Searching & Sorting

Searching

1. Linear Search
2. Binary Search

Sorting

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Sort 012 (DNF Algo)
5. Merge Sort
6. Quick Sort

Linear Search

In this algorithm we simply traverse whole array and search for an element and return its index if its present, else we return -1.

Time complexity : O(N) *- worst case when target element is last element of the array.*
Space complexity : O(1) *- constant space*

Solve it here : [Click Here]()

[code] Linear Search

```cpp
/* ✔️ Linear Search Algorithm (Time : O(n) , Space : O(1)) */

#include <iostream>
using namespace std;

// -- this function will return index of target element if present, else if not present
then it will return -1
int linearSearch(int *arr, int size, int &elementToFind){

    for(int i = 0; i < size; i++)
        if(arr[i] == elementToFind) return i;

    return -1;
}

// MAIN FUNCTION
int main(){

    cout << "- - - - - - - - - - - - - - - - - - " << endl; // for better output

    int arr[] = {10, 20, -10, 21, 5, 3, 11};
    int size = sizeof(arr)/sizeof(int);

    /* Test Cases */
    int elementToFind = 21;    // TestCase-1 (output : 3)
    // int elementToFind = -10; //TestCase-2 (output : 2)
    //int elementToFind = -100; //TestCase-3 (output : 2)
```

```cpp
    // calling function linearSearch
    int ans = linearSearch(arr, size, elementToFind);

    if(ans != -1) cout << elementToFind <<" is present at index : " << ans << endl;
    else cout << elementToFind << " is not present in the array. " << endl;


    cout << "- - - - - - - - - - - - - - - - - - - " << endl; // for better output

}
```

```
Output:
- - - - - - - - - - - - - - - - - - -
21 is present at index : 3
- - - - - - - - - - - - - - - - - - -
```

Binary Search

#1 Approach (iterative)

In this algorithm we keep on dividing the array into 2 parts until we find the target element

In this algo we need a sorted array in the input.

Time complexity : O(log(N))
Space complexity : O(1)

Solve it here : Click Here

[code] Binary Search

```
/* ✔️Binary Search (Time : O(log(n)), Space : O(1) ) */

#include<iostream>
using namespace std;

// -- this function will return the index of the target element if present, else it will return -1 if element is not found in the given array
int binarySearch(int *arr, int size, int target){

    // step 1 : create 3 variables low, mid, high
    int low = 0, high = size - 1, mid;

    while(low <= high){
        // step 2 : find mid index from the current low and high
        mid = low + (high - low)/2;

        // step 3 : if the mid element is the target element then return the mid index
        if(arr[mid] == target) return mid;

        // step 4 : if the mid element is smaller than the target, then search in the right half of the array
```

```
        else if(arr[mid] < target) low = mid + 1;

        // step 5 : if the mid element is greater than the target, then search in the
left half of the array
        else high = mid - 1;

    }

    // step 6 : if loop completes then return -1, i.e no target element present in the
given array
    return -1;

}

// -- main function
int main(){


    cout << "- - - - - - - - - - - - - - - - - - - - - - - - - - " << endl;

    int arr[] = {10, 12, 15, 19, 21, 26, 28};
    int size = sizeof(arr)/sizeof(int);

    /* TEST CASES */

    //int target = 21;  // testCase - 1 (output : 4)
    // int target = 28;  // testCase - 2 (output : 6)
    //int target = 122221;  // testCase - 3 (output : -1)
    //int target = 10;  // testCase - 4 (output : 0)
    int target = 19;  // testCase - 5 (output : 3)


    int ans = binarySearch(arr, size, target);

    if(ans != -1) cout << target <<" is present at index " << ans << endl;
    else cout << target << " not present in the array." << endl;


    cout << "- - - - - - - - - - - - - - - - - - - - - - - - - - " << endl;

}
```

```
Output:
- - - - - - - - - - - - - - - - - - - - - - - - - -
19 is present at index 3
- - - - - - - - - - - - - - - - - - - - - - - - - -
```

Binary Search

#2 Approach (Recursive)

Time complexity : O(log(N))
Space complexity : O(N)

```
Approach explanation :-

step 1 : base case - if low becomes greater than high then that
                        means target element not found in the array.
step 2 : solving 1 case i.e finding mid for initial low and high and then returning
          index if it is the target element.
step 3.1 : else if the mid element is smaller than target then recursion will search
             for the target in the right part and return the ans.
step 3.2 : else if element is greater than the target element then recursion will
             search for the target in the left part and return the ans.
```

Solve it here : [Click Here]

[code] Binary Search

```
/* ✔️⭐[Approach 2 - Recursive] Binary Search  */

#include<iostream>
using namespace std;

// -- Recursive Binary Search Function

int recBinarySearch(int *arr, int low, int high, int target){

    // step 1 : base case - if low becomes greater than high then that means target
element not found in the array
    if(low > high) return -1;

    // step 2 : solving 1 case i.e finding mid for initial low and high and then
returning index if it is the target element
```

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

    if(arr[mid] == target) return mid;

    // step 3.1 : else if the mid element is smaller than target then recursion will
search for target in the right part and return the ans
    else if(arr[mid] < target) return recBinarySearch(arr, mid+1, high, target);

    // step 3.2 : else if element is greater than the target element then recursion
will search for target in the left part and return the ans
    else return recBinarySearch(arr, low, mid-1, target);

}

// -- main function
int main(){


    cout << "- - - - - - - - - - - - - - - - - - - - - - - - - - - " << endl;

    int arr[] = {10, 12, 15, 19, 21, 26, 28};
    int size = sizeof(arr)/sizeof(int);

    /* TEST CASES */

    // int target = 21;  // testCase - 1 (output : 4) ✅
    // int target = 28;  // testCase - 2 (output : 6)✅
    // int target = 122221;  // testCase - 3 (output : -1)✅
    // int target = 10;  // testCase - 4 (output : 0)✅
    int target = 19;  // testCase - 5 (output : 3)✅



    int low = 0, high = size-1;
    int ans = recBinarySearch(arr, low, high, target);

    if(ans != -1) cout << target <<" is present at index " << ans << endl;
    else cout << target << " not present in the array." << endl;


    cout << "- - - - - - - - - - - - - - - - - - - - - - - - - - - " << endl;

}
```

```
Output:
- - - - - - - - - - - - - - - - - - - - - - - - - - -
19 is present at index 3
```

Selection Sort

#1 Approach (iterative)

In this algorithm we keep on dividing the array into 2 parts until we find the target element

In this algo we need a sorted array in the input.

Time complexity : O(N^2)
Space complexity : O(1)

Solve it here : Click Here

```
APPROACH EXPLANATION :-

    step 1 : run a loop from 0 to n-2 th element of array

           step 2 : store the ith index in a variable named 'minIndex'
           step 3 : run a loop from (i+1 to n-1)th element

                  step 4 : compare the jth element with the element at 'minIndex' and   if jth
                           element is smaller than it then update the minIndex with jth index

                         step 5 : after the j the loop completes, swap the ith index element
                                  with the minIndex element.

   step 6 : after the i th loop also completes, the array is now sorted.

       T : O(n^2)
       S : O(1)
```

[code] Selection Sort (approach - 1)

```
/* ✔️ Selection Sort - T : O(n^2) S : O(1) */

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

```
// -- this function will print the array at any instance
void printArray(int *arr, int size){

    for(int i = 0; i < size; i++) cout << arr[i] << ' ';
    cout << endl;

}

// -- this function will sort the array
void selectionSort(int *arr, int size){

    // step 1 : run a loop from 0 to n-2 th element of array
    for(int i = 0; i <= size-2; i++){

        // step 2 : store the ith index in a variable named 'minIndex'
        int minIndex = i;

        // step 3 : run a loop from (i+1 to n-1)th element
        for(int j = i + 1; j <= size-1; j++){

            // step 4 : compare the jth element with the element at 'minIndex' and if
jth element is smaller than it then update the minIndex with jth index
            if(arr[j] < arr[minIndex]) minIndex = j;

        }

        // step 5 : after the j the loop completes, swap the ith index element with
the minIndex element
        swap(arr[i], arr[minIndex]);

    }

    // step 6 : after the i th loop also completes, the array is now sorted.

}

// -- Main Function
int main(){
    cout << "- - - - - - - - - - - - - - - - - - - - " << endl;

    int arr[] = {10, 21, 17, -5, 3, 2, 11};
    int size = sizeof(arr)/sizeof(int);

    cout << "Input Array : ";
    printArray(arr, size);

    // sorting the array
```

```cpp
    selectionSort(arr, size);

    cout << "Output Array : ";
    printArray(arr, size);

    cout << "- - - - - - - - - - - - - - - - - - - " << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - - - -
Input Array : 18 -1 -3 -10 100 81 95 28
Output Array : -10 -3 -1 18 28 81 95 100
- - - - - - - - - - - - - - - - - - - - - -
```

Selection Sort

#2 Approach (Recursive)

Time complexity : O(N^2)
Space complexity : O(N) - recursive call stack

```
APPROACH EXPLANATION :-

step 1 : base case - if the array has size 1 or 0, then it's already sorted.
step 2 : solving 1 case i.e filling the 0th index with the deserving element using swap().
step 3 : since we solved a case, rest recursion will sort, recursive call ->
         selectionSort(++arr, size-1).
```

[code] Selection Sort (approach -2)

```cpp
/*
✔️⭐ SELECTION SORT [Approach 2 - Recursive]
*/

#include<iostream>
using namespace std;

// -- this function will print the array at any instance
void printArray(int *arr, int size){

    for(int i = 0; i < size; i++) cout << arr[i] << ' ';
    cout << endl;

}

void selectionSort(int *arr, int size){

    // step 1 : base case - if array has size 1 or 0, then its already sorted
    if(size == 0 || size == 1) return;

    // step 2 : solving 1 case i.e filling the 0th index with the deserving element
    int minIndex = 0;
    for(int j = 1; j <= size-1; j++){
        if(arr[j] < arr[minIndex]) minIndex = j;
    }

    swap(arr[0], arr[minIndex]);
```

```
    // step 3 : since we solved a case, rest recursion will sort
    selectionSort(++arr, size-1);

}

int main(){

    cout << "- - - - - - - - - - - - - - - - - - - - " << endl;

    /* TEST CASES */

    //int arr[] = {10, 21, 17, -5, 3, 2, 11}; // test case 1 ✅
      //int arr[] = {6, 2, 8 ,4 ,10};           // test case 2 ✅
      int arr[] = {18, -1, -3, -10, 100, 81, 95, 28}; // test case 3 ✅

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

    cout << "Input Array : ";
    printArray(arr, size);

    // sorting the array
    selectionSort(arr, size);

    cout << "Output Array : ";
    printArray(arr, size);

    cout << "- - - - - - - - - - - - - - - - - - - - " << endl;
}
```

```
- - - - - - - - - - - - - - - - - - - - - - 
Input Array : 18 -1 -3 -10 100 81 95 28
Output Array : -10 -3 -1 18 28 81 95 100
- - - - - - - - - - - - - - - - - - - - - - 
```

Bubble Sort

#1 Approach (Iterative)

Time complexity : O(N^2) - worst case, O(N) - best case
Space complexity : O(1)

Solved it here : [Click Here]

```
Approach Explanation :

step 1 : run a loop from 1 to n-1 (n-1 because it will handle the worst cases like
         (10, 9, 8, 7, 6, 5).

    step 2 : create a boolean variable 'swapped', and initialise it with false
    step 3 : run a loop from (0 to N-round)th element

        step 4 : if the (j+1)th element is smaller than the jth element than swap, and
                 mark bool 'swapped' as true
    step 5 : as the jth loop completes, check if the bool 'swapped' is false, if so
             Than break the 'round' loop also because the array is now sorted

// ARRAY SORTED
```

[code] Bubble Sort (approach - 1)

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

// -- function to print the array at any instance
void printArray(int arr[], int n){
    for(int i=0; i<n; i++){
        cout << arr[i] << ' ';
    }
    cout << endl;
}

// -- in case swap stl doesnt work
void swapFun(int *arr, int i, int j){

    int temp = arr[i];
```

```
    arr[i] = arr[j];
    arr[j] = temp;

}

// -- bubble sort iterative function
void bubbleSort(int *arr, int size){

    // step 1 : run a loop from 1 to n-1 (n-1 because it will handle the worst
cases like 10, 9, 8, 7, 6, 5)
        for(int round = 1; round <= size - 1; round++){

        // step 2 : create a boolean variable 'swapped', and initiallize it with
false
        bool swapped = false;

        // step 3 : run a loop from (0 to N-round)th element
        for(int j = 0; j <= size-round-1; j++){

            // step 4 : if the (j+1)th element is smaller than the jth element
than swap, and mark bool 'swapped' as true
            if(arr[j+1] < arr[j]){
                swapFun(arr, j, j+1);
                swapped = true;
            }
        }

        // step 5 : as the jth loop completes, check if the bool 'swapped' is
false, if so than break the 'round' loop also because the arrray is now sorted
        if(swapped == false) break;
    }

    // ARRAY SORTED
}

// -- main function
int main(){

    cout << "- - - - - - - - - - - - - - - - - - " << endl;

    /* Test Cases */

    //int arr[] = {10, 21, 17, -5, 3, 2, 11};  // testCase - 1 ✅
    //int arr[] = {10,7,17,6,9,1,5};  // testCase - 2 ✅
    int arr[] = {1, 2, 3, 4, -10, -9, -8 };  // testCase - 3 ✅
    //int arr[] = {1,2,3,4,5}; // testCase - 4 ✅
```

```cpp
    int size=sizeof(arr)/sizeof(int);

    cout << "Original Array : " ;
    printArray(arr,size);

    bubbleSort(arr,size);

    cout << "Using bubble Sort : ";
    printArray(arr,size);

    cout << "- - - - - - - - - - - - - - - - - - - " << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - - 
Original Array : 1 2 3 4 -10 -9 -8
Using bubble Sort : -10 -9 -8 1 2 3 4
- - - - - - - - - - - - - - - - - - - - 
```

Bubble Sort

#2 Approach (Recursive)

Time complexity : O(N^2) - worst case, O(N) - best case
Space complexity : O(N) - recursive stack

```
 explanation :

Step 1 :base case - when array has single element, then its sorted i.e
                    round == 1 == size, also if array has size 0 then also its
                    already sorted.

step 2 : manage a bool variable swapped initially with value false, to optimise
the algorithm to O(N) time in the best case.

step 3 : solving single case i.e filling the last index with the element it
deserves

      step 4 : swap the pair and mark swapped as true if we swap a pair

step 5 : when the i loop breaks check if we swapped is false or not is so then
return the function as the array is sorted - for optimisation

step 6 : recursion will place rest elements at their right position they belong to

// ARRAY SORTED
```

Solved it here : Click Here

[code] Bubble Sort (approach - 2)

```
// -- function to print the array at any instance
void printArray(int arr[], int n){
    for(int i=0; i<n; i++){
        cout << arr[i] << ' ';
    }
    cout << endl;
}
```

```cpp
// use this function when the original swap stl method is not working
void swapFun(int *arr, int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// this is recursive function for bubble sort
void recBubbleSort(int *arr, int size, int rounds){

    // Step 1 :base case - when array has single element, then its sorted i.e round
== 1 == size, also if array has size 0 then also its already sorted
    if(size == 0 || rounds == size) return;

    // step 2 : manage a bool variable swapped initially with value false, to
optimise the algorithm to O(N) time in the best case
    bool swapped = false;

    // step 3 : solving single case i.e filling the last index with the element it
deserves
    for(int j = 0; j <= size - rounds - 1; j++){

        // step 4 : mark swapped as true if we swap a pair
        if(arr[j+1] < arr[j]){
            swapFun(arr, j, j+1);
            swapped = true;
        }

    }

    // step 5 : when the i loop breaks check if we swapped is false or not is so then
return the function as the array is sorted - for optimisation
    if(swapped == false) return;

    // step 6 : recursion will place rest elements at their right position they
belong to
    recBubbleSort(arr, size, ++rounds);

    // ARRAY SORTED
}


// -- main function
int main(){
```

```
  cout << "- - - - - - - - - - - - - - - - - - -" << endl;

  /* Test Cases */

  //int arr[] = {10, 21, 17, -5, 3, 2, 11};  // testCase - 1 ✅
  //int arr[] = {10,7,17,6,9,1,5};  // testCase - 2 ✅
    int arr[] = {1, 2, 3, 4, -10, -9, -8 };  // testCase - 3 ✅
    //int arr[] = {1,2,3,4,5}; // testCase - 4 ✅

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

    cout << "Original Array : " ;
    printArray(arr,size);

    int rounds = 1;
    recBubbleSort(arr,size, rounds);

    cout << "Using bubble Sort : ";
    printArray(arr,size);

    cout << "- - - - - - - - - - - - - - - - - - - " << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - - 
Original Array : 1 2 3 4 -10 -9 -8
Using bubble Sort : -10 -9 -8 1 2 3 4
- - - - - - - - - - - - - - - - - - - - 
```

Insertion Sort

#1 Approach (Iterative)

Time complexity :-

O(N^2) - worst case, 5 4 3 2 1

O(N) best case-eg. 1 2 3 4 5 (no shifting,coping done in this case)

Space complexity :- O(1)

```
Approach Explanation :-

step 1 : run an i loop from (1 to n-1)th element

     step 2 : store the ith index element in a variable "tempVar"

     step 3 : run a j loop from (i-1 to 0)th index

            step 4 : if any jth element is found which is greater than    the tempVar
                     element, then perform "arr[j+1] = arr[j]".

            step 5 : else if any jth element found which is smaller or equal to the
                     tempVar element then break the jth loop.

step 6 : when the jth loop is complete/breaks then perform "arr[j+1] = tempVar".

// when both the loop completes/breaks then the array is SORTED.
```

Solved it here : [*Click Here*]()

[code] Insertion Sort (approach - 1)

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

// -- function to print the array at any given instance
void printArray(int *arr, int size){

    for(int i = 0; i < size; i++) cout << arr[i] << ' ';
    cout << endl;

}
// -- selection sort iterative function
void insertionSort(int *arr, int size){

    // step 1 : run an i loop from (1 to n-1)th element
    for(int i = 1; i < size; i++){

        // step 2 : store the ith index element in a variable "tempVar"
        int tempVar = arr[i];

        // step 3 : run a j loop from (i-1 to 0)th index
        int j = i-1;   // to use it later
        for( ; j >= 0; j--){

            // step 4 : if any jth element is found which is greater than the
tempVar element, then perform "arr[j+1] = arr[j]"
            if(tempVar < arr[j]) arr[j+1] = arr[j];

            // step 5 : else if any jth element found which is smaller or equal to
the tempVar element then break the jth loop
            else break;
        }

        // step 6 : when the jth loop is complete/breaks than perform "arr[j+1] =
tempVar"
        arr[j+1] = tempVar;

    }

    // when both the loop completes/breaks then the array is SORTED.
}
```

```cpp
// -- MAIN Function
int main(){


 cout << "- - - - - - - - - - - - - - - - - - - -" << endl;

    //int arr[] = {10, 21, 17, -5, 3, 2, 11};  // testCase - 1 ✅
    int arr[] = {1, 7, -9, -10, 20};  // testCase - 2 ✅
    //int arr[] = {10, 11, 12, 13, 14};  // testCase - 3 ✅

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

    cout << "Input Array : ";
    printArray(arr, size);

    selectionSort(arr, size);

    cout << "Output Array : ";
    printArray(arr, size);

    cout << "- - - - - - - - - - - - - - - - - - - -" << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - -
Input Array : 1 7 -9 -10 20
Output Array : -10 -9 1 7 20
- - - - - - - - - - - - - - - - - - - -
```

Insertion Sort

#2 Approach (Recursive)

Time complexity :-

O(N^2) - worst case, 5 4 3 2 1

O(N) best case-eg. 1 2 3 4 5 (no shifting,coping done in this case)

Space complexity :- O(N) - recursive call stack

```
Approach Explanation :-

step 1 : base case : when i equals n i.e the arrays last element is already
         processed, then array is sorted

step 2 : lets solve single case i.e lets place ith element at its right position

step 3 : create a tempVar having value of element at the ith index

step 4 : run a loop from i-1 to 0th index in reverse order, and whenever find an
         element which is greater than 'tempVar' then perform "arr[j+1] = arr[j]",
         else break the loop.

step 5 : whenever the loop breaks, simply perform "arr[j+1] = tempVar".

step 6 : rest elements recursion will place at their correct position, and sort the
array

//Array Sorted
```

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


// -- function to print the array at any given instance
void printArray(int *arr, int size){

    for(int i = 0; i < size; i++) cout << arr[i] << ' ';
    cout << endl;

}
```

```
// -- selection sort recursive function
void recInsertion(int *arr, int size, int i){

    // step 1 : base case : when i equals n i.e the arrays last element is already processed, then array is sorted
    if(i == size) return;

    // step 2 : lets solve single case i.e lets place ith element at its right position

    // step 3 : create a tempVar having value of element at the ith index
    int tempVar = arr[i];

    // step 4 : run a loop from i-1 to 0th index in reverse order, and whenever find an element which is greater than 'tempVar' then perform "arr[j+1] = arr[j]", else break the loop
    int j = i - 1;
    for( ; j >= 0; j--){
        if(tempVar < arr[j]) arr[j+1] = arr[j];
        else break;
    }

    // step 5 : whenever the loop breaks, simply perform "arr[j+1] = tempVar".
    arr[j+1] = tempVar;

    // step 6 : rest elements recursion will place at their correct position, and sort the array
    recInsertionSort(arr, size, ++i);

}

// -- MAIN Function
int main(){

    cout << "- - - - - - - - - - - - - - - - - - - -" << endl;

    int arr[] = {10, 21, 17, -5, 3, 2, 11};  // testCase - 1 ☑️
    //int arr[] = {1, 7, -9, -10, 20};  // testCase - 2 ☑️
    //int arr[] = {10, 11, 12, 13, 14};  // testCase - 3 ☑️

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

    cout << "Input Array : ";
    printArray(arr, size);
```

```cpp
    int i = 1;
    recSelectionSort(arr, size, i);

    cout << "Output Array : ";
    printArray(arr, size);

    cout << "- - - - - - - - - - - - - - - - - - - -" << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - -
Input Array : 10 21 17 -5 3 2 11
Output Array : -5 2 3 10 11 17 21
- - - - - - - - - - - - - - - - - - - -
```

Merge Sort

#1 Approach (Recursive)

*Time complexity : O(n*log(n)) - best & worst case*
Space complexity : O(N) - recursive call stack
- *Based on divide and conquer*

```
Approach explanation :-

   1. DIVIDE Function

        step 1 : base case - if start becomes equal to end i.e when
                 array size is 1 then its already sorted, just return
                 the function.
        step 2 : find the mid element for the given start and end at an
                 instance.
        step 3 : sort the left part of the array
        step 4 : sort the right part of the array
     <--step 5 : now since both the parts are sorted, just merge the
     |            2 arrays.
     |  //array sorted.
     |
     |
     --> CONQUER Function

     step 1 : create an array 'left' of length1 (mid-start+1)
     step 2 : create an array 'right' of length2 (end - mid)
     step 3 : now push the first half elements of original array into
              array 'left'.
     step 4 : now push the second half elements of original array into
              array 'right'.
     step 5 : now using 2 pointer variables approach, push the 2 sorted
              arrays(left & right) into original array
     step 6 : when either one of left or right sorted array all
              elements are pushed and others are still left then push
              all elements of non-empty array into original array.
     // now the 2 sorted arrays are merged.
```

Solved it here : [Click Here]

[code] Merge Sort (approach - 1)

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

void printArray(int *arr, int size){
    for(int i=0; i<size; i++) cout << arr[i] << ' ';
    cout << endl;
}

// -- function to merge the 2 sorted arrays.
void merge(int *arr, int start, int mid, int end){

    // step 1 : create an array 'left' of length1 (mid-start+1)
    int length1 = mid - start + 1;
    int *left = new int[length1];

    // step 2 : create an array 'right' of length2 (end - mid)
    int length2 = end - mid;
    int *right = new int[length2];

    // step 3 : now push the first half elements of the original array into array
'left'.
    int originalIndex = start; // to not lose the indexes of original array
    for(int i = 0; i < length1; i++) left[i] = arr[originalIndex++];

    // step 4 : now push the second half elements of the original array into array
'right'.
    for(int i = 0; i < length2; i++) right[i] = arr[originalIndex++];

    // step 5 : now using 2 pointer variables approach, push the 2 sorted
arrays(left & right) into original array
    int i = 0, j = 0;
    originalIndex = start;
    while(i < length1 && j < length2){
        if(left[i] < right[j]) arr[originalIndex++] = left[i++];
        else arr[originalIndex++] = right[j++];
    }

    // step 6 : when either one of left or right sorted array's all elements are
pushed and others are still left then push all elements of non empty array into
original array
    while(i < length1) arr[originalIndex++] = left[i++];
```

```
   while(j < length2) arr[originalIndex++] = right[j++];

    // now the 2 sorted arrays are merged.
}

// -- this function will divide the array into parts until a single element is left
void divide(int *arr, int start, int end){

    // step 1 : base case - if start becomes equal to end i.e when array size is 1
then its already sorted, just return the function
    if(start >= end) return;

    // step 2 : find the mid element for the given start and end at an instance
    int mid = start + (end - start)/2;

    // step 3 : sort the left part of the array
    divide(arr, start, mid);
    // step 4 : sort the right part of the array
    divide(arr, mid + 1, end);

    // step 5 : now since both the parts are sorted, just merge the 2 arrays
    merge(arr, start, mid, end);

    // array sorted.
}

int main(){

    cout << "- - - - - - - - - - - - - - - - - - - - - - - -" << endl;

    // int arr[] = {10, -1, -5, 6, 12, 3}; // TestCase 1 ☑️
    // int arr[] = {1, 2, 3, -1, 4, 5, 8, -5}; // TestCase 2 ☑️
    int arr[] = {1, 2, 3, 4, -4, -3, -2, -1}; // TestCase 3 ☑️
    //int arr[] = {1, 2, 3, 4};  // TestCase 4 ☑️
    int size = sizeof(arr)/sizeof(int);

    int start = 0, end = size - 1;

    cout <<"Input Array : ";
    printArray(arr,size);

    // calling the mergeSort function divide
    divide(arr, start, end);

    cout <<"Output Array : ";
```

```
    printArray(arr,size);
    cout << "- - - - - - - - - - - - - - - - - - - - - - - -" << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - - - - - -
Input Array : 1 2 3 4 -4 -3 -2 -1
Output Array : -4 -3 -2 -1 1 2 3 4
- - - - - - - - - - - - - - - - - - - - - - - -
```

Quick Sort

#1 Approach (Recursive)

*Time complexity : O(n^2) - worst case, O(n*log(n)) - best case*
Space complexity : O(n) - recursive call stack in worst case and o(log(n)) in normal case

- *Based on divide and conquer*

```
Approach explanation :-

                Quick Sort Function

                step 1 : base case - when there is single element left than array is
                         already sorted
           |--- step 2 : find the pivot index using function 'pivotIndexFinder'
           |    step 3 : now do recursive call for left part of array
           |    step 4 : do recursive call for right part of the array
           |
           |-->   PIVOT INDEX Finder

                step 1 : take starting index as the pivot of the array
                step 2 : count number of elements smaller or equal to the pivot
                         element.
                step 3 : swap the pivotElement and (pivotElement + count)th element.
                step 4 : declare 2 variables i = start and j = end, and run a loop
                         until i becomes equal to pivotIndex or j becomes equal to
                         pivotIndex.
                step 5 : run a loop until ith element is equal or smaller than the
                         pivotIndex element.
                step 6 : run a loop until jth element is greater than the pivotIndex
                         element.
                step 7 : swap the ith and jth element if index i is smaller than the
                         pivotIndex and index j is greater than the pivot index.
                step 8 : return pivot Index.
```

Solved it here : [Click Here]

[code] Quick Sort (approach - 1)

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

// -- function to print the array at any instance
void printArray(int *arr, int size){
    for(int i = 0; i < size; i++) cout << arr[i] << ' ';
    cout << endl;
}

// -- pivotIndexFinder Function -> this fuction will return the pivot element
int pivotIndexFinder(int *arr, int start, int end){

    // step 1 : take starting index as the pivot of the array
    int pivotIndex = start;

    // step 2 : count number of elements smaller or equal to the pivot element
    int count = 0;
    for(int i = start+1; i < end +1; i++){
        if(arr[i] <= arr[pivotIndex]) count++;
    }

    // step 3 : swap the pivotElement and (pivotElement + count)th element.
    swap(arr[pivotIndex], arr[pivotIndex + count]);
    pivotIndex = start + count;

    // step 4 : declare 2 variables i = start and j = end, and run a loop until i
becomes equal to pivotIndex or j becomes equal to pivotIndex.
    int i = start, j = end;
    while(i < pivotIndex && j > pivotIndex){

        // step 5 : run a loop until ith element is equal or smaller than the
pivotIndex element
        while(arr[i] <= arr[pivotIndex]) i++;
        // step 6 : run a loop until jth element is greater than the pivotIndex
element
        while(arr[j] > arr[pivotIndex]) j--;

        // step 7 : swap the ith and jth element if index i is smaller than the
pivotIndex and index j is greater than the pivot index.
```

```
        if(i < pivotIndex && j > pivotIndex) swap(arr[i++], arr[j--]);

    }

    // step 8 : return pivot Index.
    return pivotIndex;

}

// -- quick Sort function
void quickSort(int *arr, int start, int end){

    // step 1 : base case - when there is single elemnt left than array is already
sorted
    if(start >= end) return;


    // step 2 : find the pivot index using function 'pivotIndexFinder'
    int pivotIndex = pivotIndexFinder(arr, start, end);

    // step 3 : now do recursive call for left part of array
    quickSort(arr, start, pivotIndex-1);

    // step 4 : do recursive call for right part of the array
    quickSort(arr, pivotIndex+1, end);

}

int main(){

    cout << "- - - - - - - - - - - - - - - - - - - - - -" << endl;

    //int arr[] = {10, 12, -3, -5, 6, -1};  // test case 1 ✅
    //int arr[] = {1, 2, 3, -3, -2, -1};  // test case 2 ✅
    int arr[] = {1, 2, 3, 4, 5, 6, 7, -2, -3, 4, 10, 4};  // test case 3 ✅

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

    cout << "Input Array : ";
    printArray(arr, size);

    int start = 0, end = size - 1;
    quickSort(arr, start, end);

    cout << "Output Array : ";
    printArray(arr, size);
```

```
    cout << "- - - - - - - - - - - - - - - - - - - - - -" << endl;

}
```

```
- - - - - - - - - - - - - - - - - - - - - -
Input Array : 1 2 3 4 5 6 7 -2 -3 4 10 4
Output Array : -3 -2 1 2 3 4 4 4 5 6 7 10
- - - - - - - - - - - - - - - - - - - - - -
```