

return swaps;

}

**Approach:** As Pair class available in java from java 8 so we can use hashmap in older java version.

Below is the implementation of the idea.

import java.util.\*;

import java.io.\*;

class GfG

{

// Function returns the

// minimum number of swaps

// required to sort the array

public static int minSwaps(int[] nums)

{

int len = nums.length;

HashMap<Integer, Integer> map = new HashMap<>();

for(int i=0;i<len;i++)

map.put(nums[i], i);

Arrays.sort(nums);

// To keep track of visited elements. Initialize

// all elements as not visited or false.

boolean[] visited = new boolean[len];

Arrays.fill(visited, false);

// Initialize result

int ans = 0;

for(int i=0;i<len;i++) {

// already swapped and corrected or

// already present at correct pos

if(visited[i] || map.get(nums[i]) == i)

continue;

int j = i, cycle\_size = 0;

while(!visited[j]) {

visited[j] = true;

// move to next node

j = map.get(nums[j]);

cycle\_size++;

}

// Update answer by adding current cycle.

if(cycle\_size > 0) {

ans += (cycle\_size - 1);

}

}

return ans;

}

}

// Driver class

class MinSwaps

{

// Driver program to test the above function

public static void main(String[] args)

{

int []a = {1, 5, 4, 3, 2};

GfG g = new GfG();

System.out.println(g.minSwaps(a));

}

}

**Time Complexity**: O(n Log n)

**Auxiliary Space**: O(n)

**Straight forward solution**

While iterating over the array, check the current element, and if not in the correct place, replace that element with the index of the element which should have come in this place.

Below is the implementation of the above approach:

// C++ program to find minimum number

// of swaps required to sort an array

#include <bits/stdc++.h>

using namespace std;

void swap(vector<int> &arr, int i, int j)

{

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

int indexOf(vector<int> &arr, int ele)

{

for(int i = 0; i < arr.size(); i++)

{

if (arr[i] == ele)

{

return i;

}

}

return -1;

}

// Return the minimum number

// of swaps required to sort the array

int minSwaps(vector<int> arr, int N)

{

int ans = 0;

vector<int> temp(arr.begin(),arr.end());

sort(temp.begin(),temp.end());

for(int i = 0; i < N; i++)

{

// This is checking whether

// the current element is

// at the right place or not

if (arr[i] != temp[i])

{

ans++;

// Swap the current element

// with the right index

// so that arr[0] to arr[i] is sorted

swap(arr, i, indexOf(arr, temp[i]));

}

}

return ans;

}

// Driver Code

int main()

{

vector<int> a = {101, 758, 315, 730,

472, 619, 460, 479};

int n = a.size();

// Output will be 5

cout << minSwaps(a, n);

}

**Output:**

**5**

**Time Complexity:**O(n\*n)   
**Auxiliary Space:**O(n)

We can still improve the complexity by using a hashmap. The main operation here is the indexOf method inside the loop, which costs us n\*n. We can improve this section to O(n), by using a hashmap to store the indexes. Still, we use the sort method, so the complexity cannot improve beyond O(n Log n)

**Method using HashMap:**

Same as before, make a new array (called temp), which is the sorted form of the input array. We know that we need to transform the input array to the new array (temp) in the minimum number of swaps. Make a map that stores the elements and their corresponding index, of the input array.

So at each i starting from 0 to N in the given array, where N is the size of the array:

1. If i is not in its correct position according to the sorted array, then

2. We will fill this position with the correct element from the hashmap we built earlier. We know the correct element which should come here is temp[i], so we look up the index of this element from the hashmap.

3. After swapping the required elements, we update the content of the hashmap accordingly, as temp[i] to the ith position, and arr[i] to where temp[i] was earlier.

Below is the implementation of the above approach:

// C++ program to find

// minimum number of swaps

// required to sort an array

#include<bits/stdc++.h>

using namespace std;

void swap(vector<int> &arr,

int i, int j)

{

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

// Return the minimum number

// of swaps required to sort

// the array

int minSwaps(vector<int>arr,

int N)

{

int ans = 0;

vector<int>temp = arr;

// Hashmap which stores the

// indexes of the input array

map <int, int> h;

sort(temp.begin(), temp.end());

for (int i = 0; i < N; i++)

{

h[arr[i]] = i;

}

for (int i = 0; i < N; i++)

{

// This is checking whether

// the current element is

// at the right place or not

if (arr[i] != temp[i])

{

ans++;

int init = arr[i];

// If not, swap this element

// with the index of the

// element which should come here

swap(arr, i, h[temp[i]]);

// Update the indexes in

// the hashmap accordingly

h[init] = h[temp[i]];

h[temp[i]] = i;

}

}

return ans;

}

// Driver class

int main()

{

// Driver program to

// test the above function

vector <int> a = {101, 758, 315,

730, 472, 619,

460, 479};

int n = a.size();

// Output will be 5

cout << minSwaps(a, n);

}

**Output:**

5

**Time Complexity:** O(n Log n)   
**Auxiliary Space:** O(n)

**Approach:**   
The idea is to create a [vector of pair in C++](https://www.geeksforgeeks.org/sorting-vector-of-pairs-in-c-set-1-sort-by-first-and-second/) with first element as array values and second element as array indices. The next step is to sort the vector of pair according to the first element of the pair. After that traverse the vector and check if the index mapped with the value is correct or not, if not then keep swapping until the element is placed correctly and keep counting the number of swaps. **Algorithm:**

1. Create a vector of pairs and traverse the array and for every element of the array insert a element-index pair in the vector
2. Traverse the vector from start to the end (loop counter is i).
3. For every element of the pair where the second element(index) is not equal to i. Swap the ith element of the vector with the second element(index) th element of the vector
4. If the second element(index) is equal to i then skip the iteration of the loop.
5. if after the swap the second element(index) is not equal to i then decrement i.
6. Increment the counter.

**Implementation:**

// C++ program to find the minimum number

// of swaps required to sort an array

// of distinct element

#include<bits/stdc++.h>

using namespace std;

// Function to find minimum swaps to

// sort an array

int findMinSwap(int arr[] , int n)

{

// Declare a vector of pair

vector<pair<int,int>> vec(n);

for(int i=0;i<n;i++)

{

vec[i].first=arr[i];

vec[i].second=i;

}

// Sort the vector w.r.t the first

// element of pair

sort(vec.begin(),vec.end());

int ans=0,c=0,j;

for(int i=0;i<n;i++)

{

// If the element is already placed

// correct, then continue

if(vec[i].second==i)

continue;

else

{

// swap with its respective index

swap(vec[i].first,vec[vec[i].second].first);

swap(vec[i].second,vec[vec[i].second].second);

}

// swap until the correct

// index matches

if(i!=vec[i].second)

--i;

// each swap makes one element

// move to its correct index,

// so increment answer

ans++;

}

return ans;

}

// Driver code

int main()

{

int arr[] = {1, 5, 4, 3, 2};

int n = sizeof(arr)/sizeof(arr[0]);

cout<<findMinSwap(arr,n);

return 0;

}

**Output:**

2

**Complexity Analysis:**

* **Time Complexity:** O(n Log n).   
  Time required to sort the array is n log n.
* **Auxiliary Space:** O(n).   
  An extra array or vector is created. So, the space complexity is O(n )

# Bishu and Soldiers

**Problem**

Bishu went to fight for Coding Club. There were N soldiers with various powers. There will be Q rounds to fight and in each round, Bishu's power will be varied. With power M, Bishu can kill all the soldiers whose power is less than or equal to M(<=M). After each round, All the soldiers who are dead in the previous round will reborn. Such that in each round there will be N soldiers to fight. As Bishu is weak in mathematics, help him to count the number of soldiers that he can kill in each round and the total sum of their powers.

**INPUT:**

The first line of the input contains N, the number of soldiers.

The second line contains an array of N numbers denoting the power of each soldier

This third line contains Q, which denotes the number of rounds.

Q lines follow, each line having one number denoting the power of Bishu.

**OUTPUT:**

For each round, the output should be an array of two numbers. The first number should be the number of soldiers that Bishu can beat, and the second number denotes the cumulative strength of all the soldiers that Bishu can beat.

**CONSTRAINTS:**

1<=N<=10000

1<=power of each soldier<=100

1<=Q<=10000

1<=power of bishu<=100

**Sample Input**

7

1 2 3 4 5 6 7

3

3

10

2

**Sample Output**

3 6

7 28

2 3

Time Limit: 1

Memory Limit: 256

Source Limit:

**Explanation**

There are 7 soldiers, each with the indicated power.

There are 3 rounds.

In the first round, Bishu's power is 3, which is greater than 3 soldiers, with the cumulative power of 6. Hence, the answer is 3 6

Similarly for the next round.

## Solution:

**Approach1:**

Sort the power of soldiers and apply binary search for each round.

#include <bits/stdc++.h>

using namespace std;

int main() {

    int sol;

    cin>>sol;

    int powSol[sol];

    for(int i=0;i<sol;i++)

        cin>>powSol[i];

    int round;

    cin>>round;

    int powBishu[round];

    for(int i=0;i<round;i++)

        cin>>powBishu[i];

    sort(powSol,powSol+sol);

    long long cummPowSol[sol];

    cummPowSol[0] = powSol[0];

    for(int i=1;i<sol;i++)

        cummPowSol[i] = cummPowSol[i-1] + powSol[i];

    for(int i=0;i<round;i++){

        int num = powBishu[i];

        int p=0, q=sol-1, mid, ans=-1;

        while(p<=q){

            mid = p+(q-p)/2;

            if(powSol[mid]<=num){

                ans = mid;

                p = mid+1;

            }

            else

                q = mid-1;

        }

        cout<<ans+1<<" "<<cummPowSol[ans]<<endl;

    }

}

**Time Complexity:** O(nlogn)

**Space Complexity:** O(n)

**Using upper bound of STL:**

#include <bits/stdc++.h>

using namespace std;

#define ll long long

#define pb push\_back

#define vi vector<int>

#define vll vector<ll>

#define vs vector<string>

#define vc vector<char>

#define xag ios\_base::sync\_with\_stdio(false);cin.tie(0);cout.tie(0);

int main()

{

int n;

cin>>n;

int a[n];

for(int i=0;i<n;i++)cin>>a[i];

sort(a,a+n);

int p;

int q;cin>>q;

while(q--)

{

int cs=0;

cin>>p;

int x=upper\_bound(a,a+n,p)-a;

for(int i=0;i<x;i++)

{

cs+=a[i];

}

cout<<x<<" "<<cs<<endl;

}

return 0;

}

# Rasta and Kheshtak

Rasta is a big fan of Kheshtaks. A Kheshtak is a rectangle that in each of it cells there is an integer.

Today rasta came up with an interesting problem, Biggest Common Subsquare (BCS). A Kheshtak is called a Square if the number of its columns is equal to the number of its rows. A Square S is called a subsqaue of a Kheshtak A if and only if we can turn A to S by deleting some of its rows and some of its columns (maybe none).

He gives you two Kheshtaks, A and B (A one is n × m and B is x × y).

Input format

The first line of input contains n and m.

Then you are given A (n lines, in each line there are m space separated integers).

After that you are given x and y.

Then you are given B (x lines, in each line there are y space separated integers).

1 ≤ n, m, x, y ≤ 700 and all numbers in A and B are integers in the interval [1, 1000].

Output format

Print the size of BCS of A and B in a single line (size of a Square is number of its rows).

**Sample Input**

3 3

1 2 0

1 2 1

1 2 3

3 3

0 1 2

1 1 2

3 1 2

**Sample Output**

2

Time Limit: 2

Memory Limit: 256

## Solution:

**Approach 1:**

import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
public class problem3 {  
  
    /\*\*  
     \* @param args  
     \*/  
    static int[][] matrix1;  
    static int[][] matrix2;  
    static int comparison = 0;  
  
    public static void main(String[] args) throws IOException {  
        // TODO Auto-generated method stub  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
        String input = br.readLine();  
        String[] parts = input.split(" ");  
        int n = Integer.parseInt(parts[0]);  
        int m = Integer.parseInt(parts[1]);  
        matrix1 = new int[n][m];  
        for (int i = 0; i < n; i++) {  
            input = br.readLine();  
            parts = input.split(" ");  
            for (int j = 0; j < m; j++) {  
                matrix1[i][j] = Integer.parseInt(parts[j]);  
            }  
        }  
        input = br.readLine();  
        parts = input.split(" ");  
        int x = Integer.parseInt(parts[0]);  
        int y = Integer.parseInt(parts[1]);  
        matrix2 = new int[x][y];  
        for (int i = 0; i < x; i++) {  
            input = br.readLine();  
            parts = input.split(" ");  
            for (int j = 0; j < y; j++) {  
                matrix2[i][j] = Integer.parseInt(parts[j]);  
            }  
        }  
        int min = m;  
        if (n < m) {  
            min = n;  
        }  
        if (x < min) {  
            min = x;  
        }  
        if (y < min) {  
            min = y;  
        }  
//        boolean ansSet = false;  
//        for (int i = min; i > 0; i--) {  
//            if (CompareAllSquaresOfGivenSize(i, n, m, x, y)) {  
//                System.out.println(i);  
//                ansSet = true;  
//                break;  
//            }  
//        }  
//        if (!ansSet)  
//            System.out.println(0);  
         
        System.out.println(Calc(0,min,n,m,x,y,min));  
    }  
  
    public static int Calc(int start,int end,int n,int m, int x, int y,int maxSquareSide)  
    {  
        if(start==0 && end==0)  
        {  
            return 0;  
        }  
            int mid=(start+end)/2;  
            if(CompareAllSquaresOfGivenSize(mid, n,m,x,y))  
            {  
                if(!CompareAllSquaresOfGivenSize(mid+1, n,m,x,y))  
                {  
                    return mid;  
                }  
                if(mid+1<=maxSquareSide)  
                    return Calc(mid+1,end,n,m,x,y,maxSquareSide);  
                else  
                    return mid;  
            }  
            else  
            {  
                return Calc(start,mid-1,n,m,x,y,maxSquareSide);  
            }  
     
    }  
  
    public static boolean CompareAllSquaresOfGivenSize(int size, int n, int m,  
            int x, int y) {  
        for (int startingPointRow1 = 0; startingPointRow1 + size - 1 < n; startingPointRow1++) {  
            for (int startingPointColum1 = 0; startingPointColum1 + size - 1 < m; startingPointColum1++) {  
                for (int startingPointRow2 = 0; startingPointRow2 + size - 1 < x; startingPointRow2++) {  
                    for (int startingPointColum2 = 0; startingPointColum2  
                            + size - 1 < y; startingPointColum2++) {  
                        // System.out.println("matrix1row: "+startingPointRow1 +  
                        // " matrix1Col: " + startingPointColum1 +  
                        // " matrix2row: "+startingPointRow2 +  
                        // " matrix2Col: " + startingPointColum2  
                        // + " size: "+size);  
                        if (compareSquares(startingPointRow1,  
                                startingPointColum1, startingPointRow2,  
                                startingPointColum2, size))  
                            return true;  
                    }  
                }  
            }  
        }  
        return false;  
    }  
  
    public static boolean compareSquares(int square1Row, int square1Column,  
            int square2Row, int square2Column, int SquareLength) {  
        comparison++;  
        for (int i = 0; i < SquareLength; i++) {  
            for (int j = 0; j < SquareLength; j++) {  
                if (matrix1[square1Row + i][square1Column + j] != matrix2[square2Row  
                        + i][square2Column + j]) {  
                    return false;  
                }  
            }  
        }  
        return true;  
    }  
  
}

**Approach 2:**

First observation we should make to be able to solve this problem: if two matrices have common subsquare of size *X*, they also have common subsquares of sizes *X-1*, *X-2*, *X-3*,... ,*1* (you may simply take part of large subsquare).

This observation should make idea of using **binary search** here quite obvious. Now we need to learn how to check that given matrices have common subsquare of size *X*.

Quite a few of tests are small&random enough to give you points **even for some naive checking** - running over all squares in first matrix, comparing them with all squares in second matrix (you may add some smart breaks here etc.). However, this isn't supposed to give you a full score.

First improvement is to learn how to compare two squares faster than with naive checking. Here **hashing** can help us.

Hashing will allow us to **represent whole square as a single number**, and compare two squares by comparing corresponding numbers.

Type *"Rabin-Karp 2d"* in Google and at the first page you'll have several different guides/implementations for *2d* version of **Rabin-Karp algorithm**. If you are completely new to all these things - start with *1d* version of algorithm (where we are comparing **substrings** instead of **subsquares**). After understanding 1d version - 2d version should be also pretty clear, because it is a simple generalization of 1d.

Now we have O(N^4) checker for particular value of *X*, which is still too slow (but much better than before). How to improve it even more? Instead of comparing our subsquare with every single subsquare from other matrix, we may compare it with all subsquares from that matrix at the same time. We are interested to know if there is a pair of equal subsquares; let's put all subsquares of first matrix **in a set**. Now for every subsquare from second matrix we can check if it has a pair in first matrix, **by simply looking at our set**. It gives us *logN* instead of *N^2* here.

Now we have some sort of proper solution. Time limit isn't very strict; however, if you are actually going to use sets, **it may be too slow**. Instead you may store all values in **sorted vector** and use *std::lower\_bound()* to check that some value is presented there. Also you may get rid of any % operations by doing all calculations modulo 2^64 (yes, there were **no anti-hash tests in data set :)** ) Such solution should get full score without any problems.

**Author's Solution**

#include <bits/stdc++.h>

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace \_\_gnu\_pbds;

using namespace std;

#define Foreach(i, c) for(\_\_typeof((c).begin()) i = (c).begin(); i != (c).end(); ++i)

#define For(i,a,b) for(int (i)=(a);(i) < (b); ++(i))

#define rof(i,a,b) for(int (i)=(a);(i) > (b); --(i))

#define rep(i, c) for(auto &(i) : (c))

#define x first

#define y second

#define pb push\_back

#define PB pop\_back()

#define iOS ios\_base::sync\_with\_stdio(false)

#define sqr(a) (((a) \* (a)))

#define all(a) a.begin() , a.end()

#define error(x) cerr << #x << " = " << (x) <<endl

#define Error(a,b) cerr<<"( "<<#a<<" , "<<#b<<" ) = ( "<<(a)<<" , "<<(b)<<" )\n";

#define errop(a) cerr<<#a<<" = ( "<<((a).x)<<" , "<<((a).y)<<" )\n";

#define coud(a,b) cout<<fixed << setprecision((b)) << (a)

#define L(x) ((x)<<1)

#define R(x) (((x)<<1)+1)

#define double long double

typedef long long ll;

#define int ll

typedef pair<int,int>pii;

typedef vector<int> vi;

typedef complex<double> point;

template <typename T> using os = tree<T, null\_type, less<T>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

template <class T> inline void smax(T &x,T y){ x = max((x), (y));}

template <class T> inline void smin(T &x,T y){ x = min((x), (y));}

const int maxn = 710;

int a[2][maxn][maxn];

int n[2], m[2];

int h[2][maxn][maxn], H[2][maxn][maxn];

int p = 29 \* 91, pw[maxn];

int p2 = 701, pw2[maxn];

inline void init(int x){

For(i,0,n[x]){

int cur = 0;

For(j,0,m[x]){

cur = (cur \* 1LL \* p);

cur = (cur + a[x][i][j]);

h[x][i][j] = cur;

}

}

}

inline int hashrow(int x,int row,int l,int r){

int ans = h[x][row][r];

if(l){

int X = (1LL \* pw[r - l + 1] \* h[x][row][l-1]);

ans = (ans - X);

}

return ans;

}

inline int hashcol(int x,int col,int l,int r){

int ans = H[x][col][r];

if(l){

int X = (1LL \* pw2[r - l + 1] \* H[x][col][l-1]);

ans = (ans - X);

}

return ans;

}

inline bool check(int s){

if(!s) return true;

if(s > min(min(m[0], m[1]), min(n[0], n[1]))) return false;

memset(H, 0, sizeof H);

For(x,0,2)

For(j,0,m[x]){

int l = max(0LL, j - s + 1);

int cur = 0;

For(i, 0, n[x]){

int w = hashrow(x, i, l, j);

cur = (cur \* 1LL \* p2);

cur = (cur + w);

H[x][j][i] = cur;

//if(j)

//Error(cur, w);

}

}

vi mp;

For(x,0,2){

For(j,s-1,m[x])

For(i,s-1,n[x]){

int h = hashcol(x, j, i - s + 1, i);

//error(x);

//Error(i, j);

//error(H[x][j][i]);

//error(h);

if(!x)

mp.pb(h);

else{

int x = lower\_bound(all(mp), h) - mp.begin();

if(x >= 0 && x < (int)mp.size() && mp[x] == h)

return true;

}

}

if(!x){

sort(all(mp));

mp.resize(unique(all(mp)) - mp.begin());

}

}

return false;

}

int ans = 1e9;

main(){

iOS;

pw[0] = pw2[0] = 1;

For(i,1,maxn)

pw[i] = (pw[i-1] \* p), pw2[i] = (pw2[i-1] \* p2);

For(x,0,2){

cin >> n[x] >> m[x];

For(i,0,n[x])

For(j,0,m[x])

{cin >> a[x][i][j]; a[x][i][j] += 16;}

init(x);

}

int l = 0, r = min(min(n[0], n[1]), min(m[0], m[1])) + 1;

while(r - l > 1){

int mid = (l+r)/2;

if(check(mid))

l = mid;

else

r = mid;

}

smin(ans, l);

cout << ans << endl;

}

**Tester's Solution**

#include <bits/stdc++.h>

using namespace std;

long long s1[1<<10][1<<10],s2[1<<10][1<<10],

ar1[1<<10][1<<10],ar2[1<<10][1<<10],pw1[1<<22],pw2[1<<22],n2,m2;

long long l,r;

long long n1,m1;

long long get1(long a,long b,long c)

{

long long res=s1[a+c][b+c]-s1[a][b+c]-s1[a+c][b]+s1[a][b];

res\*=pw1[2000000-a]\*pw2[2000000-b];

return res;

}

long long get2(long a,long b,long c)

{

long long res=s2[a+c][b+c]-s2[a][b+c]-s2[a+c][b]+s2[a][b];

res\*=pw1[2000000-a]\*pw2[2000000-b];

return res;

}

vector<long long> have;

bool solve(int X)

{

have.clear();

for (int i=0;i+X<=n1;i++)

for (int j=0;j+X<=m1;j++)

have.push\_back(get1(i,j,X));

sort(have.begin(),have.end());

for (int i=0;i+X<=n2;i++)

for (int j=0;j+X<=m2;j++)

{

long long res=get2(i,j,X);

int id=lower\_bound(have.begin(),have.end(),res)-have.begin();

if (id!=have.size()&&have[id]==res)

return true;

}

return false;

}

int main(){

ios\_base::sync\_with\_stdio(0);

//cin.tie(0);

pw1[0]=1;

for (int i=1;i<=2000000;i++)

pw1[i]=pw1[i-1]\*173;

pw2[0]=1;

for (int i=1;i<=2000000;i++)

pw2[i]=pw2[i-1]\*137;

cin>>n1>>m1;

for (int i=1;i<=n1;i++)

for (int j=1;j<=m1;j++)

cin>>ar1[i][j];

cin>>n2>>m2;

for (int i=1;i<=n2;i++)

for (int j=1;j<=m2;j++)

cin>>ar2[i][j];

for (int i=1;i<=n1;i++)

for (int j=1;j<=m1;j++)

s1[i][j]=s1[i][j-1]+s1[i-1][j]-s1[i-1][j-1]+ar1[i][j]\*pw1[i]\*pw2[j];

for (int i=1;i<=n2;i++)

for (int j=1;j<=m2;j++)

s2[i][j]=s2[i][j-1]+s2[i-1][j]-s2[i-1][j-1]+ar2[i][j]\*pw1[i]\*pw2[j];

l=0;

r=700;

while (l<r)

{

int mid=l+r+1;

mid/=2;

if (solve(mid))l=mid;

else

r=mid-1;

}

cout<<l<<endl;

return 0;}

# Kth smallest number again

**Dexter** was good in finding the ***K*** th smallest number from a set of numbers. He thought he could solve any problem related to ***K*** th smallest number. His friend **Pipi** challenged him with a problem.  
He gave him various ranges of number, These numbers were arranged in increasing order(only distinct numbers to be taken into account). Now he asked him to find the ***K*** th smallest number in the sequence, again and again.

**Input Format**  
The first line contains ***T***, the number of test cases.  
For each test case, there will be two integers ***N*** and ***Q***.  
Then ***N*** lines follow each line containing two integers ***A*** and ***B*** (denoting the range ***A***-***B***)  
Then ***Q*** lines follow each line containing a non-negative integer ***K*** .

**Output Format**  
For each query output the ***K*** th smallest number.

**Constraints**  
1 <= ***T*** <= 100  
1 <= ***N*** <= 100  
1 <= ***Q*** <= 1000  
-10^18 <= ***A*** <= ***B*** <= 10^18  
***K*** >= 1

**N.B. If Kth smallest number is not present in the series, print -1**

**Sample Input**

1

1 3

1 5

1

3

6

**Sample Output**

1

3

-1

Time Limit: 5

Memory Limit: 256

Source Limit:

**Explanation**

The numbers are "1 2 3 4 5". The 1st smallest number is 1  
The 3rd smallest number is 3 The 6th smallest number is not present. Hence answer is -1

## Solution:

The hurdles one might face in the problem are:  
1. All numbers are given in form of ranges. 2. How to find the kth largest number in the range?

Ranges can be of two types:  
1. Overlapping and non-sorted.  
2. Non-overlapping and sorted.

Considering the simple case first  
**Non-overlapping and sorted**:

Let us consider the case of non-overlapping ranges first:  
Suppose the given ranges are:  
1-4, 8-10, 12-20, 40-50  
Now we have to find **K** th largest number in these ranges:  
Following a pattern. 1. if K>=1 and K<=4 the numbers will lie in the first range i.e. 1-4 .  
2. If K>=5 and K<=7 the numbers will lie in the second range i.e. 8-10.  
3. If K>=8 and K<=16 the numbers will lie in the third range i.e. 12-20.  
4. If K>=17 and K<=27 the numbers will lie in the fourth range i.e. 40-50.

Hence we can think of building a array where:  
arr[i]=Sum of the length of all range till i (where i starts from 1)  
(and arr[0]=0)

arr for this example will be:  
ranges: 1-4, 8-10, 12-20, 40-50  
length: 0 , 4 , 3 , 9 , 11 (Appending a zero length) cumulative: 0 , 4 , 7 , 16 , 27  
There are 27 numbers in total:

Now we have to find that **K** belongs to which interval:  
0-4 or 5-7 or 8-16 or 17-27  
We can easily find this using binary search:  
**Pseudo code:**

binary\_search(arr,value,size,)

{

low=0,high=sz,mid,calc;

while(low<=high)

{

mid=(low+high)/2;

//Value must be just greater than the interval's lower limit and lesser than or equal to interval's higher limit

if(value>arr[mid] && (mid==sz || value<=bin[mid+1]))

{

if(mid==sz)

return -1;

else

{

return mid;

}

}

//If value is greater than the current value then it lies in the part mid+1 to high

else if(value>bin[mid])

low=mid+1;

//If value is lesser than the current value then it lies in the part low to mid-1

else

high=mid-1;

}

return -1;

}

Now we know the range in which the value lies we just have add the lower limit of the range with the difference of value and present cumulative value:

answer=value-arr[mid]+range[mid]

**Overlapping and non-sorted:**  
If the ranges are overlapping and given in non-sorted order. First we need to sort the ranges based on lower limit and if the lower limits of ranges are same sort them according to higher limit.  
Now it becomes an ad-hoc problem to merge this overlapping-intervals such that we get a non-overlapping sorted interval.  
One can get the idea by following this example:  
Consider the ranges 2-4, 1-8, 1-6, 9-10  
Sorting this ranges: 1-6, 1-8, 2-4, 9-10

The steps we need to follow to do the same are:  
1. First push the range 1-6 in the new interval array:  
2. Check whether the new range lower\_limit is less than upper limit of the latest range in new interval array.  
If it's in the interval then update the upper value of the interval to max(higher\_limit of new array,higher\_limit of old array).  
If not then insert this interval in the array.

Implementing this approach in this interval:  
1. first push **1-6** in the new array. 2. **1-8** lower limit is in the range **1-6**. Hence update the maximum value of new array(i.e. 6) to **max(6,8)** i.e. 8. The range in new array is changed to **1-8**.  
3. **2-4** lower limit is in the range **1-8**. Hence update the maximum value of new array(i.e. 8) to **max(8,4)** i.e. 8. The range in new array remains same **1-8**.  
4. **9-10** lower limit is not in range **1-8**. Hence this range needs to be inserted in the array. The range is new array are: **1-8, 9-10**.

**Author's Solution**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Akash Agrawall

IIIT HYDERABAD

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<cstdio>

#include<iostream>

#include<cstdlib>

#include<cmath>

#include<cstring>

#include<climits>

#include<algorithm>

#include<vector>

#include<stdio.h>

#include<math.h>

using namespace std;

#define FOR(i,a,b) for(i= a ; i < b ; ++i)

#define rep(i,n) FOR(i,0,n)

#define pb push\_back

#define sz(x) int(x.size())

#define mp make\_pair

#define si(n) scanf("%d",&n)

#define pi(n) printf("%d ",n)

#define pin(n) printf("%d\n",n)

#define pln(n) printf("%lld\n",n)

#define pl(n) printf("%lld ",n)

#define sl(n) scanf("%lld",&n)

#define scan(v,n) vector<int> v;rep(i,n){ int j;si(j);v.pb(j);}

#define mod (int)(1e9 + 7)

#define ll long long int

#define F first

#define S second

ll modpow(ll a,ll n,ll temp){ll res=1,y=a;while(n>0){if(n&1)res=(res\*y)%temp;y=(y\*y)%temp;n/=2;}return res%temp;}

vector<pair<ll,ll> > arr,brr;

vector<ll> bin;

void modify()

{

ll i,sz,a,b,tp=0;

brr.pb(arr[0]);

sz=arr.size();

FOR(i,1,sz)

{

a=arr[i].F;

b=arr[i].S;

if(a>brr[tp].S)

{

tp++;

brr.pb(mp(a,b));

}

else

{

brr[tp].S=max(brr[tp].S,b);

}

}

}

ll binary(ll sz, ll val)

{

ll low=0,high=sz,mid,calc;

while(low<=high)

{

mid=(low+high)/2;

if(bin[mid]<val && (mid==sz || bin[mid+1]>=val))

{

if(mid==sz)

return -1;

else

{

calc=val-bin[mid];

return brr[mid].F+calc-1;

}

}

else if(val>bin[mid])

low=mid+1;

else

high=mid-1;

}

return -1;

}

int main()

{

ll t,n,q,a,b,sz,i,calc,k;

sl(t);

while(t--)

{

arr.clear();

brr.clear();

bin.clear();

sl(n);

sl(q);

rep(i,n)

{

sl(a);

sl(b);

arr.pb(mp(a,b));

}

sort(arr.begin(),arr.end());

modify();

sz=brr.size();

bin.pb(0);

rep(i,sz)

{

calc=brr[i].S-brr[i].F+1+bin[i];

bin.pb(calc);

}

rep(i,q)

{

sl(k);

calc=binary(sz,k);

pln(calc);

}

}

return 0;

}

# Find pivot element in a sorted and rotated array

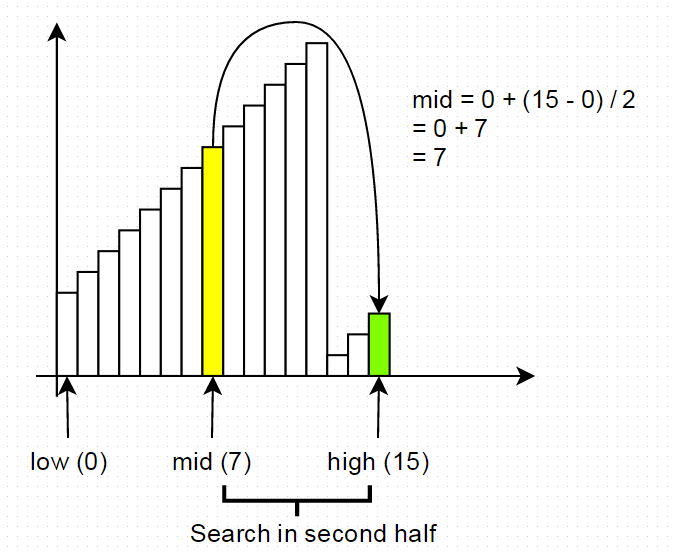
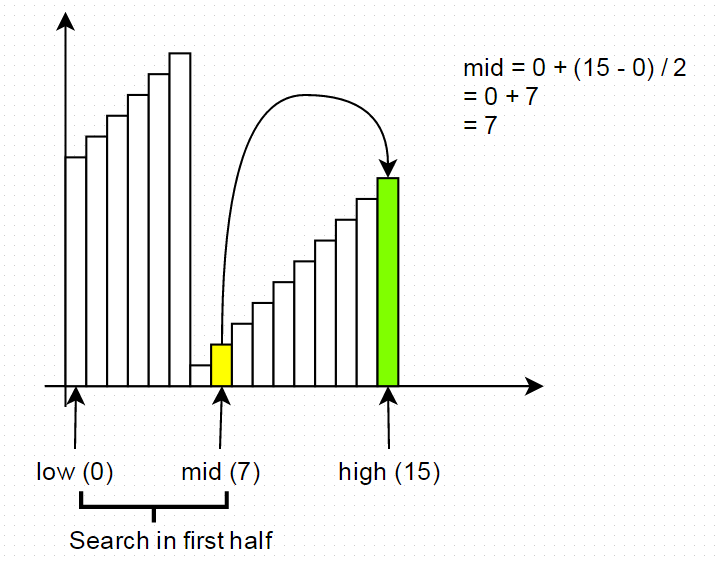
**Problem Statement** – Suppose we have a sorted array, and now we rotate it N times, find the pivot element. The pivot element would be the largest element. Also, can you calculate N?

**Clues** –

* Solution should be O(log N) in time and O(1) in space.
* Can you think of  a binary search based solution where you keep comparing the middle element with the last element?

## Solution:

 The pivot element is basically, the largest element in an array. For a sorted and rotated array, it might be somewhere in between. We can solve this in O(log N) time, through a divide-and-conquer approach, which is similar to peak finding algorithm. We will have the lower limit (low) and the upper limit (high) from which we will calculate the ‘mid’. Now, we must address a few cases –

* **arr[mid] > arr[high]** – If the mid element is greater than the last element, then the pivot should be in the second half of the array. Why is this so? This is actually because it is a sorted and rotated array. You may not understand this at first, but think of the values of the elements in the array as a histogram. Try to understand the picture below, I’m sure you can figure out why this happens –[](http://theoryofprogramming.azurewebsites.net/wp-content/uploads/2015/09/pivot-element-case-1.png)
* Pivot Element – Case 1
* **arr[mid] < arr[high]** – If the mid element is less than the last element, then the pivot should be in the first half of the range.[](http://theoryofprogramming.azurewebsites.net/wp-content/uploads/2015/09/pivot-element-case-2.png)
* Pivot Element – Case 2
* **If mid element is a peak** – If the mid element satisfies the property of the peak element (not lesser than its neighbors), then, it is the pivot.

If we can put these conditions, and handle the trivial cases, we can compute the pivot element. The number of times the array is rotated would be (IndexOfPivotReturned + 1).

int i=0, j=nums.size()-1, n=nums.size();

while(i<=j){

int m = (i+j)/2;

if(nums[m]<nums[0])

j = m-1;

else if(nums[m]>=nums[0])

i = m+1;

}

cout<<”pivot element is ”<<nums[j]<<endl;

# K-th Element of Two Sorted Arrays

Given two sorted arrays **arr1** and **arr2** of size **N** and **M** respectively and an element **K**. The task is to find the element that would be at the k’th position of the final sorted array.

**Example 1:**

**Input:**

arr1[] = {2, 3, 6, 7, 9}

arr2[] = {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.

**Example 2:**

**Input:**

arr1[] = {100, 112, 256, 349, 770}

arr2[] = {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.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **kthElement()** which takes the arrays **arr1[]**, **arr2[]**, its size **N**and **M**respectively and an integer **K**as inputs and returns the element at the Kth position.

**Expected Time Complexity:** O(Log(N) + Log(M))  
**Expected Auxiliary Space:** O(Log (N))

**Constraints:**  
1 <= N, M <= 106  
1 <= arr1i, arr2i < INT\_MAX  
1 <= K <= N+M

## Solution:

**Basic Approach**   
Since we are given two sorted arrays, we can use the merging technique to get the final merged array. From this, we simply go to the k’th index.

// 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];

}

// 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(n)   
**Auxiliary Space :** O(m + n)

**Space Optimized Version of above approach:** We can avoid the use of extra array.

// C++ program to find kth element

// from two sorted arrays

#include <bits/stdc++.h>

using namespace std;

int find(int A[], int B[], int m,

int n, int k\_req)

{

int k = 0, i = 0, j = 0;

// Keep taking smaller of the current

// elements of two sorted arrays and

// keep incrementing k

while(i < m && j < n)

{

if(A[i] < B[j])

{

k++;

if(k == k\_req)

return A[i];

i++;

}

else

{

k++;

if(k == k\_req)

return B[j];

j++;

}

}

// If array B[] is completely traversed

while(i < m)

{

k++;

if(k == k\_req)

return A[i];

i++;

}

// If array A[] is completely traversed

while(j < n)

{

k++;

if(k == k\_req)

return B[j];

j++;

}

}

// Driver Code

int main()

{

int A[5] = { 2, 3, 6, 7, 9 };

int B[4] = { 1, 4, 8, 10 };

int k = 5;

cout << find(A, B, 5, 4, k);

return 0;

}

**Output**

6

**Time Complexity:** O(k)   
**Auxiliary Space:** O(1)

1. **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.
2. 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. Set the last element of arr2 to be arr2[mid2].
3. In this way, define a new subproblem with half the size of one of the arrays.

// 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)

Space Complexity: O(logn) used for stack in calling the function.

**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. The below implementation displays this.

Explanation:

Instead of comparing the middle element of the arrays,

we compare the k / 2nd 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.

// 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);

// Check if arr1 is empty returning

// k-th element of arr2

if (m == 0)

return arr2[k - 1];

// Check 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;

}

**Output**

6

**Time Complexity:**O(log k)

**Another Approach: (Using Min Heap)**

1. Push the elements of both arrays to a priority queue (min-heap).
2. Pop-out k-1 elements from the front.
3. Element at the front of the priority queue is the required answer.

Below is the implementation of the above approach:

// C++ Program to find kth

// element from two sorted arrays

#include <bits/stdc++.h>

using namespace std;

// Function to find K-th min

int kth(int\* a, int\* b, int n, int m, int k)

{

// Declaring a min heap

priority\_queue<int, vector<int>,

greater<int> > pq;

// Pushing elements for

// array a to min-heap

for (int i = 0; i < n; i++) {

pq.push(a[i]);

}

// Pushing elements for

// array b to min-heap

for (int i = 0; i < m; i++) {

pq.push(b[i]);

}

// Poping-out K-1 elements

while (k-- > 1) {

pq.pop();

}

return pq.top();

}

//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(NlogN)

**Space Complexity:** O(m+n)

**Another Approach : (Using Upper Bound STL)**

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, The 1st element of this array is 1. The thing to notice here is upper\_bound(6) gives 5, upper\_bound(4) gives 4 that is number of element equal to or less than the number we are giving as input to upper\_bound().*

*Here is another example*

*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.*

**Observation required :**

The simplest method to solve this question is using upper\_bound to check what is the position of a element in the sorted array. The upper\_bound function return the pointer to element which is greater than the element we searched.

So to find the kth element we need to just find the element whose upper\_bound() is 4. So again now we now what upper\_bound() gives us we need 1 last observation to solve this question. If we have been given 2 arrays, We just need to the sum of upper\_bound for the 2 arrays

*Input : Array 1 – 2 3 6 7 9*

*Array 2 – 1 4 8 10*

*k = 5*

*Value of upper\_bound for value(6) in array1 is 3 and for array 2 is 2. This give us a total of 5. which is the answer.*

**Algorithm :**

* We take a mid between [L,R] using the formula mid = (L+R)/2.
* Check if the middle can be the kth element using upper\_bound() function
* Find the sum of upper\_bound() for both the arrays and if the sum is >= K, It’s a possible value of kth element.
* If sum is >= K then we assign R = mid – 1.
* else if sum <k then the current mid is too small and we assign L = mid+1.
* Repeat from top
* Return the smallest value found.

**Here is the implementation for the optimized method :**

// C++ program to find the kth element

#include <bits/stdc++.h>

using namespace std;

long long int maxN

= 1e10; // the maximum value in the array possible.

long long int kthElement(int arr1[], int arr2[], int n,

int m, int k)

{

long long int left = 1,

right

= maxN; // The range of where ans can lie.

long long int ans = 1e15; // We have to find min of all

// the ans so take .

// using binary search to check all possible values of

// kth element

while (left <= right) {

long long int mid = (left + right) / 2;

long long int up\_cnt

= upper\_bound(arr1, arr1 + n, mid) - arr1;

up\_cnt += upper\_bound(arr2, arr2 + m, mid) - arr2;

if (up\_cnt >= k) {

ans = min(ans,

mid); // find the min of all answers.

right

= mid - 1; // Try to find a smaller answer.

}

else

left = mid + 1; // Current mid is too small so

// shift right.

}

return ans;

}

// Driver code

int main()

{

// Example 1

int n = 5, m = 7, k = 7;

int arr1[n] = { 100, 112, 256, 349, 770 };

int arr2[m] = { 72, 86, 113, 119, 265, 445, 892 };

cout << kthElement(arr1, arr2, n, m, k) << endl;

return 0;

}

**Output**

256

Time Complexity : O( Log( maxN ).log( N+M ) )  
Auxiliary Space : O( 1 )

**Another approach (Most efficient)**

Use the modification of binary search in the similar way as we did in case of finding Median of two sorted arrays of different size. (Question 36 of Arrays).

Implementation:

int kthElement(int arr1[], int arr2[], int n, int m, int k)

{

if(n>m)

return kthElement(arr2, arr1, m, n, k);

int low = max(0,k-m), high = min(k,n);

while(low<=high){

int cut1 = low + (high-low)/2;

int cut2 = k - cut1;

int l1 = cut1==0 ? INT\_MIN : arr1[cut1-1];

int l2 = cut2==0 ? INT\_MIN : arr2[cut2-1];

int r1 = cut1==n ? INT\_MAX : arr1[cut1];

int r2 = cut2==m ? INT\_MAX : arr2[cut2];

if(l1<=r2 && l2<=r1)

return max(l1,l2);

else if(l1>r2)

high = cut1-1;

else

low = cut1+1;

}

}

**Time Complexity :** O(log(min(m,n)))

**Space Complexity :** O(1)

# Aggressive cows

Farmer John has built a new long barn, with N (2 <= N <= 100,000) stalls. The stalls are located along a straight line at positions x1,...,xN (0 <= xi <= 1,000,000,000).  
  
His C (2 <= C <= N) cows don't like this barn layout and become aggressive towards each other once put into a stall. To prevent the cows from hurting each other, FJ wants to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

Input

*t* – the number of test cases, then *t* test cases follows.  
\* Line 1: Two space-separated integers: N and C  
\* Lines 2..N+1: Line i+1 contains an integer stall location, xi

Output

For each test case output one integer: the largest minimum distance.

Example

**Input:**

1

5 3

1

2

8

4

9

**Output:**

3

**Output details:**

FJ can put his 3 cows in the stalls at positions 1, 4 and 8,  
resulting in a minimum distance of 3.

## Solution:

Use binary search on the range of minimum distance possible between cows.

#include <bits/stdc++.h>

using namespace std;

int main()

{

int t;

cin>>t;

while(t--){

int stall,cow;

cin>>stall>>cow;

vector<int> stalls(stall);

for(int i=0;i<stall;i++)

cin>>stalls[i];

if(cow>stall)

cout<<"0"<<endl;

else{

sort(stalls.begin(),stalls.end());

int low=1, high=stalls[stall-1]-stalls[0], ans=0;

while(low<=high){

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

int tempc=cow-1, temps=1, prevs=0;

while(tempc>0 && temps<stall){

if(stalls[temps]-stalls[prevs]>=mid){

prevs = temps;

tempc--;

}

temps++;

}

if(tempc==0){

ans = mid;

low = mid+1;

}

else

high = mid-1;

}

cout<<ans<<endl;

}

}

return 0;

}

**Time Complexity:** O(log(range of stall position))

**Space Complexity:** O(1)

# Book Allocation Problem

You are given **N** number of books. Every ith book has**Ai**number of pages.   
You have to allocate contagious books to **M** number of students. There can be many ways or permutations to do so. In each permutation, one of the **M** students will be allocated the maximum number of pages. Out of all these permutations, the task is to find that particular permutation in which the maximum number of pages allocated to a student is minimum of those in all the other permutations and print this minimum value.

Each book will be allocated to exactly one student. Each student has to be allocated at least one book.

**Note:**Return **-1** if a valid assignment is not possible, and **allotment should be in contiguous order (see the explanation for better understanding).**

**Example 1:**

**Input:**

N = 4

A[] = {12,34,67,90}

M = 2

**Output:**

113

**Explanation:**

Allocation can be done in following ways:

{12} and {34, 67, 90} Maximum Pages = 191

{12, 34} and {67, 90} Maximum Pages = 157

{12, 34, 67} and {90}  Maximum Pages =113

Therefore, the minimum of these cases is

113, which is selected as the output.

**Example 2:**

**Input:**

N = 3

A[] = {15,17,20}

M = 2

**Output:**

32

**Explanation:**

Allocation is done as

{15,17} and {20}

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findPages()** which takes 2 Integers N, and m and an array A[] of length N as input and returns the expected answer.

**Expected Time Complexity**: O(NlogN)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 105  
1 <= A [ i ] <= 106  
1 <= M <= 105

## Solution:

The idea is to use [Binary Search](https://www.geeksforgeeks.org/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++ 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))

{

// update result to current distribution

// as it's the best we have found till now.

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;

}

**Time Complexity:** O(n\*log(total pages))

**Space Complexity:** O(1)

# EKOSPOJ:

Lumberjack Mirko needs to chop down **M** metres of wood. It is an easy job for him since he has a nifty new woodcutting machine that can take down forests like wildfire. However, Mirko is only allowed to cut a single row of trees.

Mirko‟s machine works as follows: Mirko sets a height parameter **H** (in metres), and the machine raises a giant sawblade to that height and cuts off all tree parts higher than **H** (of course, trees not higher than **H** meters remain intact). Mirko then takes the parts that were cut off. For example, if the tree row contains trees with heights of 20, 15, 10, and 17 metres, and Mirko raises his sawblade to 15 metres, the remaining tree heights after cutting will be 15, 15, 10, and 15 metres, respectively, while Mirko will take 5 metres off the first tree and 2 metres off the fourth tree (7 metres of wood in total).

Mirko is **ecologically** minded, so he doesn‟t want to cut off more wood than necessary. That‟s why he wants to set his sawblade as high as possible. Help Mirko find the **maximum integer height** of the sawblade that still allows him to cut off **at least M** metres of wood.

Input

The first line of input contains two space-separated positive integers, **N** (the number of trees, 1 ≤ **N** ≤ 1 000 000) and **M** (Mirko‟s required wood amount, 1 ≤ **M** ≤ 2 000 000 000).

The second line of input contains **N** space-separated positive integers less than 1 000 000 000, the heights of each tree (in metres). The sum of all heights will exceed **M**, thus Mirko will always be able to obtain the required amount of wood.

Output

The first and only line of output must contain the required height setting.

Example

**Input:**

4 7

20 15 10 17

**Output:**

15

**Input:**

5 20

4 42 40 26 46

**Output:**

36

## Solution:

Apply modification of Binary Search.

Implementation:

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n,req;

cin>>n>>req;

int tree[n];

for(int i=0;i<n;i++)

cin>>tree[i];

int max\_height = INT\_MIN;

for(int i=0;i<n;i++)

max\_height = max(max\_height,tree[i]);

int low = max\_height-req, high = max\_height-1, ans=1;

while(low<=high){

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

long long wood=0;

for(int i=0;i<n;i++){

if(tree[i]>mid)

wood += tree[i]-mid;

}

if(wood>=req){

ans = max(ans,mid);

low = mid+1;

}

else

high = mid-1;

}

cout<<ans<<endl;

return 0;

}

**Time Complexity:** O(n\*log(req wood))

**Space Complexity:** O(1)

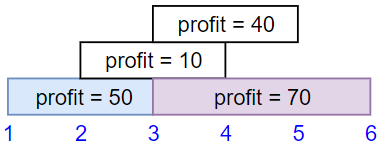
# Weighted Job Scheduling Algo

We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i].

You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.

If you choose a job that ends at time X you will be able to start another job that starts at time X.

**Example 1:**

****

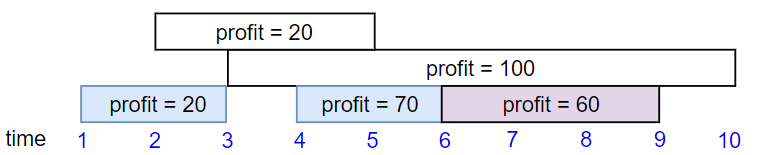
**Input:** startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

**Output:** 120

**Explanation:** The subset chosen is the first and fourth job.

Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.

**Example 2:**

****

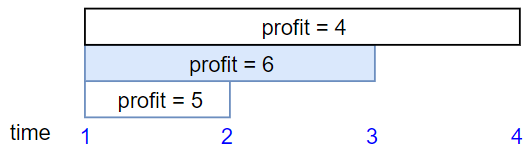
**Input:** startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]

**Output:** 150

**Explanation:** The subset chosen is the first, fourth and fifth job.

Profit obtained 150 = 20 + 70 + 60.

**Example 3:**

****

**Input:** startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]

**Output:** 6

**Constraints:**

* 1 <= startTime.length == endTime.length == profit.length <= 5 \* 104
* 1 <= startTime[i] < endTime[i] <= 109
* 1 <= profit[i] <= 104

## Solution:

The above problem can be solved using following recursive solution.

1) First sort jobs according to finish time.

2) Now apply following recursive process.

// Here arr[] is array of n jobs

findMaximumProfit(arr[], n)

{

a) if (n == 1) return arr[0];

b) Return the maximum of following two profits.

(i) Maximum profit by excluding current job, i.e.,

findMaximumProfit(arr, n-1)

(ii) Maximum profit by including the current job

}

**How to find the profit including current job?**

The idea is to find the latest job before the current job (in

sorted array) that doesn't conflict with current job 'arr[n-1]'.

Once we find such a job, we recur for all jobs till that job and

add profit of current job to result.

In the above example, "job 1" is the latest non-conflicting

for "job 4" and "job 2" is the latest non-conflicting for "job 3".

We have discussed recursive and Dynamic Programming based approaches in the [previous article](https://www.geeksforgeeks.org/weighted-job-scheduling/). The implementations discussed in above post uses linear search to find the previous non-conflicting job. In this post, Binary Search based solution is discussed. The time complexity of Binary Search based solution is O(n Log n).

The algorithm is:

1. Sort the jobs by non-decreasing finish times.
2. For each i from 1 to n, determine the maximum value of the schedule from the subsequence of jobs[0..i]. Do this by comparing the inclusion of job[i] to the schedule to the exclusion of job[i] to the schedule, and then taking the max.

To find the profit with inclusion of job[i]. we need to find the latest job that doesn’t conflict with job[i]. The idea is to use Binary Search to find the latest non-conflicting job.

// C++ program for weighted job scheduling using Dynamic

// Programming and Binary Search

#include <iostream>

#include <algorithm>

using namespace std;

// A job has start time, finish time and profit.

struct Job

{

int start, finish, profit;

};

// A utility function that is used for sorting events

// according to finish time

bool myfunction(Job s1, Job s2)

{

return (s1.finish < s2.finish);

}

// A Binary Search based function to find the latest job

// (before current job) that doesn't conflict with current

// job. "index" is index of the current job. This function

// returns -1 if all jobs before index conflict with it.

// The array jobs[] is sorted in increasing order of finish

// time.

int binarySearch(Job jobs[], int index)

{

// Initialize 'lo' and 'hi' for Binary Search

int lo = 0, hi = index - 1;

// Perform binary Search iteratively

while (lo <= hi)

{

int mid = (lo + hi) / 2;

if (jobs[mid].finish <= jobs[index].start)

{

if (jobs[mid + 1].finish <= jobs[index].start)

lo = mid + 1;

else

return mid;

}

else

hi = mid - 1;

}

return -1;

}

// The main function that returns the maximum possible

// profit from given array of jobs

int findMaxProfit(Job arr[], int n)

{

// Sort jobs according to finish time

sort(arr, arr+n, myfunction);

// Create an array to store solutions of subproblems. table[i]

// stores the profit for jobs till arr[i] (including arr[i])

int \*table = new int[n];

table[0] = arr[0].profit;

// Fill entries in table[] using recursive property

for (int i=1; i<n; i++)

{

// Find profit including the current job

int inclProf = arr[i].profit;

int l = binarySearch(arr, i);

if (l != -1)

inclProf += table[l];

// Store maximum of including and excluding

table[i] = max(inclProf, table[i-1]);

}

// Store result and free dynamic memory allocated for table[]

int result = table[n-1];

delete[] table;

return result;

}

// Driver program

int main()

{

Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Optimal profit is " << findMaxProfit(arr, n);

return 0;

}

Output:

Optimal profit is 250

**My code:**

struct Job{

int start;

int end;

int pro;

};

bool compareEnd(Job a, Job b){

if(a.end==b.end)

return a.start<b.start;

else

return a.end<b.end;

}

class Solution {

public:

int jobScheduling(vector<int>& startTime, vector<int>& endTime, vector<int>& profit) {

int n = startTime.size();

if(n==0)

return 0;

Job jobs[n];

for(int i=0;i<n;i++){

jobs[i].start = startTime[i];

jobs[i].end = endTime[i];

jobs[i].pro = profit[i];

}

sort(jobs,jobs+n,compareEnd);

vector<int> dp(n,0);

dp[0] = jobs[0].pro;

for(int i=1;i<n;i++){

int low = 0, high = i-1, ans = -1;

while(low<=high){

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

if(jobs[mid].end<=jobs[i].start){

ans = max(ans,mid);

low = mid+1;

}

else

high = mid-1;

}

if(ans!=-1)

dp[i] = max(dp[i-1], dp[ans]+jobs[i].pro);

else

dp[i] = max(dp[i-1], jobs[i].pro);

}

return dp[n-1];

}

};

# Missing Number in AP

Given an array that represents elements of arithmetic progression in order. One element is missing in the progression, find the missing number.

Input: arr[] = {2, 4, 8, 10, 12, 14}

Output: 6

Input: arr[] = {1, 6, 11, 16, 21, 31};

Output: 26

## Solution:

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/2th 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++ program to find the missing number

// in a given arithmetic progression

#include<iostream>

using namespace std;

#define INT\_MAX 2147483647;

class GFG

{

// A binary search based recursive function that returns

// the missing element in arithmetic progression

public: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 Code

int main()

{

GFG g;

int arr[] = {2, 4, 8, 10, 12, 14};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "The missing element is "

<< g.findMissing(arr, n);

return 0;

}

**Output**

The missing element is 6

**Iterative:**

The idea is to go to the middle element. Check if the index of middle element is equal to (nth position of middle element in AP) – 1 then the missing element lies at right half if not then the missing element lies at left half (this idea is similar to [Find the only repeating element in a sorted array of size n](https://www.geeksforgeeks.org/find-repeating-element-sorted-array-size-n/) ).  After breaking out of binary search loop the missing element will lie between high and low. We can find the missing element by adding a common difference with element at index high or by subtracting a common difference with element at index low.

Following is implementation of above idea.

// C++ program to find the missing number

// in a given arithmetic progression

#include<iostream>

using namespace std;

#define INT\_MAX 2147483647;

class GFG

{

// A binary search based function that returns

// the missing element in arithmetic progression

public:int findMissingUtil(int arr[], int low,

int high, int diff)

{

// Find index of middle element

int mid;

while (low <= high)

{

// find index of middle element

mid = (low + high) / 2;

// if mid == (nth position of element in AP)-1

// the missing element will exist in right half

if ((arr[mid] - arr[0]) / diff == mid)

low = mid + 1;

else

// the missing element will exist in left half

high = mid - 1;

}

// after breaking out of binary search loop

// our missing element will exist between high and low

// our missing element will be a[high] + commom difference

// or a[low] - commom difference

return arr[high] + 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 Code

int main()

{

GFG g;

int arr[] = {2, 4, 8, 10, 12, 14};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "The missing element is "

<< g.findMissing(arr, n);

return 0;

}

**Output**

The missing element is 6

**Time Complexity:** O(logn)

**Space Complexity:** O(1)

# Smallest number with atleastn trailing zeroes infactorial

Given a number **n**. The task is to find the smallest number whose factorial contains at least n trailing zeroes.

**Example 1:**

**Input:**

n = 1

**Output:** 5

**Explanation :** 5! = 120 which has at

least 1 trailing 0.

**Example 2:**

**Input:**

n = 6

**Output:** 25

**Explanation :** 25! has at least

6 trailing 0.

**User Task:**  
Complete the function **findNum()** which takes an integer N as input parameters, and returns the answer.

**Expected Time Complexity:**O(log2 N \* log5 N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= n <= 104

## Solution:

In the article for [Count trailing zeroes in factorial of a number](https://www.geeksforgeeks.org/count-trailing-zeroes-factorial-number/), we have discussed number of zeroes is equal to number of 5’s in prime factors of x!. We have discussed below formula to count number of 5’s.

Trailing 0s in x! = Count of 5s in prime factors of x!

= floor(x/5) + floor(x/25) + floor(x/125) + ....

Let us take few examples to observe pattern

5! has 1 trailing zeroes

[All numbers from 6 to 9

have 1 trailing zero]

10! has 2 trailing zeroes

[All numbers from 11 to 14

have 2 trailing zeroes]

15! to 19! have 3 trailing zeroes

20! to 24! have 4 trailing zeroes

25! to 29! have 6 trailing zeroes

We can notice that, the maximum value whose factorial contain n trailing zeroes is 5\*n.  
So, to find minimum value whose factorial contains n trailing zeroes, use binary search on range from 0 to 5\*n. And, find the smallest number whose factorial contains n trailing zeroes.

// C++ program tofind smallest number whose

// factorial contains at least n trailing

// zeroes.

#include<bits/stdc++.h>

using namespace std;

// Return true if number's factorial contains

// at least n trailing zero else false.

bool check(int p, int n)

{

int temp = p, count = 0, f = 5;

while (f <= temp)

{

count += temp/f;

f = f\*5;

}

return (count >= n);

}

// Return smallest number whose factorial

// contains at least n trailing zeroes

int findNum(int n)

{

// If n equal to 1, return 5.

// since 5! = 120.

if (n==1)

return 5;

// Initialising low and high for binary

// search.

int low = 0;

int high = 5\*n;

// Binary Search.

while (low <high)

{

int mid = (low + high) >> 1;

// Checking if mid's factorial contains

// n trailing zeroes.

if (check(mid, n))

high = mid;

else

low = mid+1;

}

return low;

}

// driver code

int main()

{

int n = 6;

cout << findNum(n) << endl;

return 0;

}

**Output :** 

25

# Painters Partition Problem

Dilpreet wants to paint his dog's home that has **n** boards with different lengths. The length of ithboard is given by **arr[i]** where **arr[]** is an array of **n** integers. He hired **k** painters for this work and each painter takes **1 unit time to paint 1 unit of the board.**

The problem is to find the minimum time to get this job done if all painters start together with the constraint that any painter will only paint continuous boards, say boards numbered **{2,3,4}**or only board **{1}** or nothing but not boards **{2,4,5}**.

**Example 1:**

**Input:**

n = 5

k = 3

arr[] = {5,10,30,20,15}

**Output:** 35

**Explanation:** The most optimal way will be:

Painter 1 allocation : {5,10}

Painter 2 allocation : {30}

Painter 3 allocation : {20,15}

Job will be done when all painters finish

i.e. at time = max(5+10, 30, 20+15) = 35

**Example 2:**

**Input:**

n = 4

k = 2

arr[] = {10,20,30,40}

**Output:** 60

**Explanation:** The most optimal way to paint:

Painter 1 allocation : {10,20,30}

Painter 2 allocation : {40}

Job will be complete at time = 60

**Your task:**  
Your task is to complete the function **minTime()**which takes the integers **n**and **k** and the array **arr[]** as input and returns the minimum time required to paint all partitions.

**Expected Time Complexity:** O(n log m) , m = sum of all boards' length  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ n ≤ 105  
1 ≤ k ≤ 105  
1 ≤ arr[i] ≤ 105

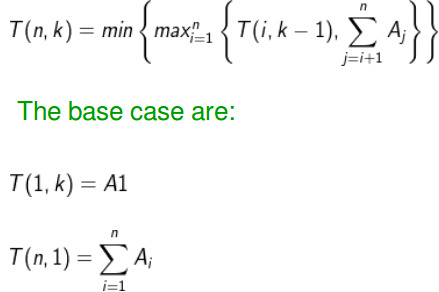
## Solution:

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:



Following is the implementation of the above recursive equation:

// 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;

}

**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:

// 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;

}

**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.

**Using Binary Search:**

In the [previous post](https://www.geeksforgeeks.org/painters-partition-problem/) 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 elements in 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.

// 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;

}

**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[]))).

# ROTI-Prata SPOJ

IEEE is having its AGM next week and the president wants to serve cheese prata after the meeting. The subcommittee members are asked to go to food connection and get P(P<=1000) pratas packed for the function. The stall has L cooks(L<=50) and each cook has a rank R(1<=R<=8). A cook with a rank R can cook 1 prata in the first R minutes 1 more prata in the next 2R minutes, 1 more prata in 3R minutes and so on(he can only cook a complete prata) ( For example if a cook is ranked 2.. he will cook one prata in 2 minutes one more prata in the next 4 mins an one more in the next 6 minutes hence in total 12 minutes he cooks 3 pratas in 13 minutes also he can cook only 3 pratas as he does not have enough time for the 4th prata). The webmaster wants to know the minimum time to get the order done. Please write a program to help him out.

Input

The first line tells the number of test cases. Each test case consist of 2 lines. In the first line of the test case we have P the number of prata ordered. In the next line the first integer denotes the number of cooks L and L integers follow in the same line each denoting the rank of a cook.

Output

Print an integer which tells the number of minutes needed to get the order done.

Example

**Input:**  
  
3  
10  
4 1 2 3 4  
8  
1 1  
8  
8 1 1 1 1 1 1 1 1  
**Output:**  
12  
36  
1

## Solution:

Apply modification of binary search.

#include <bits/stdc++.h>

using namespace std;

int main()

{

int t;

cin>>t;

while(t--){

int order,cooks;

cin>>order>>cooks;

vector<int> ranks(cooks);

for(int i=0;i<cooks;i++)

cin>>ranks[i];

vector<int> countPerRank(9,0);

for(int i=0;i<cooks;i++)

countPerRank[ranks[i]]++;

int low=0, high=1e8,ans=1e8;

while(low<=high){

int mid = low + (high-low)/2, prata=0;

for(int i=1;i<=8;i++){

int temp=i,j=1;

while(temp<=mid){

prata += countPerRank[i];

j++;

temp += j\*i;

}

}

if(prata>=order){

ans = min(ans,mid);

high = mid-1;

}

else

low = mid+1;

}

cout<<ans<<endl;

}

return 0;

}

**Time Complexity:** O(log(1e8)\*8\*((1e8)/2)), Bcz outer while loop will execute log(1e8) times while inner while loop will execute maximum (1e8)/2 time for each iteration in outer loop.

**Space Complexity:** O(1)

# DoubleHelix SPOJ

Two ﬁnite, strictly increasing, integer sequences are given. Any common integer between the two sequences constitute an intersection point. Take for example the following two sequences where intersection points are  
printed in bold:

* First= 3 5 **7** 9 20 **25** 30 40 **55** 56 **57** 60 62
* Second= 1 4 **7** 11 14 **25** 44 47 **55** **57** 100

You can ‘walk” over these two sequences in the following way:

1. You may start at the beginning of any of the two sequences. Now start moving forward.
2. At each intersection point, you have the choice of either continuing with the same sequence you’re currently on, or switching to the other sequence.

The objective is ﬁnding a path that produces the maximum sum of data you walked over. In the above example, the largest possible sum is 450, which is the result of adding 3, 5, 7, 9, 20, 25, 44, 47, 55, 56, 57, 60, and 62

Input

Your program will be tested on a number of test cases. Each test case will be speciﬁed on two separate lines. Each line denotes a sequence and is speciﬁed using the following format:

n v1 v2 ... vn

Where n is the length of the sequence and vi is the ith element in that sequence. Each sequence will have at least one element but no more than 10,000. All elements are between -10,000 and 10,000 (inclusive).  
The last line of the input includes a single zero, which is not part of the test cases.

Output

For each test case, write on a separate line, the largest possible sum that can be produced.

Sample

**Input:**

13 3 5 7 9 20 25 30 40 55 56 57 60 62

11 1 4 7 11 14 25 44 47 55 57 100

4 -5 100 1000 1005

3 -12 1000 1001

0

**Output:**

450

2100

## Solution:

**Two pointer approach:**

DoubleHelix(A,B,m,n)

sum1 = 0

sum2 = 0

i = 0

j = 0

while i < m and j < n

if A[i] < B[j]

sum1 += A[i++]

else if A[i] > B[j]

sum2 += B[j++]

else

sum1 += A[i]

sum2 += B[j]

sum1 = sum2 = max(sum1, sum2)

i++, j++

while i < m

sum1 += A[i++]

while j < n

sum2 += B[j++]

return max(sum1, sum2)

* **Time Complexity**: **O(m+n)** where m is the length of first sequence and n is the length of second sequence. This is because the worst case for this algorithm occurs when there are no intersection points and thus each element in both lists will be added to the sum variables once leading to the above time complexity.
* **Space Complexity**: **O(1)**

**My Implementation:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int m;

cin>>m;

while(m>0){

//cout<<m<<endl;

vector<int> cummSumA(m,0);

unordered\_map<int,int> mpA;

int t;

for(int i=0;i<m;i++){

cin>>t;

mpA[t] = i;

if(i>0)

cummSumA[i] = t + cummSumA[i-1];

else

cummSumA[i] = t;

}

int n, prev =-1;

long long res=0, temp\_Sum=0;

cin>>n;

for(int i=0;i<n;i++){

cin>>t;

temp\_Sum += t;

if(mpA.find(t)!=mpA.end()){

if(prev!=-1){

if(temp\_Sum>cummSumA[mpA[t]]-cummSumA[prev])

res += temp\_Sum;

else

res += cummSumA[mpA[t]]-cummSumA[prev];

}

else{

if(temp\_Sum>cummSumA[mpA[t]])

res += temp\_Sum;

else

res += cummSumA[mpA[t]];

}

temp\_Sum = 0;

prev = mpA[t];

}

}

if(temp\_Sum>cummSumA[m-1]-cummSumA[prev])

res += temp\_Sum;

else

res += cummSumA[m-1]-cummSumA[prev];

cout<<res<<endl;

cin>>m;

}

return 0;

}

**Time Complexity:** O(m+n)

**Space Complexity:** O(m)

# Subset Sums

Given a sequence of N (1 ≤ N ≤ 34) numbers S1, ..., SN (-20,000,000 ≤ Si ≤ 20,000,000), determine how many subsets of S (including the empty one) have a sum between A and B (-500,000,000 ≤ A ≤ B ≤ 500,000,000), inclusive.

### Input

The first line of standard input contains the three integers N, A, and B. The following N lines contain S1 through SN, in order.

### Output

Print a single integer to standard output representing the number of subsets satisfying the above property. Note that the answer may overflow a 32-bit integer.

### Example

**Input:**

3 -1 2

1

-2

3

**Output:**

5

The following 5 subsets have a sum between -1 and 2:

* 0 = 0 (the empty subset)
* 1 = 1
* 1 + (-2) = -1
* -2 + 3 = 1
* 1 + (-2) + 3 = 2

## Solution:

#include <bits/stdc++.h>

using namespace std;

void solve(vector<int> arr, int start, int end, vector<long long> &v){

int n = end-start+1;

for(int i=0;i<(1<<n);i++){

int j = start, x = i;

long long sum=0;

while(x){

if(x&1)

sum += arr[j];

j++;

x >>= 1;

}

v.push\_back(sum);

}

}

int main()

{

int n,a,b;

cin>>n>>a>>b;

vector<int> arr(n);

for(int i=0;i<n;i++)

cin>>arr[i];

vector<long long> v1,v2;

//store the sum of all subsets of first half of the array in v1

solve(arr, 0, (n/2)-1, v1);

//store the sum of all subsets of second half of the array in v2

solve(arr, n/2, n-1, v2);

//sort v2 so that we can apply binary search on it

sort(v2.begin(),v2.end());

long long ans=0;

for(int i=0;i<v1.size();i++){

int low = lower\_bound(v2.begin(), v2.end(), a-v1[i]) - v2.begin();

int high = upper\_bound(v2.begin(), v2.end(), b-v1[i]) - v2.begin();

ans += (high-low);

}

cout<<ans<<endl;

return 0;

}

**Time Complexity:** O( (2^(n-1)) \* (n-1)), bcz for loop will execute 2^(n-1) times and inside that lower bound and upper bound function will take O(log(2^(n-1)) time which is equal to O((n-1)\*log2) which is equal to O(n-1) so, overall complexity will be O( (2^(n-1)) \* (n-1)).

**Space Complexity:** O(2^n)

# Find the inversion count

## Same as question 16 of Arrays.

# Implement Merge-sort in-place

Implement [Merge Sort](http://www.geeksforgeeks.org/merge-sort/) i.e. standard implementation keeping the sorting algorithm as in-place.   
In-place means it does not occupy extra memory for merge operation as in the standard case.

**Examples:**

***Input:****arr[] = {2, 3, 4, 1}****Output:****1 2 3 4*

***Input:****arr[] = {56, 2, 45}****Output:****2 45 56*

## Solution:

**Approach 1:**

* Maintain two pointers that point to the start of the segments which have to be merged.
* Compare the elements at which the pointers are present.
* If *element1 < element2* then *element1* is at right position, simply increase *pointer1*.
* Else shift all the elements between element1 and *element2(including element1 but excluding element2)*right by 1 and then place the element2 in the previous place*(i.e. before shifting right)*of element1. Increment all the pointers by *1*.

Below is the implementation of the above approach:

// C++ program in-place Merge Sort

#include <iostream>

using namespace std;

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

// Inplace Implementation

void merge(int arr[], int start, int mid, int end)

{

int start2 = mid + 1;

// If the direct merge is already sorted

if (arr[mid] <= arr[start2]) {

return;

}

// Two pointers to maintain start

// of both arrays to merge

while (start <= mid && start2 <= end) {

// If element 1 is in right place

if (arr[start] <= arr[start2]) {

start++;

}

else {

int value = arr[start2];

int index = start2;

// Shift all the elements between element 1

// element 2, right by 1.

while (index != start) {

arr[index] = arr[index - 1];

index--;

}

arr[start] = value;

// Update all the pointers

start++;

mid++;

start2++;

}

}

}

/\* 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 r

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++)

cout <<" "<< A[i];

cout <<"\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]);

mergeSort(arr, 0, arr\_size - 1);

printArray(arr, arr\_size);

return 0;

}

**Output**

5 6 7 11 12 13

Note: [Time Complexity](http://penguin.ewu.edu/cscd300/Topic/AdvSorting/MergeSorts/InPlace.html#Top) of above approach is O(n2\* log(n)) because merge is O(n2). Time complexity of [standard merge sort](https://www.geeksforgeeks.org/merge-sort/) is less, O(n Log n).

**Approach 2:**The idea: We start comparing elements that are far from each other rather than adjacent. Basically we are using shell sorting to [merge two sorted arrays with O(1) extra space](https://www.geeksforgeeks.org/efficiently-merging-two-sorted-arrays-with-o1-extra-space/).

mergeSort():

* Calculate mid two split the array in two halves(left sub-array and right sub-array)
* Recursively call merge sort on left sub-array and right sub-array to sort them
* Call merge function to merge left sub-array and right sub-array

merge():

* For every pass, we calculate the gap and compare the elements towards the right of the gap.
* Initiate the gap with ceiling value of n/2 where n is the combined length of left and right sub-array.
* Every pass, the gap reduces to the ceiling value of gap/2.
* Take a pointer i to pass the array.
* Swap the ith and (i+gap)th elements if (i+gap)th element is smaller than(or greater than when sorting in decreasing order) ith element.
* Stop when (i+gap) reaches n.

***Input:****10, 30, 14, 11, 16, 7, 28*

*Note: Assume left and right subarrays has been sorted so we are merging sorted subarrays [10, 14, 30] and [7, 11, 16, 28]*

*Start with*

*gap =  ceiling of n/2 = 7/2 = 4*

*[This gap is for whole merged array]*

***10****, 14, 30, 7,****11****, 16, 28*

*10,****14****, 30, 7, 11,****16****, 28*

*10, 14,****30****, 7, 11, 16,****28***

*10, 14, 28, 7, 11, 16, 30*

*gap =  ceiling of 4/2 = 2*

***10****, 14,****28****, 7, 11, 16, 30*

*10,****14****, 28,****7****, 11, 16, 30*

*10, 7,****28****, 14,****11****, 16, 30*

*10, 7, 11,****14****, 28,****16****, 30*

*10, 7, 11, 14,****28****, 16,****30***

*gap =  ceiling of 2/2 = 1*

***10****,****7****, 11, 14, 28, 16, 30*

*7,****10****,****11****, 14, 28, 16, 30*

*7, 10,****11****,****14****, 28, 16, 30*

*7, 10, 11,****14****,****28****, 16, 30*

*7, 10, 11, 14,****28****,****16****, 30*

*7, 10, 11, 14, 16,****28****,****30***

***Output:****7, 10, 11, 14, 16, 28, 30*

Below is the implementation of the above approach:

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Calculating next gap

int nextGap(int gap)

{

if (gap <= 1)

return 0;

return (int)ceil(gap / 2.0);

}

// Function for swapping

void swap(int nums[], int i, int j)

{

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

// Merging the subarrays using shell sorting

// Time Complexity: O(nlog n)

// Space Complexity: O(1)

void inPlaceMerge(int nums[], int start,

int end)

{

int gap = end - start + 1;

for(gap = nextGap(gap);

gap > 0; gap = nextGap(gap))

{

for(int i = start; i + gap <= end; i++)

{

int j = i + gap;

if (nums[i] > nums[j])

swap(nums, i, j);

}

}

}

// merge sort makes log n recursive calls

// and each time calls merge()

// which takes nlog n steps

// Time Complexity: O(n\*log n + 2((n/2)\*log(n/2)) +

// 4((n/4)\*log(n/4)) +.....+ 1)

// Time Complexity: O(logn\*(n\*log n))

// i.e. O(n\*(logn)^2)

// Space Complexity: O(1)

void mergeSort(int nums[], int s, int e)

{

if (s == e)

return;

// Calculating mid to slice the

// array in two halves

int mid = (s + e) / 2;

// Recursive calls to sort left

// and right subarrays

mergeSort(nums, s, mid);

mergeSort(nums, mid + 1, e);

inPlaceMerge(nums, s, e);

}

// Driver Code

int main()

{

int nums[] = { 12, 11, 13, 5, 6, 7 };

int nums\_size = sizeof(nums) / sizeof(nums[0]);

mergeSort(nums, 0, nums\_size-1);

for(int i = 0; i < nums\_size; i++)

{

cout << nums[i] << " ";

}

return 0;

}

**Output**

5 6 7 11 12 13

**Time Complexity:**O(log n\*nlog n)

**Note:**mergeSort method makes log n recursive calls and each time merge is called which takes n log n time to merge 2 sorted sub-arrays

**Approach 3:**Here we use the below technique:

Suppose we have a number A and we want to

convert it to a number B and there is also a

constraint that we can recover number A any

time without using other variable.To achieve

this we chose a number N which is greater

than both numbers and add B\*N in A.

so A --> A+B\*N

To get number B out of (A+B\*N)

we divide (A+B\*N) by N (A+B\*N)/N = B.

To get number A out of (A+B\*N)

we take modulo with N (A+B\*N)%N = A.

-> In short by taking modulo

we get old number back and taking divide

we new number.

**mergeSort():**

* Calculate mid two split the array into two halves(left sub-array and right sub-array)
* Recursively call merge sort on left sub-array and right sub-array to sort them
* Call merge function to merge left sub-array and right sub-array

**merge():**

* We first find the maximum element of both sub-array and increment it one to avoid collision of 0 and maximum element during modulo operation.
* The idea is to traverse both sub-arrays from starting simultaneously. One starts from l till m and another starts from m+1 till r. So, We will initialize 3 pointers say i, j, k.
* i will move from l till m; j will move from m+1 till r; k will move from l till r.
* Now update value a[k] by adding min(a[i],a[j])\*maximum\_element.
* Then also update those elements which are left in both sub-arrays.
* After updating all the elements divide all the elements by maximum\_element so we get the updated array back.

Below is the implementation of the above approach:

// C++ program in-place Merge Sort

#include <bits/stdc++.h>

using namespace std;

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

// Inplace Implementation

void mergeInPlace(int a[], int l, int m, int r)

{

// increment the maximum\_element by one to avoid

// collision of 0 and maximum element of array in modulo

// operation

int mx = max(a[m], a[r]) + 1;

int i = l, j = m + 1, k = l;

while (i <= m && j <= r && k <= r) {

// recover back original element to compare

int e1 = a[i] % mx;

int e2 = a[j] % mx;

if (e1 <= e2) {

a[k] += (e1 \* mx);

i++;

k++;

}

else {

a[k] += (e2 \* mx);

j++;

k++;

}

}

// process those elements which are left in the array

while (i <= m) {

int el = a[i] % mx;

a[k] += (el \* mx);

i++;

k++;

}

while (j <= r) {

int el = a[j] % mx;

a[k] += (el \* mx);

j++;

k++;

}

// finally update elements by dividing with maximum

// element

for (int i = l; i <= r; i++)

a[i] /= mx;

}

/\* 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 r

int m = l + (r - l) / 2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

mergeInPlace(arr, l, m, r);

}

}

// Driver Code

int main()

{

int nums[] = { 12, 11, 13, 5, 6, 7 };

int nums\_size = sizeof(nums) / sizeof(nums[0]);

mergeSort(nums, 0, nums\_size - 1);

for (int i = 0; i < nums\_size; i++) {

cout << nums[i] << " ";

}

return 0;

}

**Output**

5 6 7 11 12 13

**Time Complexity:**O(n log n)  
**Note:** Time Complexity of above approach is O(n2) because merge is O(n). Time complexity of standard merge sort is  O(n log n).

**Approach 4**: Here we use the following technique to perform an in-place merge

Given 2 adjacent sorted sub-arrays within an array (hereafter

named A and B for convenience), appreciate that we can swap

some of the last portion of A with an equal number of elements

from the start of B, such that after the exchange, all of the

elements in A are less than or equal to any element in B.

After this exchange, this leaves with the A containing 2 sorted

sub-arrays, being the first original portion of A, and the first

original portion of B, and sub-array B now containing 2 sorted

sub-arrays, being the final original portion of A followed by

the final original portion of B

We can now recursively call the merge operation with the 2

sub-arrays of A, followed by recursively calling the merge

operation with the 2 sub-arrays of B

We stop the recursion when either A or B are empty, or when

either sub-array is small enough to efficiently merge into

the other sub-array using insertion sort.

The above procedure naturally lends itself to the following implementation of an in-place merge sort.

**merge():**

* Hereafter, for convenience, we’ll refer to the first sub-array as A, and the second sub-array as B
* If either A or B are empty, or if the first element B is not less than the last element of A then we’re done
* If the length of A is small enough and if it’s length is less than the length of B, then use insertion sort to merge A into B and return
* If the length of B is small enough then use insertion sort to merge B into A and return
* Find the location in A where we can exchange the remaining portion of A with the first-portion of B, such that all the elements in A are less than or equal to any element in B
* Perform the exchange between A and B
* Recursively call **merge()** on the 2 sorted sub-arrays now residing in A
* Recursively call **merge()**on the 2 sorted sub-arrays now residing in B

**merge\_sort():**

* Split the array into two halves(left sub-array and right sub-array)
* Recursively call **merge\_sort()** on left sub-array and right sub-array to sort them
* Call merge function to merge left sub-array and right sub-array

// Merge In Place in C++

#include <iostream>

using namespace std;

#define \_\_INSERT\_THRESH 5

#define \_\_swap(x, y) (t = \*(x), \*(x) = \*(y), \*(y) = t)

// Both sorted sub-arrays must be adjacent in 'a'

// 'an' is the length of the first sorted section in 'a'

// 'bn' is the length of the second sorted section in 'a'

static void merge(int\* a, size\_t an, size\_t bn)

{

int \*b = a + an, \*e = b + bn, \*s, t;

// Return right now if we're done

if (an == 0 || bn == 0 || !(\*b < \*(b - 1)))

return;

// Do insertion sort to merge if size of sub-arrays are

// small enough

if (an < \_\_INSERT\_THRESH && an <= bn) {

for (int \*p = b, \*v; p > a;

p--) // Insert Sort A into B

for (v = p, s = p - 1; v < e && \*v < \*s;

s = v, v++)

\_\_swap(s, v);

return;

}

if (bn < \_\_INSERT\_THRESH) {

for (int \*p = b, \*v; p < e;

p++) // Insert Sort B into A

for (s = p, v = p - 1; s > a && \*s < \*v;

s = v, v--)

\_\_swap(s, v);

return;

}

// Find the pivot points. Basically this is just

// finding the point in 'a' where we can swap in the

// first part of 'b' such that after the swap the last

// element in 'a' will be less than or equal to the

// least element in 'b'

int \*pa = a, \*pb = b;

for (s = a; s < b && pb < e; s++)

if (\*pb < \*pa)

pb++;

else

pa++;

pa += b - s;

// Swap first part of b with last part of a

for (int \*la = pa, \*fb = b; la < b; la++, fb++)

\_\_swap(la, fb);

// Now merge the two sub-array pairings

merge(a, pa - a, pb - b);

merge(b, pb - b, e - pb);

} // merge\_array\_inplace

#undef \_\_swap

#undef \_\_INSERT\_THRESH

// Merge Sort Implementation

void merge\_sort(int\* a, size\_t n)

{

size\_t m = (n + 1) / 2;

// Sort first and second halves

if (m > 1)

merge\_sort(a, m);

if (n - m > 1)

merge\_sort(a + m, n - m);

// Now merge the two sorted sub-arrays together

merge(a, m, n - m);

}

// Function to print an array

void print\_array(int a[], size\_t n)

{

if (n > 0) {

cout <<" "<< a[0];

for (size\_t i = 1; i < n; i++)

cout <<" "<< a[i];

}

cout <<"\n";

}

// Driver program to test sort utility

int main()

{

int a[] = { 3, 16, 5, 14, 8, 10, 7, 15,

1, 13, 4, 9, 12, 11, 6, 2 };

size\_t n = sizeof(a) / sizeof(a[0]);

merge\_sort(a, n);

print\_array(a, n);

return 0;

}

**Output**

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

**Time Complexity of merge()**:  *Worst Case:* O(n^2),  *Average:* O(n log n),  *Best:* O(1)

**Time Complexity of merge\_sort()**function:  *Overall:* O(log n) for the recursion alone, due to always evenly dividing the array in 2

**Time Complexity of** **merge\_sort()**overall:  *Worst Case:*O(n^2 log n),  *Average:* O(n (log n)^2), *Best:* O(log n)

The worst-case occurs when every sub-array exchange within **merge()** results in just *\_INSERT\_THRESH-1* elements being exchanged

# Partitioning and Sorting Arrays with Many Repeated Entries

## 1. Overview

The run-time complexity of algorithms is often dependent on the nature of the input.

In this tutorial, we’ll see how the **trivial implementation of the Quicksort algorithm has a poor performance for repeated elements**.

Further, we’ll learn a few Quicksort variants to efficiently partition and sort inputs with a high density of duplicate keys.

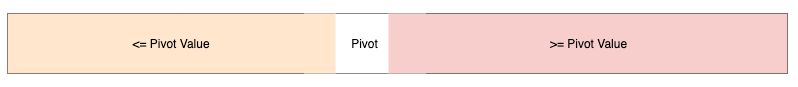
## 2. Trivial Quicksort

[Quicksort](https://www.baeldung.com/java-quicksort) is an efficient sorting algorithm based on the divide and conquer paradigm. Functionally speaking, it **operates in-place on the input array and rearranges the elements with simple comparison and swap operations**.

### 2.1. Single-pivot Partitioning

A trivial implementation of the Quicksort algorithm relies heavily on a single-pivot partitioning procedure. In other words, partitioning divides the array A=[ap, ap+1, ap+2,…, ar] into two parts A[p..q] and A[q+1..r] such that:

* All elements in the first partition, **A[p..q] are lesser than or equal to the pivot value A[q]**
* All elements in the second partition, **A[q+1..r] are greater than or equal to the pivot value A[q]**

[](https://www.baeldung.com/wp-content/uploads/2020/01/trivial_quicksort.png)

After that, the two partitions are treated as independent input arrays and fed themselves to the Quicksort algorithm.

### 2.2. Performance with Repeated Elements

Let’s say we have an array A = [4, 4, 4, 4, 4, 4, 4] that has all equal elements.

On partitioning this array with the single-pivot partitioning scheme, we'll get two partitions. The first partition will be empty, while the second partition will have N-1 elements. Further, **each subsequent invocation of the partition procedure will reduce the input size by only one**.

Since the partition procedure has linear time complexity, the overall time complexity, in this case, is quadratic. This is the worst-case scenario for our input array.

## 3. Three-way Partitioning

To efficiently sort an array having a high number of repeated keys, we can choose to handle the equal keys more responsibly. The idea is to place them in the right position when we first encounter them. So, what we're looking for is a three partition state of the array:

* The left-most partition contains elements which are strictly less than the partitioning key
* **The** **middle partition contains all elements which are equal to the partitioning key**
* The right-most partition contains all elements which are strictly greater than the partitioning key

[](https://www.baeldung.com/wp-content/uploads/2020/01/3-way-partition-preview.png)

We'll now dive deeper into a couple of approaches that we can use to achieve three-way partitioning.

## 4. Dijkstra's Approach

Dijkstra's approach is an effective way of doing three-way partitioning. To understand this, let's look into a classic programming problem.

### 4.1. Dutch National Flag Problem

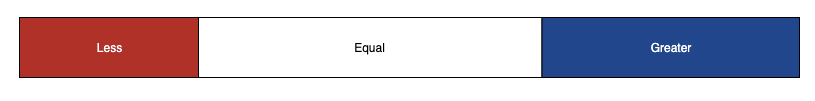
Inspired by the tricolor [flag of the Netherlands](https://en.wikipedia.org/wiki/Flag_of_the_Netherlands), [Edsger Dijkstra](https://en.wikipedia.org/wiki/Edsger_Dijkstra) proposed a programming problem called the [Dutch National Flag Problem](https://en.wikipedia.org/wiki/Dutch_national_flag_problem) (DNF).

In a nutshell, it's **a rearrangement problem where we're given balls of three colors placed randomly in a line, and we're asked to group the same colored balls together**. Moreover, the rearrangement must ensure that groups follow the correct order.

Interestingly, the DNF problem makes a striking analogy with the 3-way partitioning of an array with repeated elements.

We can categorize all the numbers of an array into three groups with respect to a given key:

* The Red group contains all elements that are strictly lesser than the key
* The White group contains all elements that are equal to the key
* The Blue group contains all elements that strictly greater than the key

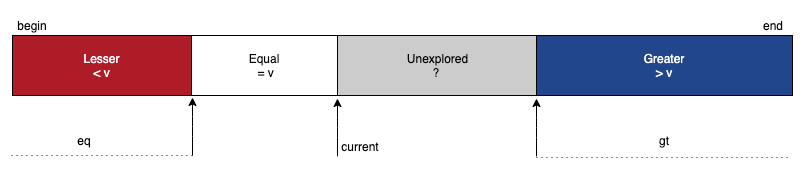
[](https://www.baeldung.com/wp-content/uploads/2020/01/DNF_Partition_1.png)

### 4.2. Algorithm

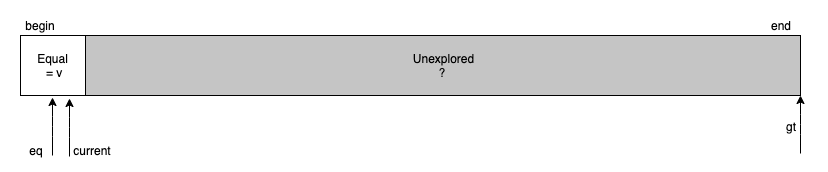
One of the approaches to solve the DNF problem is to pick the first element as the partitioning key and scan the array from left to right. As we check each element, we move it to its correct group, namely Lesser, Equal, and Greater.

To keep track of our partitioning progress, we'd need the help of three pointers, namely lt, current, and gt. **At any point in time, the elements to the left of lt will be strictly less than the partitioning key, and the elements to the right of gt will be strictly greater than the key**.

Further, we'll use the current pointer for scanning, which means that all elements lying between the current and gt pointers are yet to be explored:

[](https://www.baeldung.com/wp-content/uploads/2020/01/DNF_Invariant.png)

To begin with, we can set lt and current pointers at the very beginning of the array and the gt pointer at the very end of it:

[](https://www.baeldung.com/wp-content/uploads/2020/01/DNF_Algo_1.png)

For each element read via the current pointer, we compare it with the partitioning key and take one of the three composite actions:

* If input[current] < key, then we exchange input[current] and input[lt] and increment both current and lt pointers
* If input[current] == key, then we increment current pointer
* If input[current] > key, then we exchange input[current] and input[gt] and decrement gt

Eventually, **we'll stop when the current and gt pointers cross each other**. With that, the size of the unexplored region reduces to zero, and we'll be left with only three required partitions.

### 4.3. Implementation

First, let's write a utility procedure named compare() to do a three-way comparison between two numbers:

**public** **static** **int** **compare**(**int** num1, **int** num2) {

**if** (num1 > num2)

**return** 1;

**else** **if** (num1 < num2)

**return** -1;

**else**

**return** 0;

}

Next, let's add a method called swap() to exchange elements at two indices of the same array:

**public** **static** **void** **swap**(**int**[] array, **int** position1, **int** position2) {

**if** (position1 != position2) {

**int** temp = array[position1];

array[position1] = array[position2];

array[position2] = temp;

}

}

To uniquely identify a partition in the array, we'll need its left and right boundary-indices. So, let's go ahead and create a Partition class:

**public** **class** **Partition** {

**private** **int** left;

**private** **int** right;

}

Now, we're ready to write our three-way partition() procedure:

**public** **static** Partition **partition**(**int**[] input, **int** begin, **int** end) {

**int** lt = begin, current = begin, gt = end;

**int** partitioningValue = input[begin];

**while** (current <= gt) {

**int** compareCurrent = compare(input[current], partitioningValue);

**switch** (compareCurrent) {

**case** -1:

swap(input, current++, lt++);

**break**;

**case** 0:

current++;

**break**;

**case** 1:

swap(input, current, gt--);

**break**;

}

}

**return** **new** Partition(lt, gt);

}

Finally, let's write a **quicksort() method that leverages our 3-way partitioning scheme to sort the left and right partitions recursively**:

**public** **static** **void** **quicksort**(**int**[] input, **int** begin, **int** end) {

**if** (end <= begin)

**return**;

Partition middlePartition = partition(input, begin, end);

quicksort(input, begin, middlePartition.getLeft() - 1);

quicksort(input, middlePartition.getRight() + 1, end);

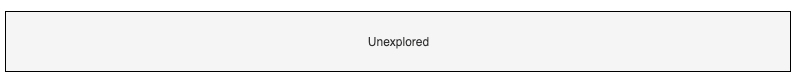
}

## 5. Bentley-McIlroy's Approach

[Jon Bentley](https://en.wikipedia.org/wiki/Jon_Bentley_(computer_scientist)) and [Douglas McIlroy](https://en.wikipedia.org/wiki/Douglas_McIlroy) co-authored an **optimized version of the Quicksort algorithm**. Let's understand and implement this variant in Java:

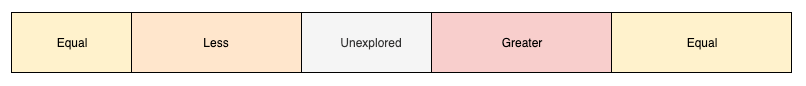
### 5.1. Partitioning Scheme

The crux of the algorithm is an iteration-based partitioning scheme. In the start, the entire array of numbers is an unexplored territory for us:

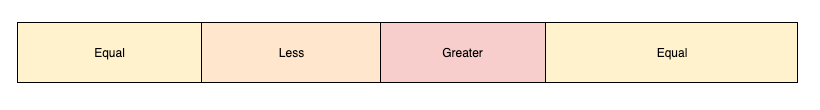
[](https://www.baeldung.com/wp-content/uploads/2020/01/Bentley-Unexplored.png)

We then start exploring the elements of the array from the left and right direction. Whenever we enter or leave the loop of exploration, **we can visualize the array as a composition of five regions**:

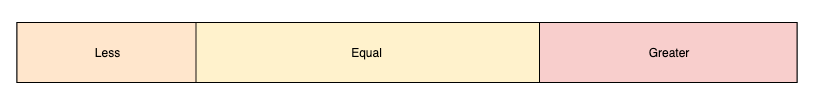
* On the extreme two ends, lies the regions having elements that are equal to the partitioning value
* The unexplored region stays in the center and its size keeps on shrinking with each iteration
* On the left of the unexplored region lies all elements lesser than the partitioning value
* On the right side of the unexplored region are elements greater than the partitioning value

[](https://www.baeldung.com/wp-content/uploads/2020/01/Bentley-Partitioning-Invariant.png)

Eventually, our loop of exploration terminates when there are no elements to be explored anymore. At this stage, the **size of the unexplored region is effectively zero**, and we're left with only four regions:

[](https://www.baeldung.com/wp-content/uploads/2020/01/Bentley-loop-termination.png)

Next, we **move all the elements from the two equal-regions in the center** so that there is only one equal-region in the center surrounding by the less-region on the left and the greater-region on the right. To do so, first, we swap the elements in the left equal-region with the elements on the right end of the less-region. Similarly, the elements in the right equal-region are swapped with the elements on the left end of the greater-region.

[](https://www.baeldung.com/wp-content/uploads/2020/01/Bentley-partition.png)

Finally, we'll be **left with only three partitions**, and we can further use the same approach to partition the less and the greater regions.

### 5.2. Implementation

In our recursive implementation of the three-way Quicksort, we'll need to invoke our partition procedure for sub-arrays that'll have a different set of lower and upper bounds. So, our partition() method must accept three inputs, namely the array along with its left and right bounds.

**public** **static** Partition **partition**(**int** input[], **int** begin, **int** end){

// returns partition window

}

For simplicity, we can **choose the partitioning value as the last element of the array**. Also, let's define two variables left=begin and right=end to explore the array inward.

Further, We'll also need to **keep track of the number of equal elements lying on the leftmost and rightmost**. So, let's initialize leftEqualKeysCount=0 and rightEqualKeysCount=0, and we're now ready to explore and partition the array.

First, we start moving from both the directions and **find an inversion** where an element on the left is not less than partitioning value, and an element on the right is not greater than partitioning value. Then, unless the two pointers left and right have crossed each other, we swap the two elements.

In each iteration, we move elements equal to partitioningValue towards the two ends and increment the appropriate counter:

**while** (**true**) {

**while** (input[left] < partitioningValue) left++;

**while** (input[right] > partitioningValue) {

**if** (right == begin)

**break**;

right--;

}

**if** (left == right && input[left] == partitioningValue) {

swap(input, begin + leftEqualKeysCount, left);

leftEqualKeysCount++;

left++;

}

**if** (left >= right) {

**break**;

}

swap(input, left, right);

**if** (input[left] == partitioningValue) {

swap(input, begin + leftEqualKeysCount, left);

leftEqualKeysCount++;

}

**if** (input[right] == partitioningValue) {

swap(input, right, end - rightEqualKeysCount);

rightEqualKeysCount++;

}

left++; right--;

}

In the next phase, we need to **move all the equal elements from the two ends in the center**. After we exit the loop, the left-pointer will be at an element whose value is not less than *partitioningValue*. Using this fact, we start moving equal elements from the two ends towards the center:

right = left - 1;

**for** (**int** k = begin; k < begin + leftEqualKeysCount; k++, right--) {

**if** (right >= begin + leftEqualKeysCount)

swap(input, k, right);

}

**for** (**int** k = end; k > end - rightEqualKeysCount; k--, left++) {

**if** (left <= end - rightEqualKeysCount)

swap(input, left, k);

}

In the last phase, we can return the boundaries of the middle partition:

**return** **new** Partition(right + 1, left - 1);

## 6. Algorithm Analysis

In general, the Quicksort algorithm has an average-case time complexity of O(n\*log(n)) and worst-case time complexity of O(n2). With a high density of duplicate keys, we almost always get the worst-case performance with the trivial implementation of Quicksort.

However, when we use the three-way partitioning variant of Quicksort, such as DNF partitioning or Bentley's partitioning, we're able to prevent the negative effect of duplicate keys. Further, as the density of duplicate keys increase, the performance of our algorithm improves as well. As a result, we get the best-case performance when all keys are equal, and we get a single partition containing all equal keys in linear time.

Nevertheless, we must note that we're essentially adding overhead when we switch to a three-way partitioning scheme from the trivial single-pivot partitioning.

For DNF based approach, the overhead doesn't depend on the density of repeated keys. So, if we use DNF partitioning for an array with all unique keys, then we'll get poor performance as compared to the trivial implementation where we're optimally choosing the pivot.

But, Bentley-McIlroy's approach does a smart thing as the overhead of moving the equal keys from the two extreme ends is dependent on their count. As a result, if we use this algorithm for an array with all unique keys, even then, we'll get reasonably good performance.

## 7. Conclusion

In this tutorial, we learned about the performance issues with the trivial implementation of the Quicksort algorithm when the input has a large number of repeated elements.

With a motivation to fix this issue, we **learned different three-way partitioning schemes** and how we can implement them in Java.