# STL Functions

While working on a problem that requires some basic tasks like sorting, searching, etc we are not required to make a separate functionality to achieve them. Rather, these functionalities are already present in C++ STL.

## 1. sort()

The sort function in STL is given a start pointer and an end pointer and it sorts any array in **O(log(N))** time complexity.
It can be used in any file using **#include<algorithm>.**

**For example, arr = [ 1, 3, 2, 5, 7, 6 ]**

| | |
|---|---|
| **sort( arr, arr+6 );** | **Sorts the array in ascending order** |
| **sort( arr, arr+6, greater<int>() );** | **Sorts the array in descending order** |

```
int main(){

    int arr[] = {1, 3, 2, 5, 7, 6};
    sort(arr, arr+6);
    for(int i=0; i<6; i++){
        cout << arr[i] << " ";
    }

    return 0;
}
```

We can also sort the array of objects on the basis of one of the properties of a class or structure to whom the object belongs. For that we are required to pass a **compare function** along with other parameters and have to define this compare function.

```
#include<iostream>
#include<algorithm>

using namespace std;

// creating a structure Interval
// with a start time and end time
struct Interval{
    int st;
    int et;
};
// compare function to pass in the sort function
// to sort on the basis of start time
bool compare(Interval i1,Interval i2){
    return i1.st < i2.st; // > for descending
}

int main(){

    Interval arr[] = {{6, 4}, {3, 4}, {4, 6}, {8, 13}};
    sort(arr, arr+4, compare);

    for(int i=0; i<4; i++){
        cout << arr[i].st << " : " << arr[i].et << endl;
    }

    return 0;
}
```

## 2. binary_search

Binary search can be applied only on a sorted array and its time complexity is **O(log(N)).**

Syntax : **binary_search(arr, arr+n, element to be searched);**

## 3. lower_bound

lower_bound() returns an iterator pointing to the first element in the range [first, last). The function returns the index of the next smallest number just greater than or equal to that number we are searching for. If there are multiple values that are equal to the element to be found, lower_bound() returns the index of the first such value.

Syntax : **lower_bound(arr, arr+n, element);**

Examples:

1. **Input: 10 20 30 40 50**
   **Output: lower_bound for element 30 at index 2**

2. **Input: 10 20 30 40 50**
   **Output: lower_bound for element 35 at index 3**

Lower_bound & upper_bound can be applied only on a sorted array and its time complexity is **O(log(N)).**

## 4. upper_bound

upper_bound() returns an iterator pointing to the first element in the range [first, last) that is greater than the element that needs to be found, or last if no such element is found. The elements in the range shall already be sorted or at least partitioned with respect to the element.

Syntax : **lower_bound(arr, arr+n, element);**
Examples :

1. **Input : 10 20 30 30 40 50**
   **Output : upper_bound for element 30 is at index 4**

2. **Input : 10 20 30 40 50**
   **Output : upper_bound for element 45 is at index 4**

Lower_bound & upper_bound can be applied only on a sorted array and its time complexity is **O(log(N)).**

```
int main(){

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

    sort(arr,arr+6);

    for(int i=0;i<6;i++){
        cout<<arr[i] << " ";
    }
    cout<<endl;
    cout << binary_search(arr,arr+6,2) << endl;

    cout<<lower_bound(arr,arr+6,3) - arr << endl;
    cout<<upper_bound(arr,arr+6,3) - arr << endl;

    return 0;
}
```

# STL Examples

## 1. Remove Duplicates from Array

Print the array of unique elements. (you can change the order of occurrence of the elements.)

**Input Format :**

Line 1: Contains the size of array

Line 2: M integers which are elements of the array separated by space

**Output :**

Resultant array

**Sample Input :**

**7**

**2 4 3 3 5 3 4**

**Sample Output :**

**2 3 4 5**

**Explanation :**

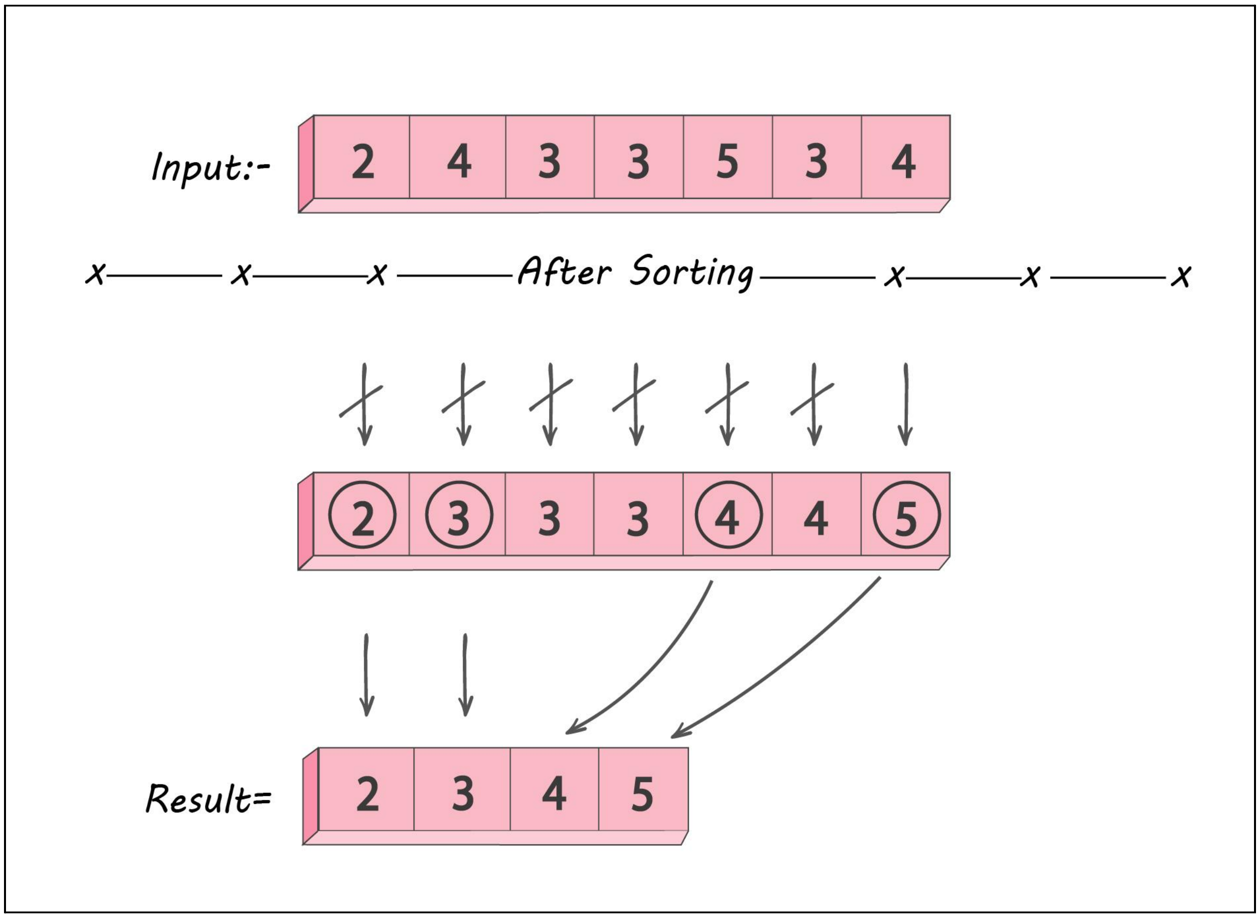
**Approach 1 :** This problem can be easily solved using a Set. We will first make a set using the inbuilt STL and a **result** array. Then we will traverse through our input array and start adding the elements in the set. While traversing the input array if we encounter any duplicates, that is if the element has already been added in the set , then we will not add the element to the result array otherwise we will.

```
set<int> removeDuplicates(vector<int> input){
    set<int> result;
    for(int i=1; i<input.size(); i++){
       result.insert(input[i]);
    }
    return result;
}
```

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

**Approach 2 :** Sort the elements in the input array so that all the duplicate elements will be together. Then traverse through the array and compare elements with the element present at the previous index; if they are equal do not add them in the result array otherwise add them.

**For example :**



```
vector<int> removeDuplicates(vector<int> input){
    vector<int> result;
    sort(input.begin(), inpt.end());

    result.push_back(input[0]);

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

        if(input[i] != input[i-1]){
            result.push_back(input[i]);
        }
    }
    return result;
}
```

**Time Complexity : O(N * log (N))**

**Space Complexity: O(N)**

## 2. First Non Repeating Character in a String

If there are multiple non repeating characters, print the unique character which comes first in the string. Or if all the characters are repeating, print the first character of the initial string.

**Input Format :**

Line 1: A String

**Output :**

First non repeating character

**Sample Input :**

**aDcadhc**

**Sample Output :**

**D**

**Explanation :** This problem can be solved using a **map** because with a map we can store the elements along with their frequencies.

```
char nonRepeatingCharacter(string str){

    map<char, int> frequency;
    for(int i=0; i<str.length(); i++){
        char currentChar = str[i];
        frequency[currentChar]++;
    }

    for(int i=0; i<str.length(); i++){
        if(frequency[str[i]] == 1){
            return str[i];
        }
    }
    // if there is no unique character in the string
    return str[0];
}
```

**Time Complexity: O(N)**