**Binary Search:**

**Regular Implementation Of Binary Search:**

// Returns location of key, or -1 if not found

int BinarySearch(int A[], int l, int r, int key)

{

int m;

while( l <= r )

{

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

if( A[m] == key ) // first comparison

return m;

if( A[m] < key ) // second comparison

l = m + 1;

else

r = m - 1;

}

return -1;

}

**Optimized Implementation Of Binary Search:**

int BinarySearch(int A[], int l, int r, int key)

{

int m;

while( r - l > 1 )

{

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

if( A[m] <= key )

l = m;

else

r = m;

}

if( A[l] == key )

return l;

else

return -1;

}

**The second one uses less number of comparisons.**

**Common Mistake In Binary Search:**int mid=l+(r-l)/2 is better than mid=(r+l)/2 in case of overflow handling.

**Interpolation Search:**

****

**Time complexity of Interpolation Search:**

O(log(logn))  
  
 **Jump Search:**

Like Binary Search, Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

**Exponential Search:**The name of this searching algorithm may be misleading as it works in O(Log n) time. The name comes from the way it searches an element.

The idea is to start with subarray size 1, compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.

Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration)

**(this is a good modification from jump Search)**

**Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time)**  
Given a function ‘int f(unsigned int x)’ which takes a non-negative integer ‘x’ as input and returns an integer as output. The function is monotonically increasing with respect to value of x, i.e., the value of f(x+1) is greater than f(x) for every input x. Find the value ‘n’ where f() becomes positive for the first time. Since f() is monotonically increasing, values of f(n+1), f(n+2),… must be positive and values of f(n-2), f(n-3), .. must be negative.

Find n in O(logn) time, you may assume that f(x) can be evaluated in O(1) time for any input x.

A simple solution is to start from i equals to 0 and one by one calculate value of f(i) for 1, 2, 3, 4 .. etc until we find a positive f(i). This works, but takes O(n) time.

Can we apply Binary Search to find n in O(Logn) time? We can’t directly apply Binary Search as we don’t have an upper limit or high index. The idea is to do repeated doubling until we find a positive value, i.e., check values of f() for following values until f(i) becomes positive.

#include <stdio.h>

int binarySearch(int low, int high); // prototype

// Let's take an example function as f(x) = x^2 - 10\*x - 20

// Note that f(x) can be any monotonically increasing function

int f(int x) { return (x\*x - 10\*x - 20); }

// Returns the value x where above function f() becomes positive

// first time.

int findFirstPositive()

{

// When first value itself is positive

if (f(0) > 0)

return 0;

// Find 'high' for binary search by repeated doubling

int i = 1;

while (f(i) <= 0)

i = i\*2;

// Call binary search

return binarySearch(i/2, i);

}

// Searches first positive value of f(i) where low <= i <= high

int binarySearch(int low, int high)

{

if (high >= low)

{

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

// If f(mid) is greater than 0 and one of the following two

// conditions is true:

// a) mid is equal to low

// b) f(mid-1) is negative

if (f(mid) > 0 && (mid == low || f(mid-1) <= 0))

return mid;

// If f(mid) is smaller than or equal to 0

if (f(mid) <= 0)

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

else // f(mid) > 0

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

}

/\* Return -1 if there is no positive value in given range \*/

return -1;

}

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

int main()

{

printf("The value n where f() becomes positive first is %d",

findFirstPositive());

return 0;

}

**Linear Search vs Binary Search:**

I know.

**Binary Search Vs Interpolation Search:**Now, well the elements of sorted array is well distributed, ternary search will work fine. In worst case, it can take upto O(loglogn)

**Binary Search vs Ternary Search:**

**Now, a standard binary search implementation is:**int BinarySearch(int A[], int l, int r, int key)

{

int m;

while( l <= r )

{

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

if( A[m] == key ) // first comparison

return m;

if( A[m] < key ) // second comparison

l = m + 1;

else

r = m - 1;

}

return -1;

}

**A standard ternary search implementation is:**

// A recursive ternary search function. It returns location of x in

// given array arr[l..r] is present, otherwise -1

int ternarySearch(int arr[], int l, int r, int x)

{

if (r >= l)

{

int mid1 = l + (r - l)/3;

int mid2 = mid1 + (r - l)/3;

// If x is present at the mid1

if (arr[mid1] == x) return mid1;

// If x is present at the mid2

if (arr[mid2] == x) return mid2;

// If x is present in left one-third

if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);

// If x is present in right one-third

if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);

// If x is present in middle one-third

return ternarySearch(arr, mid1+1, mid2-1, x);

}

// We reach here when element is not present in array

return -1;

}

Now, check the number of comparisons. And, find the recurrence relation:

**Number of Comparisons in case of binary search:**

int BinarySearch(int A[], int l, int r, int key)

{

int m;

while( l <= r )

{

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

if( A[m] == key ) **// first comparison**

return m;

if( A[m] < key ) **// second comparison**

l = m + 1;

else

r = m - 1;

}

return -1;

}

As you can see, in **binary search**, in one execution there are two comparisons.

Binary search’s recurrence relation is **T(n)=T(n/2)+2  
  
Number Of Comparisons In Case of Ternary Search:**

int ternarySearch(int arr[], int l, int r, int x)

{

if (r >= l)

{

int mid1 = l + (r - l)/3;

int mid2 = mid1 + (r - l)/3;

// If x is present at the mid1

if (arr[mid1] == x) return mid1;

**//first comparison**

// If x is present at the mid2

if (arr[mid2] == x) return mid2;

**//second comparison**

// If x is present in left one-third

if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);

**//third comparison**

// If x is present in right one-third

if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);

**//fourth comparison**

// If x is present in middle one-third

return ternarySearch(arr, mid1+1, mid2-1, x);

}

// We reach here when element is not present in array

return -1;

}

**So, Ternary Search’s recurrence relation is T(n)=T(n/3)+4  
  
Now, there’s a theorem named Master’s theorem:**what does is state:  
  
It states suppose, for some algorithm: the recurrence relation is:  
  


1. If  and c<logba , 
2. If  and c=logba,   
     
   the, case 2 case further be extended to, if  and and c=logba,



1. If  and c>logba, 

Now, go to the recurrence relation for binary search:

For binary search, the recurrence relation was: **T(n)=T(n/2)+2**

Now, logba is 0. (log21)

In binary search, there are 2Log2n + 1 comparisons in worst case. In ternary search, there are 4Log3n + 1 comparisons in worst case.

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions 2Log3n and Log2n . The value of 2Log3n can be written as (2 / Log23) \* Log2n . Since the value of (2 / Log23) is more than one, Ternary Search does more comparisons than Binary Search in worst case.

**General Problems:**

**Finding The missing Element:**

**Method 1:**

1. Get the sum of numbers

total = n\*(n+1)/2

2 Subtract all the numbers from sum and

you will get the missing number.

**Method 2:** 1) XOR all the array elements, let the result of XOR be X1.

2) XOR all numbers from 1 to n, let XOR be X2.

3) XOR of X1 and X2 gives the missing number.

**Search an element In A sorted and rotated array:**Example: input: arr[]={3,4,5,1,2};

Element to search=1

1. Find out pivot element and divide the array into two sub arrays
2. Now call binary search for one of the subarrays. From here, you can develop your own logic.

**Two Elements whose sum is closest to zero:**1) Sort all the elements of the input array.

2) Use two index variables l and r to traverse from left and right ends respectively. Initialize l as 0 and r as n-1.

3) sum = a[l] + a[r]

4) If sum is -ve, then l++

5) If sum is +ve, then r–

6) Keep track of abs min sum.

**Find the smallest and second smallest elements in an array:**1) Initialize both first and second smallest as INT\_MAX

first = second = INT\_MAX

2) Loop through all the elements.

a) If the current element is smaller than first, then update first

and second.

b) Else if the current element is smaller than second then update

Second

**Maximum And Minimum Of An Array:**

**METHOD 1 (Simple Linear Search)**

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

**METHOD 2 (Tournament Method)**

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

Pair MaxMin(array, array\_size)

if array\_size = 1

return element as both max and min

else if arry\_size = 2

one comparison to determine max and min

return that pair

else /\* array\_size > 2 \*/

recur for max and min of left half

recur for max and min of right half

one comparison determines true max of the two candidates

one comparison determines true min of the two candidates

return the pair of max and min

**METHOD 3 (Compare in Pairs)**

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

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

For rest of the elements, pick them in pairs and compare their

maximum and minimum with max and min respectively.

**k largest(or smallest) elements in an array | added Min Heap method:**Min heap **Ceiling in a sorted array:**

We need to modify binary search.

**Count Number Of Occurrences In A Sorted Array:**

Binary search based method for finding first and last occurrence.  
  
**Find the repeating and the missing:**I can do it.

**Find a fixed point in an array:**

**Problem statement:**Given an array of n distinct integers sorted in ascending order, write a function that returns a Fixed Point in the array, if there is any Fixed Point present in array, else returns -1. Fixed Point in an array is an index i such that arr[i] is equal to i. Note that integers in array can be negative.

First check whether middle element is Fixed Point or not. If it is, then return it; otherwise check whether index of middle element is greater than value at the index. If index is greater, then Fixed Point(s) lies on the right side of the middle point (obviously only if there is a Fixed Point). Else the Fixed Point(s) lies on left side.

**Find the maximum element in an array which is first increasing then decreasing:**

First check whether middle element is Fixed Point or not. If it is, then return it; otherwise check whether index of middle element is greater than value at the index. If index is greater, then Fixed Point(s) lies on the right side of the middle point (obviously only if there is a Fixed Point). Else the Fixed Point(s) lies on left side.

**Insert, Search and Delete in an unsorted array:**

In an unsorted array, the insert operation is faster as compared to sorted array because we don’t have to care about the position at which the element to be placed.

**Insert, Search and Delete in a sorted array:**

In a sorted array, the search operation can be performed by using binary search.

In an unsorted array, the insert operation is faster as compared to sorted array because we don’t have to care about the position at which the element to be placed.

In delete operation, the element to be deleted is searched using binary search and then delete operation is performed followed by shifting the elements

**Count 1’s in a sorted binary array:**

I can do that. We just need to find the first occurrence of 1 in a sorted array using modified version of binary search.

**Given a sorted array and a number x, find the pair in array whose sum is closest to x**

I can do that.

**Find Common Elements In Three Sorted Arrays:**merge algorithm

**Find k closest Values Of A Given Number In An Array:**finding the floor element and then two pointer.

**Find The First Repeating Element In A Sorted Array:**Use Hashing to solve this in O(n) time on average. The idea is to traverse the given array from right to left and update the minimum index whenever we find an element that has been visited on right side.

**Find a pair with the given difference:**

Given an unsorted array and a number n, find if there exists a pair of elements in the array whose difference is n.

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.

**(but, sorting is important)**

# Find the k most frequent words from a file:

We can use Trie and Min Heap to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in this post).

Trie and Min Heap are linked with each other by storing an additional field in Trie ‘indexMinHeap’ and a pointer ‘trNode’ in Min Heap. The value of ‘indexMinHeap’ is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, ‘indexMinHeap’ contains, index of the word in Min Heap. The pointer ‘trNode’ in Min Heap points to the leaf node corresponding to the word in Trie.

(remember this, trie and min heap combination)

**Kth smallest element in a row-wise and column-wise sorted 2D array:**

**(now, column number is important. As, after extracting an element the next element from same column is entered)**

The idea is to use min heap. Following are detailed step.

1) Build a min heap of elements from first row. A heap entry also stores row number and column number.

2) Do following k times.

…a) Get minimum element (or root) from min heap.

…b) Find row number and column number of the minimum element.

…c) Replace root **with the next element from same column and min-heapify the root.**

3) Return the last extracted root.  
  
**Find the closest pair from two sorted arrays:**

1) Initialize a variable diff as infinite (Diff is used to store the

difference between pair and x). We need to find the minimum diff.

2) Initialize two index variables l and r in the given sorted array.

(a) Initialize first to the leftmost index in ar1: l = 0

(b) Initialize second the rightmost index in ar2: r = n-1

3) Loop while l < m and r >= 0

(a) If abs(ar1[l] + ar2[r] - sum) < diff then

update diff and result

(b) Else if(ar1[l] + ar2[r] < sum ) then l++

(c) Else r--

4) Print the result.

**Kth Smallest/Largest element in a unsorted array:**1) Sort it. And find it.2)mIn heap based  
3) Quick select method

**Given an array of of size n and a number k, find all elements that appear more than n/k times**

Following is an interesting O(nk) solution:

We can solve the above problem in O(nk) time using O(k-1) extra space. Note that there can never be more than k-1 elements in output (because, we are finding elements that appear more than n/k times). There are mainly three steps in this algorithm.

1) Create a temporary array of size (k-1) to store elements and their counts (The output elements are going to be among these k-1 elements). Following is structure of temporary array elements.

struct eleCount

{

int element;

int count;

};

struct eleCount temp[];

This step takes O(k) time.

2) Traverse through the input array and update temp[] (add/remove an element or increase/decrease count) for every traversed element. The array temp[] stores potential (k-1) candidates at every step. This step takes O(nk) time.

3) Iterate through final (k-1) potential candidates (stored in temp[]). or every element, check if it actually has count more than n/k. This step takes O(nk) time.

The main step is step 2, how to maintain (k-1) potential candidates at every point? The steps used in step 2 are like famous game: Tetris. We treat each number as a piece in Tetris, which falls down in our temporary array temp[]. Our task is to try to keep the same number stacked on the same column (count in temporary array is incremented).

Consider k = 4, n = 9

Given array: 3 1 2 2 2 1 4 3 3

i = 0

3 \_ \_

temp[] has one element, 3 with count 1

i = 1

3 1 \_

temp[] has two elements, 3 and 1 with

counts 1 and 1 respectively

i = 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 1 respectively.

i = 3

- - 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 2 respectively.

i = 4

- - 2

- - 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 3 respectively.

i = 5

- - 2

- 1 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 2 and 3 respectively.

Now the question arises, what to do when temp[] is full and we see a new element – we remove the bottom row from stacks of elements, i.e., we decrease count of every element by 1 in temp[]. We ignore the current element.

i = 6

- - 2

- 1 2

temp[] has two elements, 1 and 2 with

counts as 1 and 2 respectively.

i = 7

- 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 2 respectively.

i = 8

3 - 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 2, 1 and 2 respectively.

Finally, we have at most k-1 numbers in temp[]. The elements in temp are {3, 1, 2}. Note that the counts in temp[] are useless now, the counts were needed only in step 2. Now we need to check whether the actual counts of elements in temp[] are more than n/k (9/4) or not. The elements 3 and 2 have counts more than 9/4. So we print 3 and 2.

// A C++ program to print elements with count more than n/k

#include<iostream>

using namespace std;

// A structure to store an element and its current count

struct eleCount

{

int e; // Element

int c; // Count

};

// Prints elements with more than n/k occurrences in arr[] of

// size n. If there are no such elements, then it prints nothing.

void moreThanNdK(int arr[], int n, int k)

{

// k must be greater than 1 to get some output

if (k < 2)

return;

/\* Step 1: Create a temporary array (contains element

and count) of size k-1. Initialize count of all

elements as 0 \*/

struct eleCount temp[k-1];

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

temp[i].c = 0;

/\* Step 2: Process all elements of input array \*/

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

{

int j;

/\* If arr[i] is already present in

the element count array, then increment its count \*/

for (j=0; j<k-1; j++)

{

if (temp[j].e == arr[i])

{

temp[j].c += 1;

break;

}

}

/\* If arr[i] is not present in temp[] \*/

if (j == k-1)

{

int l;

/\* If there is position available in temp[], then place

arr[i] in the first available position and set count as 1\*/

for (l=0; l<k-1; l++)

{

if (temp[l].c == 0)

{

temp[l].e = arr[i];

temp[l].c = 1;

break;

}

}

/\* If all the position in the temp[] are filled, then

decrease count of every element by 1 \*/

if (l == k-1)

for (l=0; l<k; l++)

temp[l].c -= 1;

}

}

/\*Step 3: Check actual counts of potential candidates in temp[]\*/

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

{

// Calculate actual count of elements

int ac = 0; // actual count

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

if (arr[j] == temp[i].e)

ac++;

// If actual count is more than n/k, then print it

if (ac > n/k)

cout << "Number:" << temp[i].e

<< " Count:" << ac << endl;

}

}

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

int main()

{

cout << "First Test\n";

int arr1[] = {4, 5, 6, 7, 8, 4, 4};

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

int k = 3;

moreThanNdK(arr1, size, k);

cout << "\nSecond Test\n";

int arr2[] = {4, 2, 2, 7};

size = sizeof(arr2)/sizeof(arr2[0]);

k = 3;

moreThanNdK(arr2, size, k);

cout << "\nThird Test\n";

int arr3[] = {2, 7, 2};

size = sizeof(arr3)/sizeof(arr3[0]);

k = 2;

moreThanNdK(arr3, size, k);

cout << "\nFourth Test\n";

int arr4[] = {2, 3, 3, 2};

size = sizeof(arr4)/sizeof(arr4[0]);

k = 3;

moreThanNdK(arr4, size, k);

return 0;

}

**Binary Search for Rational Numbers without using floating point arithmetic:**

// C program for Binary Search for Rationalnal Numbers

// without using floating point arithmetic

#include <stdio.h>

struct Rational

{

int p;

int q;

};

// Utility function to compare two Rationalnal numbers

// 'a' and 'b'. It returns

// 0 --> When 'a' and 'b' are same

// 1 --> When 'a' is greater

//-1 --> When 'b' is greate

int compare(struct Rational a, struct Rational b)

{

// If a/b == c/d then a\*d = b\*c:

// method to ignore division

**if (a.p \* b.q == a.q \* b.p)**

**return 0;**

**if (a.p \* b.q > a.q \* b.p)**

**return 1;**

**return -1;**

}

// Returns index of x in arr[l..r] if it is present, else

// returns -1. It mainly uses Binary Search.

int binarySearch(struct Rational arr[], int l, int r,

struct Rational x)

{

if (r >= l)

{

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

// If the element is present at the middle itself

if (compare(arr[mid], x) == 0) return mid;

// If element is smaller than mid, then it can

// only be present in left subarray

if (compare(arr[mid], x) > 0)

return binarySearch(arr, l, mid-1, x);

// Else the element can only be present in right

// subarray

return binarySearch(arr, mid+1, r, x);

}

return -1;

}

// Driver method

int main()

{

Rational arr[] = {{1, 5}, {2, 3}, {3, 2}, {13, 2}};

Rational x = {3, 2};

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

printf("Element found at index %d",

binarySearch(arr, 0, n-1, x));

}

**Median Of Two Sorted Arrays Of Equal Size:  
  
Method 1 (Simply count while Merging)  
  
Method 2 (By comparing the medians of two arrays)**

1) Calculate the medians m1 and m2 of the input arrays ar1[]

and ar2[] respectively.

2) If m1 and m2 both are equal then we are done.

return m1 (or m2)

1. If m1 is greater than m2, then median is present in one of the below two subarrays.

a) From first element of ar1 to m1 (ar1[0...|\_n/2\_|])

b) From m2 to last element of ar2 (ar2[|\_n/2\_|...n-1])

4) If m2 is greater than m1, then median is present in one of the below two subarrays.

a) From m1 to last element of ar1 (ar1[|\_n/2\_|...n-1])

b) From first element of ar2 to m2 (ar2[0...|\_n/2\_|])

5) Repeat the above process until size of both the subarrays becomes 2.

6) If size of the two arrays is 2 then use below formula to get

the median.

Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

// A divide and conquer based efficient solution to find median

// of two sorted arrays of same size.

#include<bits/stdc++.h>

using namespace std;

int median(int [], int); /\* to get median of a sorted array \*/

/\* This function returns median of ar1[] and ar2[].

Assumptions in this function:

Both ar1[] and ar2[] are sorted arrays

Both have n elements \*/

int getMedian(int ar1[], int ar2[], int n)

{

**/\* return -1 for invalid input \*/**

if (n <= 0)

return -1;

if (n == 1)

return (ar1[0] + ar2[0])/2;

**//obvious, since, 2n becomes even**

if (n == 2)

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

int m1 = median(ar1, n); /\* get the median of the first array \*/

int m2 = median(ar2, n); /\* get the median of the second array \*/

/\* If medians are equal then return either m1 or m2 \*/

if (m1 == m2)

return m1;

/\* if m1 < m2 then median must exist in ar1[m1....] and

ar2[....m2] \*/

if (m1 < m2)

{

if (n % 2 == 0)

return getMedian(ar1 + n/2 - 1, ar2, n - n/2 +1);

return getMedian(ar1 + n/2, ar2, n - n/2);

}

/\* if m1 > m2 then median must exist in ar1[....m1] and

ar2[m2...] \*/

if (n % 2 == 0)

return getMedian(ar2 + n/2 - 1, ar1, n - n/2 + 1);

return getMedian(ar2 + n/2, ar1, n - n/2);

}

/\* Function to get median of a sorted array \*/

int median(int arr[], int n)

{

if (n%2 == 0)

return (arr[n/2] + arr[n/2-1])/2;

else

return arr[n/2];

}

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

int main()

{

int ar1[] = {1, 2, 3, 6};

int ar2[] = {4, 6, 8, 10};

int n1 = sizeof(ar1)/sizeof(ar1[0]);

int n2 = sizeof(ar2)/sizeof(ar2[0]);

if (n1 == n2)

printf("Median is %d", getMedian(ar1, ar2, n1));

else

printf("Doesn't work for arrays of unequal size");

return 0;

}

Now, check the cases.

If n<=0 it’s an erroneous case.

If n=1 then we have two elements in the combined array. So, it would be (arr1[0]+arr2[0])/2

Otherwise, we find the median of two arrays. Now, it is easy.

int median(int arr[], int n)

{

if (n%2 == 0)

return (arr[n/2] + arr[n/2-1])/2;

else

return arr[n/2];

}

This can return the median for an array.

Now, there are three conditions:

Say, m1 is the median of first array and m2 is the median of second array:  
  
Now, if m1==m2 it will be the result.

Now, if m1>m2   
  
 a) From first element of ar1 to m1 (ar1[0...|\_n/2\_|])

b) From m2 to last element of ar2 (ar2[|\_n/2\_|...n-1])

If m2 is greater than m1, then median is present in one of the below two subarrays.

a) From m1 to last element of ar1 (ar1[|\_n/2\_|...n-1])

b) From first element of ar2 to m2 (ar2[0...|\_n/2\_|])

Now, even for each of these two conditions, two sub-conditions exist:

If current n is even

If current n is odd.

For instance, consider the condition where m1<m2

if (m1 < m2)

{

if (n % 2 == 0)

{

return getMedian(ar1 + n/2 - 1, ar2, n - n/2 +1);

}

return getMedian(ar1 + n/2, ar2, n - n/2);

}

getMedian(ar1 + n/2 - 1, ar2, n - n/2 +1);

Now, if n is even then we see, n-n/2+1 elements for our subarray for each of the two arrays.

If n is odd, we see n-n/2 which again odd (and bigger than n/2) for each of the subarray.

Now, it is actually depended on the median calculation, Since, median of an array of size n is calculated differently. (median of arr[n/2] and arr[n/2-1] if n is even and arr[n/2] is n is odd.)

Now, see, we n/2-1 in 0 indexing is actually n/2 in 1 indexing and n/2 is n/2+1

Now, you will see, if m1<m2 and n%2==0 we are not including the arr1[n/2-1] in recursive calculation. Since, (arr1[n/2-1]+arr[n/2])<(arr2[n/2-1]+arr2[n/2]))

So, when we are recursively going for second half of arr1 we are not including arr1[n/2-1] and when we are recursively going to first half of arr2 (because, that’s what we do in case of m1<m2) we are not including arr2[n/2].

Similar kind of observations are to be done in cases.

**Median of Two Sorted Arrays Of Different Size:**

This is an extension of median of two sorted arrays of equal size problem. Here we handle arrays of unequal size also.

The approach discussed in this post is similar to method 2 of equal size post. The basic idea is same, we find the median of two arrays and compare the medians to discard almost half of the elements in both arrays. Since the number of elements may differ here, there are many base cases that need to be handled separately. Before we proceed to complete solution, let us first talk about all base cases.

Let the two arrays be A[N] and B[M]. In the following explanation, it is assumed that N is smaller than or equal to M.

**Base cases:**

The smaller array has only one element

**Other cases:**

Case 0: N = 0, M = 2

Case 1: N = 1, M = 1.

Case 2: N = 1, M is odd

Case 3: N = 1, M is even

The smaller array has only two elements

Case 4: N = 2, M = 2

Case 5: N = 2, M is odd

Case 6: N = 2, M is even

**Base case:**

That means either N=0 M=1 or N=1 M=0

Now, if one array does not contain any element and another one only contains one element, we return that one element.

**There are two other base cases: (actually)**

**And what are those?**

**If N=0 or M=0**

If N=0 and M is even then median would be sum(arr2[M/2]+arr2[M/2-1])/2

(even case, Now, check, it’s M/2 and M/2-1 not, M/2 and M/2+1)

If N=0 and M is odd then median would be arr2[(M+1)/2];

And from here we can surely guess the cases for M=0

**Case 1:** N=1 M=1

It will be (arr1[0]+arr2[0])/2;

**Case 2:** N=1 M is even

Now, there will be two medians for arr2. that is arr2[M/2] and arr2[M/2-1]

 in this case, find the median of three elements arr2[ M / 2 – 1 ], arr2[ M / 2] and arr1[ 0 ].

**Case 3:** N=even M is 1.

We can do that.

**Case 4:** N is 2 M is 2

Median of four elements.

**Case 5:** N=2 M is odd. **(this case is tricky and important)**

The median is given by median of following three elements: arr2[M/2], max(arr1[0], arr2[M/2 – 1]), min(arr1[1], arr2[M/2 + 1]).

**Case 6:** M is 2 N is odd.

We know how to deal with it.

**Case 7:** Remaining cases:

Once we have handled the above base cases, following is the remaining process.

1) Find the middle item of A[] and middle item of B[].

…..1.1) If the middle item of A[] is greater than middle item of B[], ignore the last half of A[], let length of ignored part is idx. Also, cut down B[] by idx from the start.

…..1.2) else, ignore the first half of A[], let length of ignored part is idx. Also, cut down B[] by idx from the last.

**// A C++ program to find median of two sorted arrays of**

**// unequal sizes**

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

**using namespace std;**

**// A utility function to find median of two integers**

**float MO2(int a, int b)**

**{ return ( a + b ) / 2.0; }**

**// A utility function to find median of three integers**

**float MO3(int a, int b, int c)**

**{**

**return a + b + c - max(a, max(b, c))**

**- min(a, min(b, c));**

**}**

**// A utility function to find median of four integers**

**float MO4(int a, int b, int c, int d)**

**{**

**int Max = max( a, max( b, max( c, d ) ) );**

**int Min = min( a, min( b, min( c, d ) ) );**

**return ( a + b + c + d - Max - Min ) / 2.0;**

**}**

**// Utility function to find median of single array**

**float medianSingle(int arr[], int n)**

**{**

**if (n == 0)**

**return -1;**

**if (n%2 == 0)**

**return (double)(arr[n/2] + arr[n/2-1])/2;**

**return arr[n/2];**

**}**

**// This function assumes that N is smaller than or equal to M**

**// This function returns -1 if both arrays are empty**

**float findMedianUtil( int A[], int N, int B[], int M )**

**{**

**// If smaller array is empty, return median from second array**

**if (N == 0)**

**return medianSingle(B, M);**

**// If the smaller array has only one element**

**if (N == 1)**

**{**

**// Case 1: If the larger array also has one element,**

**// simply call MO2()**

**if (M == 1)**

**return MO2(A[0], B[0]);**

**// Case 2: If the larger array has odd number of elements,**

**// then consider the middle 3 elements of larger array and**

**// the only element of smaller array. Take few examples**

**// like following**

**// A = {9}, B[] = {5, 8, 10, 20, 30} and**

**// A[] = {1}, B[] = {5, 8, 10, 20, 30}**

**if (M & 1)**

**return MO2( B[M/2], MO3(A[0], B[M/2 - 1], B[M/2 + 1]) );**

**// Case 3: If the larger array has even number of element,**

**// then median will be one of the following 3 elements**

**// ... The middle two elements of larger array**

**// ... The only element of smaller array**

**return MO3( B[M/2], B[M/2 - 1], A[0] );**

**}**

**// If the smaller array has two elements**

**else if (N == 2)**

**{**

**// Case 4: If the larger array also has two elements,**

**// simply call MO4()**

**if (M == 2)**

**return MO4(A[0], A[1], B[0], B[1]);**

**// Case 5: If the larger array has odd number of elements,**

**// then median will be one of the following 3 elements**

**// 1. Middle element of larger array**

**// 2. Max of first element of smaller array and element**

**// just before the middle in bigger array**

**// 3. Min of second element of smaller array and element**

**// just after the middle in bigger array**

**if (M & 1)**

**return MO3 ( B[M/2],**

**max(A[0], B[M/2 - 1]),**

**min(A[1], B[M/2 + 1])**

**);**

**// Case 6: If the larger array has even number of elements,**

**// then median will be one of the following 4 elements**

**// 1) & 2) The middle two elements of larger array**

**// 3) Max of first element of smaller array and element**

**// just before the first middle element in bigger array**

**// 4. Min of second element of smaller array and element**

**// just after the second middle in bigger array**

**return MO4 ( B[M/2],**

**B[M/2 - 1],**

**max( A[0], B[M/2 - 2] ),**

**min( A[1], B[M/2 + 1] )**

**);**

**}**

**int idxA = ( N - 1 ) / 2;**

**int idxB = ( M - 1 ) / 2;**

**/\* if A[idxA] <= B[idxB], then median must exist in**

**A[idxA....] and B[....idxB] \*/**

**if (A[idxA] <= B[idxB] )**

**return findMedianUtil(A + idxA, N/2 + 1, B, M - idxA );**

**/\* if A[idxA] > B[idxB], then median must exist in**

**A[...idxA] and B[idxB....] \*/**

**return findMedianUtil(A, N/2 + 1, B + idxA, M - idxA );**

**}**

**// A wrapper function around findMedianUtil(). This function**

**// makes sure that smaller array is passed as first argument**

**// to findMedianUtil**

**float findMedian( int A[], int N, int B[], int M )**

**{**

**if (N > M)**

**return findMedianUtil( B, M, A, N );**

**return findMedianUtil( A, N, B, M );**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**int A[] = {900};**

**int B[] = {5, 8, 10, 20};**

**int N = sizeof(A) / sizeof(A[0]);**

**int M = sizeof(B) / sizeof(B[0]);**

**printf("%f", findMedian( A, N, B, M ) );**

**return 0;**

**}**