Important STL Algorithms

sort()

Used to sort some arrays/vectors in ascending, descending or your own custom order.

- It generally takes two parameters, the first one being the point of the array/vector from where the sorting needs to begin and the second parameter being the length up to which we want the array/vector to get sorted.

```
// C++ program to demonstrate default behaviour of sort() in STL.
#include <bits/stdc++.h>
using namespace std;
int main() {
    int arr[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
    int n = sizeof(arr) / sizeof(arr[0]);
    // Here we take two parameters, the beginning of the array and the length
    sort(arr, arr + n);
    cout << "\nArray after sorting using "
            "default sort is : \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    return 0;
}
```

- We can also sort using the iterators:

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

- We can also select some specific range in the given array/vector to sort:

```
sort(arr+2, arr+5);
```

- To sort in a descending order we can use the greater comparator as the third argument:

```
sort(arr.begin(), arr.end(), greater<int>);
```

- If we want to sort according to our own custom way, we can create a comparator function and pass it as the third argument to the sort function:

```
// Sort it according to the second element
// If second element is the same, then sort according to descending order of

bool comp(pair<int, int> p1, pair<int, int> p2){
    if(p1.second < p2.second) return true;
    if(p1.second > p2.second) return false;
    if(p1.first > p2.first) return true;
    return false;
}

pair<int, int> arr[] = {{1, 2}, {2, 1}, {4, 1}};

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

__builtin_popcount()

It is a feature of the GCC compiler. This function is used to count the number of set bits in an unsigned integer. In other words, it counts the number of 1’s in the binary form of a positive integer.

The syntax is:

```
__builtin_popcount(int number);
```

```
// C++ code to demonstrate the __builtin_popcount function
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n = 4;
    // Printing the number of set bits in n
    cout << __builtin_popcount(n); // Output is 1 since 4 in binary is 100
    return 0;
}
```

If the number is a long long int , you can use the __builtin_popcountll() function.

next_permutation()

It is used to rearrange the elements in the range [first, last) into the next lexicographically greater permutation.

To use next_permutation(), you have to include the 'algorithms' header file.

```
#include <algorithms>
```

```
// C++ program to illustrate next_permutation example
#include <algorithm>
#include <iostream>
using namespace std;
int main() {
    int arr[] = { 1, 2, 3 };
    sort(arr, arr + 3); // We have used sort here to first sort them otherwis
    cout << "The 3! possible permutations with 3 elements:\n";
    do {
        cout << arr[0] << " " << arr[1] << " " << arr[2] << "\n";
    } while (next_permutation(arr, arr + 3));

    cout << "After loop: " << arr[0] << ' '
        << arr[1] << ' ' << arr[2] << '\n';

    return 0;
}
```

The output of the above code is:

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
After loop: 1 2 3
```

Similarly, we can use prev_permutation() to get the previous permutations.

max_element()

We have *std::max* to find maximum of 2 or more elements, but what if we want to find the largest element in an array or vector or list or in a sub-section. To serve this purpose, we have *std::max_element* in C++.
It returns an **iterator** pointing to the element with the **largest** value in the range [first, last).

To use max_element(), you have to include the 'algorithm' header file

```
#include <algorithm>
```

```
// C++ program to demonstrate the use of std::max_element
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
int v[] = { 5, 3, 10, 9, 2, 3 };
// Finding the maximum value between the first and the fourth element
int maxi = max_element(v, v + 4);
cout << *(maxi) << "\n"; // Output - 10
return 0;
}
```

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

We can also use it with a **comparator** function.

Similarly to find the **minimum** element, we can use **min_element()**.