sort() in C++ STL

The **sort()** function is a built-in function in the C++ STL. This function is used to sort elements of containers that allow random access, like vector, arrays, deque, etc.

Header File: The sort() function is declared in the *"algorithm"* header file.

Syntax: The sort() function can be called in two ways, by passing either address of the first element and element just after the last element or by passing iterators pointing to the first element and element just after the last element.

```
Syntax 1: sort(arr, arr + n);
This syntax takes the address of first element in the array
and address just after the last element and sorts the
array in increasing order.

Syntax 2: sort(c.begin(), c.end());
This syntax takes the address of first element of the container
and address just after the last element and sorts the
container in increasing order.
```

Note: By default, the sort function sorts a container in increasing order.

How to sort in descending order?

The sort() function takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass “greater()” function to sort in descending order. This function does a comparison in a way that puts greater elements before.

Program 1: Example to sort elements of an Array.

```cpp

// C++ program to illustarte sort() in STL
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    int arr[] = {1, 5, 8, 9, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Sorts array in increasing order
    // by default
    sort(arr, arr+n);

    cout << "Array after sorting using "<<
        "default sort is : \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    // Sorts array in decreasing order
    sort(arr, arr + n, greater<int>());

    cout<<"\n\nArray after sorting in "<<
        "decreasing order:\n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

Run

Output:

```
Array after sorting using default sort is :
1 5 6 7 8 9

Array after sorting in decreasing order:
9 8 7 6 5 1
```

Program 2: Example to sort elements of a Vector.

```cpp

// C++ program to illustarte sort() in STL
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> vec = {5, 7, 20, 10};

    // Sorts vector in increasing order
    // by default
    sort(vec.begin(), vec.end());

    cout << "Vector after sorting using "<<
        "default sort is : \n";
    for (int x:vec)
        cout << x << " ";

    // Sorts vector in decreasing order
    sort(vec.begin(), vec.end(), greater<int>());

    cout<<"\n\nVector after sorting in "<<
        "decreasing order:\n";
    for (int x:vec)
        cout << x << " ";

    return 0;
}
```

Run

Output:

```
Vector after sorting using default sort is :
5 7 10 20

Vector after sorting in decreasing order:
20 10 7 5
```

How to sort in particular order?

We can also write our own comparator function and pass it as a third parameter. This “comparator” function returns a value; convertible to bool, which basically tells us whether the passed “first” argument should be placed before the passed “second” argument or not.

For Example: In the code below, suppose points in 2D space {3, 10} and {2, 8} are passed as arguments in the "myCmp" function(comparator function). Now as p1.x (=3) > p2.x (=2), so our function returns "false", which tells us that "first" argument should not be placed before "second" argument and so sorting will be done in order like {2, 8} first and then {3, 10} as next.

```cpp

// A C++ program to demonstrate STL sort() using
// our own comparator
#include<bits/stdc++.h>
using namespace std;

// A point in 2D space has x and y coordinates
struct Point
{
    int x, y;
};

// Compares two points according to x coordinates
bool myCmp(Point p1, Point p2)
{
    return (p1.x < p2.x);
}

int main()
{
```

```
21      Point arr[] = { {3,10}, {2, 8}, {5, 4} };
22      int n = sizeof(arr)/ sizeof(arr[0]);
23
24      // sort the points in increasing order of
25      // x coordinates
26      sort(arr, arr+n, myCmp);
27
28      for(auto i : arr)
29          cout<<i.x<<" "<<i.y<<endl;
30
```

Run

Output:

```
2 8
3 10
5 4
```

Time Complexity and Internal Working

Worst case and Average Case Time Complexity: The worst and average case time complexity of sort() function in O(N*logN), where N is the number of elements in the container.

Internal Working: This function internally uses IntroSort. In more details it is implemented using hybrid of QuickSort, HeapSort and InsertionSort.

By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort.

- Sample Problem - sort(): Fractional Knapsack

Problem: Maximize The Number Of Toys That Can Be Purchased With Amount K.

Given an array consisting of the cost of toys. Given an integer K depicting the amount of money available to purchase toys. Write a program to find the maximum number of toys one can buy with the amount K.

Note: One can buy only 1 quantity of a particular toy.

Example:

```
Input:  N = 10, K =  50
        cost = { 1, 12, 5, 111, 200, 1000, 10, 9, 12, 15 }
Output: 6
Explanation: Toys with amount 1, 5, 9, 10, 12, and 12
can be purchased resulting in a total amount of 49. Hence,
the maximum number of toys is 6.

Input: N = 7, K = 50
       cost = { 1, 12, 5, 111, 200, 1000, 10 }
Output: 4
```

The idea to solve this problem is to first sort the cost array in ascending order. This will arrange the toys in increasing order of the cost. Now iterate over the cost array and keep calculating the sum of costs until the sum is less than or equal to K. Finally return the number of toys used to calculate the sum which is just less than or equals to K.

Below image is an illustration of the above approach:

Initially : arr[] = { 100, 20, 50, 2, 10 }, k = 35
Sum = 0, Count = 0

Step 1: Sort the array
arr[] = { 2, 10, 20, 50, 100 }

Step 2: arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] <= k. Increment count and sum
Count = 1 , Sum = 2

Step 3: arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] <= k. Increment count and sum
Count = 2 , Sum = 12

Step 4: arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] <= k. Increment count and sum
Count = 3 , Sum = 32

Step 5: arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] > k. No other toys can added
Count = 3

Below is the implementation of the above approach:

```cpp

// C++ Program to maximize the
// number of toys with K amount
#include <bits/stdc++.h>
using namespace std;

// This functions returns the required
// number of toys
int maximum_toys(int cost[], int N, int K)
{
    int count = 0, sum = 0;

    // sort the cost array
    sort(cost, cost + N);
    for (int i = 0; i < N; i++) {

        // Check if we can buy ith toy or not
        if (sum + cost[i] <= K) {
            sum = sum + cost[i];
            // Increment count
            count++;
        }
    }
    return count;
}

// Driver Code
int main()
{
```

Run

Output:

```
6
```

Time Complexity: O(N * logN), where N is the size of cost array.

Sample Problem - sort(): Chocolate Distribution Problem

Problem: Chocolate Distribution Problem.

Given an array of n integers where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are m students, the task is to distribute chocolate packets such that:

1. Each student gets one packet.
2. The difference between the number of chocolates in the packet with maximum chocolates and packet with minimum chocolates given to the students is minimum.

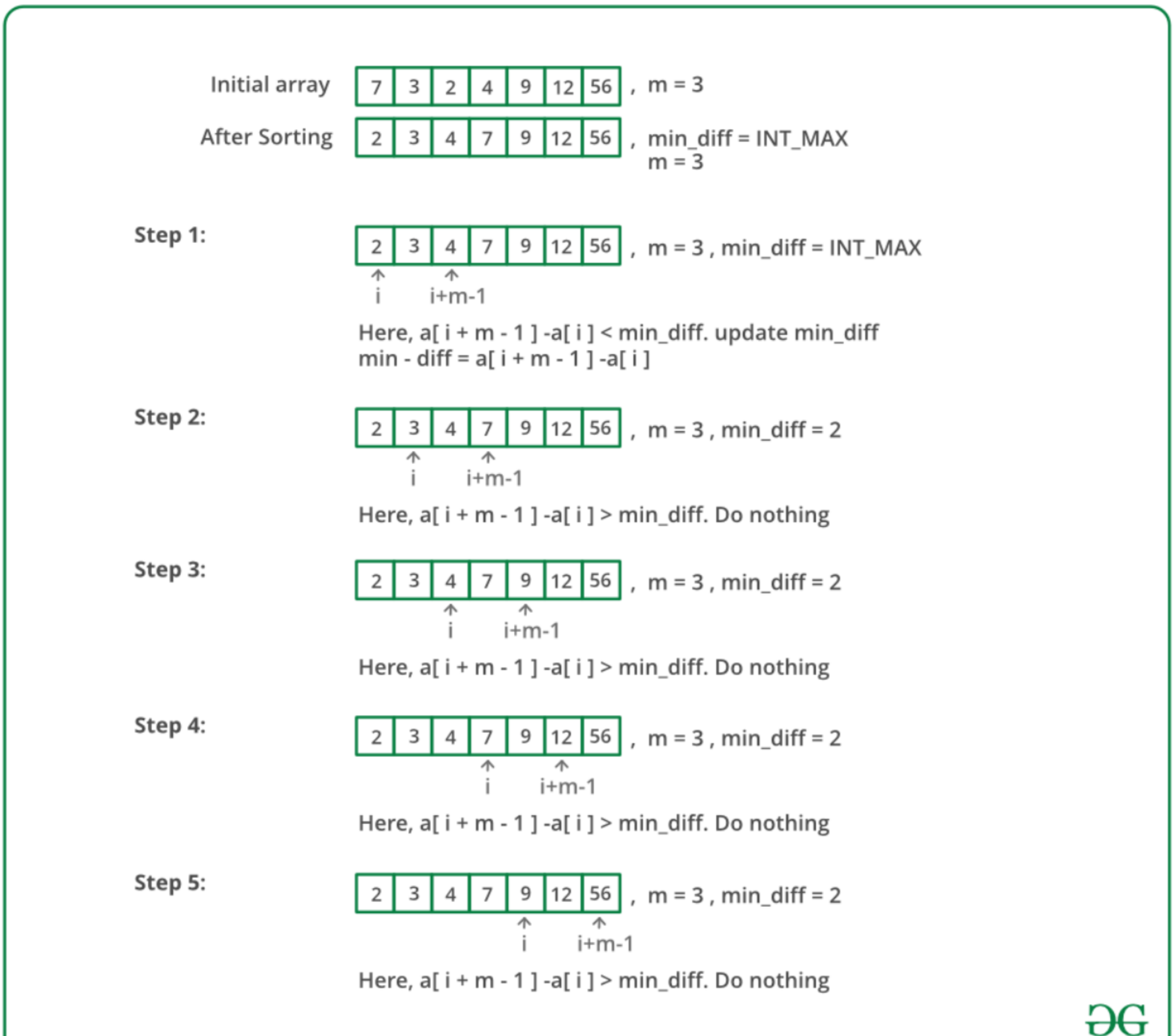
Example:

```
Input : arr[] = {7, 3, 2, 4, 9, 12, 56}
m = 3
Output: Minimum Difference is 2
Explanation: We have seven packets of chocolates and
we need to pick three packets for 3 students
If we pick 2, 3 and 4, we get the minimum
difference between the maximum and minimum packet
sizes.
```

A simple solution is to generate all subsets of size m of arr[0..n-1]. For every subset, find the difference between the maximum and minimum elements in it. Finally, return the minimum difference.

An efficient solution is based on the observation that to minimize the difference, we must choose consecutive elements from a sorted packet. We first sort the array arr[0..n-1], then find the subarray of size m with the minimum difference between last and first elements.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

```cpp

// C++ program to solve chocolate distribution
// problem
#include <bits/stdc++.h>
using namespace std;

// arr[0..n-1] represents sizes of packets
// m is number of students.
// Returns minimum difference between maximum
// and minimum values of distribution.
int findMinDiff(int arr[], int n, int m)
{
    // if there are no chocolates or number
    // of students is 0
    if (m == 0 || n == 0)
        return 0;

    // Sort the given packets
    sort(arr, arr + n);

    // Number of students cannot be more than
    // number of packets
    if (n < m)
        return -1;

    // Largest number of chocolates
    int min_diff = INT_MAX;

    // Find the subarray of size m such that
```

Run

Output:

```
Minimum difference is 10
```

Time Complexity: O(n Log n) as we apply sorting before subarray search.

Sample Problem - sort(): Maximize The Number Of Items Purchased

Problem: Given an array consisting of the cost of items. Given an integer K depicting the amount of money available to purchase the items. To find the maximum number of items one can buy with the amount K.

Note: One can buy only 1 quantity of a particular toy.

Example:

```
Input:  N = 10, K =  50, cost = { 1, 12, 5, 111, 200, 1000, 10, 9, 12, 15 }
Output: 6
Explanation: Toys with amount 1, 5, 9, 10, 12, and 12 can be
purchased resulting in a total amount of 49. Hence, maximum number of toys is 6.
```

The idea to solve this problem is to first sort the cost array in ascending order. This will arrange the items in increasing order of the cost. Now, iterate over the cost array and keep calculating the sum of costs until the sum is less than or equal to K. Finally, return the number of items used to calculate the sum which is just less than or equals to K.

Below image is a dry run of the above approach:

Initially :

arr[] = { 100, 20, 50, 2, 10 }, k = 35
Sum = 0, Count = 0

Step 1: Sort the array
arr[] = { 2, 10, 20, 50, 100 }

Step 2:
arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] <= k. Increment count and sum
Count = 1 , Sum = 2

Step 3:
arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] <= k. Increment count and sum
Count = 2 , Sum = 12

Step 4:
arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] <= k. Increment count and sum
Count = 3 , Sum = 32

Step 5:
arr[] = { 2, 10, 20, 50, 100 }
i
Sum + arr[i] > k. No other toys can added
Count = 3

Below is the implementation of the above approach:

```cpp
// C++ program to maximize the number of items purchased
#include<bits/stdc++.h>
using namespace std;

// Function to find the maximum items
int maximum_item(int cost[], int n, int k)
{
    int curr_sum = 0;
    int count = 0;

    // Using STL sort
    sort(cost, cost + n);

    // Loop with the condition of the cost limit
    for(int i = 0; i < n; i++)
    {
        if(curr_sum + cost[i] <= k)
        {
            curr_sum += cost[i];
            count++;
        }
    }
    return count;
}

int main()
{

    int cost[] = {1, 12, 5, 111, 200, 1000, 10, 9, 12, 15};
```

Run

Output:

```
6
```

Time Complexity: O(N * logN) where N is the size of the cost array.

Sample Problem - sort(): Sorting Array Elements By Frequency

Problem: Given an array of integers, sort the array according to the frequency of elements. If frequencies of two elements are the same, print them in increasing order.

Example:

```
Input: arr[] = {2, 3, 2, 4, 5, 12, 2, 3, 3, 3, 12}
Output: 3 3 3 3 2 2 2 12 12 4 5
Explanation:
No.: Freq
2  : 3
3  : 4
4  : 1
5  : 1
12 : 2
```

We can solve this problem by using maps and pairs. Initially, we create a map such that map[element] = freq. Once we are done building the map, we create an array of pairs. A pair that stores elements and their corresponding frequency will be used for the purpose of sorting. We write a custom compare function that compares two pairs firstly on the basis of freq and if there is a tie on the basis of values.

Below is the implementation of the above approach:

```cpp

// C++ program to sort elements by frequency
#include <bits/stdc++.h>
using namespace std;

// Function to compare two pairs
bool compare(pair<int,int> &p1,
             pair<int, int> &p2)
{
    // If frequencies are same, compare the values
    if (p1.second == p2.second)
        return p1.first < p2.first;
    return p1.second > p2.second;
}

// Function to print elements
void printSorted(int arr[], int n)
{
    // Store items and their frequencies
    map<int, int> m;
    for (int i = 0; i < n; i++)
        m[arr[i]]++;

    // No. of distinct values are equal to the size of the map
    int s = m.size();

    // An array of pairs
    pair<int, int> p[s];

    // Fill (val, freq) pairs in an array of pairs
```

Run

Output:

```
Elements sorted by frequency are: 3 3 3 3 2 2 2 12 12 4 5
```

Time Complexity: This will be O(n Log n).

next_permutation() in C++ STL

The **next_permutation()** is a built-in function in C++ STL, which is used to rearrange the elements in the range [first, last) into lexicographical next permutation of a given sequence. It provides the lexicographically smallest sequence that is just greater than the given sequence. Let's consider the

sequence {1, 2, 3, 4, 5}. So all next smaller permutations that are lexicographically greater than the given sequence are:

```
1, 2, 3, 5, 4
1, 2, 4, 3, 5
1, 2, 4, 5, 3
1, 2, 5, 3, 4
1, 2, 5, 4, 3
1, 3, 2, 4, 5
...
...
...
```

Therefore for a sequence of size N, there will be N! permutations.

Header File: The next_permutation() function is declared in the *"algorithm"* header file.

Syntax:

```
bool next_permutation (BidirectionalIterator first,

                       BidirectionalIterator last);
```

Parameter: Both the **first** and **last** are Bidirectional iterators pointing to the initial and final positions of the sequence. The range used is [first, last), which contains all the elements between first and last, including the element pointed by first and the index pointed beyond the last element.

Return value: The function returns **true** if it could rearrange the object as a lexicographically greater permutation. Otherwise, the function returns **false** to indicate that no next permutation is possible.

Examples:

```
Input: {1, 2, 3, 4, 5}
Output: {1, 2, 3, 5, 4}

Input: {1, 2, 5, 4, 3}
Output: {1, 3, 2, 4, 5}

Input: {5, 4, 3, 2, 1}
Output: {5, 4, 3, 2, 1}
```

Program:

```
1
2  // C++ program illustrating
3  // next_permutation() function
4  #include <algorithm>
5  #include <iostream>
6  #include <vector>
7  using namespace std;
8
9  int main()
10 {
11
12     // The sample Vector
13     vector<int> v = { 1, 2, 5, 4, 3 };
14
15     // Performing next_permutation
16     // operation
17     next_permutation(v.begin(), v.end());
18
19     // Displaying the sequence
20     for (int x : v)
21         cout << x << " ";
22     return 0;
23 }
24
```

Run

Output:

```
1 3 2 4 5
```

Algorithm for next_permutation() function:

Let the initial sequence be:

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

Step 1: Traverse from right, find the first element that is not in decreasing order. Let this element be x.

```
For the given sequence {1, 2, 5, 4, 3},

x would be 2
```

Step 2: Find the smallest element on the right of x that is just greater than x. Let this element be y.

```
For the given sequence {1, 2, 5, 4, 3}

y = 3
```

Step 3: Swap x and y. This will give the lexicographically greater sequence.

```
After swap: {1, 3, 5, 4, 2}
```

Step 4: Now, to get the smallest greater sequence, just reverse the subsequence after new position of y or previous position of x.

```
After reversing: {1, 3, 2, 4, 5}
```

Time Complexity:

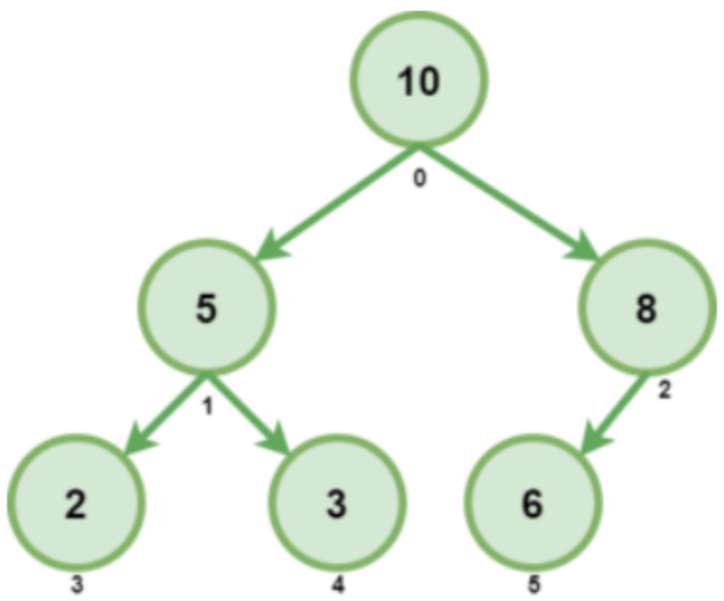
- Step 2 will take O(n) time.
- Step 3 will take O(n) time but it can be optimised using binary search into O(logn) time.
- Swapping in Step 3 is a constant operation.
- Reversing in step 4 will take O(n) time.

Therefore the overall time complexity of this operation is **O(n)** time.

make_heap() in C++ STL

The **make_heap()** is a built-in function in C++ STL which is used to transform a sequence or container into a heap. The heap can be a MAX_HEAP, a MIN_HEAP or in any order as decided by the user. A heap is a data structure which points to the highest or lowest element and makes the elements access in **O(1)** time. Order of all the other elements depends upon the particular implementation but remains consistent throughout.
Note: By default, the make_heap() function converts the container into a MAX_HEAP.

A **MAX_HEAP** is a heap, in which the root element is maximum among all the elements in the binary heap. There are few formulas using which one can obtain the location of the elements in a binary heap. Consider the below max heap binary tree and the formulas to find the elements in left, right or parent position.




```
left(i) = 2i + 1
Eg., left(1) = 2*1 + 1 = 1 (2)

right(i) = 2i + 2
```

```
Eg., right(1) = 2*1 + 2 = 4 (3)

parent(i) = floor((i - 1)/2)
parent(1) = floor((1 - 1) / 2) = 0 (10)
```

Header File: The make_heap() function is declared in the *"algorithm"* header file.

Syntax: There are two implementations of make_heap() function.

Syntax 1:

```
void make_heap(RandomAccessIterator first, RandomAccessIterator last)
```

Parameters: The method takes two parameters.

- *first*: The pointer to the starting element of the sequence that has to be transformed into a heap.
- *last*: The pointer to the next address to the last element of the sequence that has to be transformed into a heap.

Syntax 2:

```
void make_heap (RandomAccessIterator first, RandomAccessIterator last, comp)
```

Parameters:

first: The pointer to the starting element of the sequence that has to be transformed into a heap.

last: The pointer to the next address to the last element of the sequence that has to be transformed into a heap.

comp: The comparator function that returns a boolean true/false of each element compared. This function accepts two arguments. This can be a function pointer or function object and cannot change values.

Program 1: Getting the maximum element of a vector.

```
1
2  // CPP program to get maximum
3  // element of a vector
4  #include <algorithm>
5  #include <iostream>
6  using namespace std;
7
8  // Drivers Method
9  int main()
10 {
11
12     // Creating a vector
13     vector<int> v = { 15, 6, 7, 12, 30 };
14
15     // Converting the vector into
16     // MAX_HEAP
17     make_heap(v.begin(), v.end());
18
19     // Printing the maximum element
20     // which is present at the root
21     cout << v.front() << endl;
22     return 0;
23 }
24
```

Run

Output:

```
30
```

How to convert a container to MIN_HEAP

The make_heap() function takes a third parameter as specified in Syntax 2 that is used to specify the order in which elements are to be arranged in the heap. We can pass “greater()” function to convert the heap into a MIN_HEAP. This function reverses the order of comparison.

Program 2: Getting the minimum element of a vector.

```
// CPP program to get minimum
// element of a vector
#include <algorithm>
#include <iostream>
using namespace std;

// Drivers Method
int main()
{

    // Creating a vector
    vector<int> v = { 15, 6, 7, 12, 30 };

    // Converting the vector into
    // MIN_HEAP
    make_heap(v.begin(), v.end(), greater<int>());

    // Printing the minimum element
    // which is present at the root
    cout << v.front() << endl;
    return 0;
}

```

Run

Output:

```
6
```

Implementation of push_heap() and pop_heap() function

push_heap(): The *push_heap()* function is used to insert elements into heap. The size of the heap is increased by 1 and the new element is placed appropriately in the heap. Thios is similar to the insert() function of heap.

pop_heap(): The *pop_heap()* function is used to delete the front element of the heap. The size of the heap is decreased by 1. The heap elements are reorganised accordingly after this operation.

Note: There are no changes in the size of the vector. pop_heap() moves the maximum or minimum element to the end of the vector.

Program 3: This program shows the implementation of push_heap() and pop_heap() fucntions.

```
// CPP program to illustrate
// push_heap() and pop_heap()
#include <algorithm>
#include <iostream>
using namespace std;

// Drivers Method
int main()
{

    // Creating a vector
    vector<int> v = { 15, 6, 7, 12, 30 };

    // Converting the vector into
    // MIN_HEAP
    make_heap(v.begin(), v.end(), greater<int>());

    // Printing the minimum element
    cout << v.front() << endl;

    // Removing the min element from the heap
    pop_heap(v.begin(), v.end(), greater<int>());
```

```
24
25      // Printing the min element in the
26      // remaining heap
27      cout << v.front() << endl;
28
29      // Overwriting the removed element by 2
30      v[4] = 2;
```

Run

Output:

```
6
7
2
```

Working:

1. A vector is created with the following elements.

```
Vector = {15, 6, 7, 12, 30}
```

2. The vector is converted to a MIN_HEAP

```
Heap = {6, 7, 12, 15, 30}
```

3. The minimum element at the root is displayed.

```
Print 6
```

4. Using the pop_heap() function, the element at the front or root of the heap i.e., 6, is removed. The heap size is reduced from 5 to 4. This element is moved to the end of the vector and there is no change in the size of the vector. Therefore 6 is moved to the last position(v[4]) in the vector and the present heap looks like:

```
Current Heap = {7, 12, 15, 30}
```

5. Displaying the minimum element at the root from the remaining heap.

```
Print 7
```

6. 6 is overwritten by 2.

7. 2 is added to the MIN_HEAP using the push_heap() function. Also, it is shifted to the front as it is a MIN_HEAP and 2 being the minimum value in the heap is shifted to the root position. The heap size is increased by 1 from 4 to 5.

```
Current Heap: {2, 7, 12, 15, 30}
```

8. The minimum element at the root is displayed.

```
Print 2
```

Time Complexities:

- **make_heaP():** This function takes O(n) time as the heap construction from a container is done in a linear time.

- **push_heap():** This function takes O(log n) time as it is just extraction and insertion of elements.

Implementation of sort_heap() function

The sort_heap() function is used to sort the heap in either an increasng or decreasing order, based on the 3rd parameter as mentioned in in Syntax 2. By default, the sort_heap() function sorts the heap in an increasing order.

Syntax:

```
sort_heap(start_address, end_address, order())
```

Parameters: The function accepts two parameters which are actually the iterators addressing to the given heap.

- *start_address:* It refers to the address of the first element of the heap.
- *end_address:* If refers to the address of the next contiguous location beyond the last element of the heap.
- *order()(optional):* This refers to the order in which the function must sort the heap. It is generally a comparison function that compares the elements of the container for eg., greater(). If this parameter is not passed then by default the function sorts the container in increasing order. The greater() function, in particular, reverses the order of the container.

Program 4: This program shows the implementation sort_heap() function.

Note: The same function(3rd parameter) must be passed to both the make_heap() and the sort_heap() function.

```
1
2  // CPP program to illustrate
3  // sort_heap() function
4  #include <algorithm>
5  #include <iostream>
6  using namespace std;
7
8  // Drivers Method
9  int main()
10 {
11
12     // Creating a vector
13     vector<int> v = { 15, 6, 7, 12, 30 };
14
15     // Converting the vector into
16     // MIN_HEAP
17     make_heap(v.begin(), v.end(), greater<int>());
18
19     // Sorts the heap in decreasing order
20     sort_heap(v.begin(), v.end(), greater<int>());
21
22     for (int x : v)
23         cout << x << " ";
24     return 0;
25 }
26
```

Run

Output:

```
30 15 12 7 6
```

Problem 5: Merge two sorted arrays in-place where no variable extra space must be used. In-place means the merging should be done in constant extra space. You must use the concept of make_heap() so that the time complexity can be reduced to O(mlogn), where m is the size of the first array and n is the size of the second array.

Example:

```
Input: a[] = {3, 20, 40}

       b[]: {2, 10, 12}

Output: a[] = {2, 3, 10}

       b[]: {12, 20, 40}

Explanation: The first array contains the smaller 3 elements.

       The second array contains the greater 3 elements.
```

Input: a[] = {30, 40}

b[]: {2, 8, 9, 10}

Output: a[] = {2, 8}

b[]: {9, 10, 30, 40}

Explanation: The first array contains the smaller 2 elements.

The second array contains the greater 4 elements.

Input: a[] = {3, 8}

b[]: {4, 5, 6}

Output: a[] = {3, 4}

b[]: {5, 6, 8}

Explanation: The first array contains the smaller 2 elements.

The second array contains the greater 3 elements.

Approach: As we have assumed the second array to be sorted, therefore the smallest element must be at the beginning of the array. Now traverse the first array and compare the first element of the first array with the 0th element of the second array. If the element in the first array is greater than the element at the second array, then

- Pop-out the element at b[]. This will move the element beyond the last position of second array.
- Swap the element at a[i] with the popped-out element.
- Push back the popped element to b[].

Solution:

```cpp
// CPP program to illustrate
// sort_heap() function
#include <algorithm>
#include <bits/stdc++.h>
#include <iostream>
using namespace std;

// merge() function for in-place sort and merge
void merge(int a[], int b[], int m, int n)
{

    // Traversing the first array
    for (int i = 0; i < m; i++) {
        if (a[i] > b[0]) {
            // Popping the element at root
            pop_heap(b, b + n, greater<int>());

            // Swapping the a[i] with popped element
            swap(a[i], b[n - 1]);

            // Pushing the popped element back
            push_heap(b, b + n, greater<int>());
        }
    }

    // Displaying the first array
    for (int i = 0; i < m; i++)
        cout << a[i] << " ";
    cout << endl;
```

Run

Output:

```
2 3 10
12 20 40
```

- prev_permutation in C++ STL

The **prev_permutation()** is a built-in function in C++ STL, which is used to rearrange the elements in the range [first, last) into previous lexicographically-ordered permutation. It basically finds the largest permutation of the input sequence, which is smaller than the given sequence. Let's consider the sequence {1, 2, 3, 4, 5}. So all lexicographically ordered permutations of the given sequence are:

```
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

So for {2, 1, 3}, the previous permutation is {1, 3, 2}.
Note: or a sequence of size N, there will be N! permutations.

Header File: The prev_permutation() function is declared in the *"algorithm"* header file.

Syntax:

```
bool prev_permutation(BidirectionalIterator first,

                      BidirectionalIterator last);
```

Parameter: Both the **first** and **last** are Bidirectional iterators pointing to the initial and final positions of the sequence. The range used is [first, last), which contains all the elements between first and last, including the element pointed by first and the index pointed beyond the last element. You can also pass random access iterators which are more powerful than bidirectional iterators.

Return value: The function returns **true** if it could rearrange the object as a lexicographically smaller permutation and return the previous permutation. Otherwise, the function returns **false** to indicate that no previous permutation is possible.

Examples:

```
Input: {2, 1, 3}
Output: {1, 3, 2}

Input: {2, 3, 1}
Output: {2, 1, 3}

Input: {1, 3, 2, 4, 5}
Output: {1, 2, 5, 4, 3}

Input: {5, 4, 1, 2, 3}
Output: {5, 3, 4, 2, 1}
```

Example:

```
1
2  // C++ program illustrating
3  // prev_permutation() function
4  #include <algorithm>
5  #include <iostream>
6  #include <vector>
7  using namespace std;
8
9  int main()
10 {
11
12     // The sample Vector
13     vector<int> v = { 5, 4, 1, 2, 3 };
14
15     // Performing prev_permutation
16     // operation
17     prev_permutation(v.begin(), v.end());
```

```
18
19      // Displaying the sequence
20      for (int x : v)
21          cout << x << " ";
22      return 0;
23  }
24
```

Run

Output:

```
5 3 4 2 1
```

Algorithm for prev_permutation() function:

Let the initial sequence be:

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

Step 1: Traverse from right, find the first element that is NOT in increasing order. Let this element be x.

```
For the given sequence {5, 4, 1, 2, 3},

x would be 4
```

Step 2: Find the smallest element on the right of x that is smaller than x. Let this element be y.

```
For the given sequence {5, 4, 1, 2, 3}

y = 3
```

Step 3: Swap x and y. This will give the lexicographically previous sequence.

```
After swap: {5, 3, 1, 2, 4}
```

Step 4: Now, to get the largest previous permutation which is smaller than the given permutation, just reverse the subsequence after new position of y or previous position of x.

```
After reversing: {5, 3, 4, 2, 1}
```

Time Complexity:

- Step 2 will take O(n) time.
- Step 3 will take O(n) time but it can be optimised using binary search into O(logn) time.
- Swapping in Step 3 is a constant operation.
- Reversing in step 4 will take O(n) time.

Therefore the overall time complexity of this operation is **O(n)** time.

reverse() in C++ STL

The **reverse()** is a built-in function in C++ STL which is used to reverse the order of the elements in the range of first and last element of any given container.

Header File: The reverse() function is declared in the *"algorithm"* header file.

Syntax:

```
bool reverse(BidirectionalIterator first,

                      BidirectionalIterator last);
```

Parameter: Both the **first** and **last** are Bidirectional iterators pointing to the initial and final positions of the sequence. The range used is [first, last], which contains all the elements between first and last, including the element pointed by first and the index pointed beyond the last element.

Program 1: Reverse a given vector.

```
Input: vector v = {10, 20, 30}

Operation: reverse(v.begin(), v.end());

Output: {30, 20, 10}

Explanation: In the above function, we have applied reverse()
in the range of [v.begin, v.end). Here v.end points to the element
beyond the last element of the vector.
```

```cpp

// C++ program illustrating
// reverse() function
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{

    // The sample vector
    vector<int> v = { 10, 20, 30 };

    // Performing reverse operation
    // from begin to end
    reverse(v.begin(), v.end());

    // Displaying the sequence
    for (int x : v)
        cout << x << " ";
    return 0;
}

```

Run

Output:

```
30 20 10
```

Program 2: Reverse a given vector while excluding the first element.

```
Input: vector v = {10, 20, 30, 40, 50}

Operation: reverse(v.begin()+1, v.end());

Output: {10, 50, 40, 30, 20}

Explanation: In the above function, we have applied reverse()
in the range of [1, end), therefore the 0th element remains unaffected. Here v.end
points to the element beyond the last element of the vector.
```

```cpp

// C++ program illustrating
// reverse() function
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{

    // The sample vector
    vector<int> v = { 10, 20, 30, 40, 50 };

    // Performing reverse operation
    // from begin+1 to end
    reverse(v.begin() + 1, v.end());

    // Displaying the sequence
    for (int x : v)
        cout << x << " ";
    return 0;
}

```

Run

Output:

```
10 50 40 30 20
```

Program 3: Reverse a given array.

Input: arr[] = {10, 20, 30, 40, 50}

Operation: reverse(arr, arr+5);

Output: {50, 40, 30, 20, 10}

Explanation: In the above function, we have applied reverse()
in the range of [0, 5), therefore the 0th element remains unaffected. Here arr+5
points to the element beyond the last element of the array.

```cpp

// C++ program illustrating
// reverse() function
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{

    // The sample array
    int arr[] = { 10, 20, 30, 40, 50 };

    // Performing reverse operation
    // from [0 to 5)
    reverse(arr, arr + 5);

    // Displaying the sequence
    for (int x : arr)
        cout << x << " ";
    return 0;
}

```

Run

Output:

```
50 40 30 20 10
```

Program 4: Reverse a given string.

```
Input: String s= "geeks"

Operation: reverse(s.begin(), s.end());

Output: skeeg

Explanation: In the above function, we have applied reverse()
function on the string. Here s.end() points to the element beyond
the last element of the string.
```

```
1
2  // C++ program illustrating
3  // reverse() function
4  #include <algorithm>
5  #include <iostream>
6  #include <vector>
7  using namespace std;
8
9  int main()
10 {
11
12     // The sample string
13     string s = "geeks";
14
15     // Performing reverse operation
16     // on the string
17     reverse(s.begin(), s.end());
18
19     // Displaying the string
20     cout << s;
21     return 0;
22 }
23
```

Run

Output:

```
skeeg
```

Time Complexity: The reverse() function takes linear time for the whole operation, O(n).

merge() in C++ STL

The **merge()** is a built-in function in C++ STL, that takes two sorted containers and merge them. This merged container is stored in a third container which is also passed as a parameter to the function. If the first container size is m and the second container size is n, then the third container size must be greater than or equal to m+n.

Syntax:

```
merge(container1_first, container1_last,

      container2_first, container2_last,

      container3_first)
```

Parameters:

- **container1_first:** Input iterator to the initial position of the first sequence.

- **container1_last:** Input iterator to the final position of the first sequence.
- **container2_first:** Input iterator to the initial position of the second sequence.
- **container2_last:** Input iterator to the final position of the second sequence.
- **container3_first:** Output Iterator to initial position of the resultant container.

Return Value: Iterator to the last element of the resulting container.

Program 1: This program takes two sorted vectors of size 3 each and merges them to a 3rd vector.

```
Input: v1 = {10, 20, 40}

       v2 = {5, 15, 30}

Output: v3 = {5, 10, 15, 20, 30, 40}
```

```cpp

// C++ program illustrating
// merge() function
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{

    // The sample input Vectors
    vector<int> v1 = { 10, 20, 40 };
    vector<int> v2 = { 5, 15, 30 };

    // Output vector
    vector<int> v3(6);

    // Performing merge operation
    merge(
        v1.begin(), v1.end(),
        v2.begin(), v2.end(),
        v3.begin());

    // Displaying the v3
    for (int x : v3)
        cout << x << " ";
    return 0;
}

```

Run

Output:

```
5 10 15 20 30 40
```

Program 2: This program takes two sorted arrays of size 3 and 4 respectively and merges them to a 3rd array.

```
Input: ar1= {10, 20, 30}

       ar2 = {5, 15, 40, 80}

Output: ar3 = {5, 10, 15, 20, 30, 40, 80}
```

```cpp

// C++ program illustrating
// merge() function
#include <algorithm>
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

int main()
{

    // The sample input arrays
    int ar1[] = { 10, 20, 30 };
    int ar2[] = { 5, 15, 40, 80 };

    // Output array
    int ar3[7];

    // Performing merge operation
    merge(ar1, ar1 + 3, ar2, ar2 + 4, ar3);

    // Displaying the ar3
    for (int x : ar3)
        cout << x << " ";
    return 0;
}
```

Run

Output:

```
5 10 15 20 30 40 80
```

Time Complexity: For the containers of size m and n respectively, the time taken by the merge() function is O(m+n).

Practical Implementation: This can be used during the MergeSort to merge and sort a container.

Note: The list has its own merge function, so it is not a good practice to use this merge() function on a list container.
It is general advice to not to use the merge function of the C++ STL library, for the container class that have there own supporting functions to do the same operations. For eg., Map and Set.