A Sorting Algorithm is used to rearrange a list of elements according to a comparison based operator. It is basically a permutation of elements a1, a2,..., an such that a1'<=a2'<=...<=an'. It has three major components -

Order: It refers to the manner in which the elements are to be sorted.
Eg. Arr[4] = {-5, 7, 1, -9}
Ascending order of value: [-9, -5, 1, 7]
Descending order of value: [7, 1, -5, -9]
Ascending order of magnitude: [1, -5, 7, -9]

Stability: For an array with duplicate elements, more than one permutation of the array will represent the sorted form of the array.
Eg. Arr[4] = {2(1), 1, 2(2), 3}
Stable: [1, 2(1), 2(2), 3] (Duplicate elements appear in the original order)
Unstable: [1, 2(2), 2(1), 3] (Duplicate elements do not appear in the original order)

Comparators: A comparator basically tells the rule on the basis of which the method has to sort the received data input. The in-built sorting method of any language library generally takes two parameters - data and the comparator.

Call statement: sort(v.begin(), v.end(), cmp)

Given below is an example of a comparator that is sorting elements into ascending order while ensuring stability:

```
bool cmp(pair<int, int> x, pair<int, int> y){
  if(x.first!=y.first){
    return x.first<y.first;
  }
  else{
    return x.second<y.second;
  }
}
```

Note:

bool cmp(int x, int y);
If the above comparator function returns "TRUE" then 'x' will come before 'y', otherwise 'y' will appear before 'x' in the sorted array.
Avoid using equal to sign (<= or >=) in the comparator function as sometimes it gives an error in languages like C++, which is basically due to a bug in the internal implementation.

Custom Comparators

In this lecture, we will learn how to write custom comparators by discussing a few problems.

Q. We have been given the coordinates of 'N' points and we have to find the first 'k' points which are closest to the origin.

Approach: We can create a comparator function to sort the points according to their distance (d=√(x^2 + y^2)) from the origin.
Note: We can compare (x^2+y^2) rather than √(x^2 + y^2) for faster calculation.

Q. We have been given an array Arr[N] such that 0<=Arr[i]<=100 and we have to sort it according to the following rule:

Smaller frequency elements should come first
Smaller value element should come first if the frequency is same
Smaller index element should come first if the value is same

Approach:

We can initialise a frequency array f[101] with 0 to count the frequency of each value.
We can create a structure containing the frequency, value & index of an element and pass it to the comparator function for sorting the array according to the given conditions.

Note: The time complexity is equal to O(NlogN) which is typically the time taken by the default sort function in the language library.

Custom Sorting

We have been given 2 strings 'S' and 'T' both containing lowercase alphabets only. We have to sort characters of 'T' based on the order of characters in 'S' given that all the characters in 'S' are distinct.
Input: S = "cba"
T = "abcd"
Output: T = "cbad"

Approach:

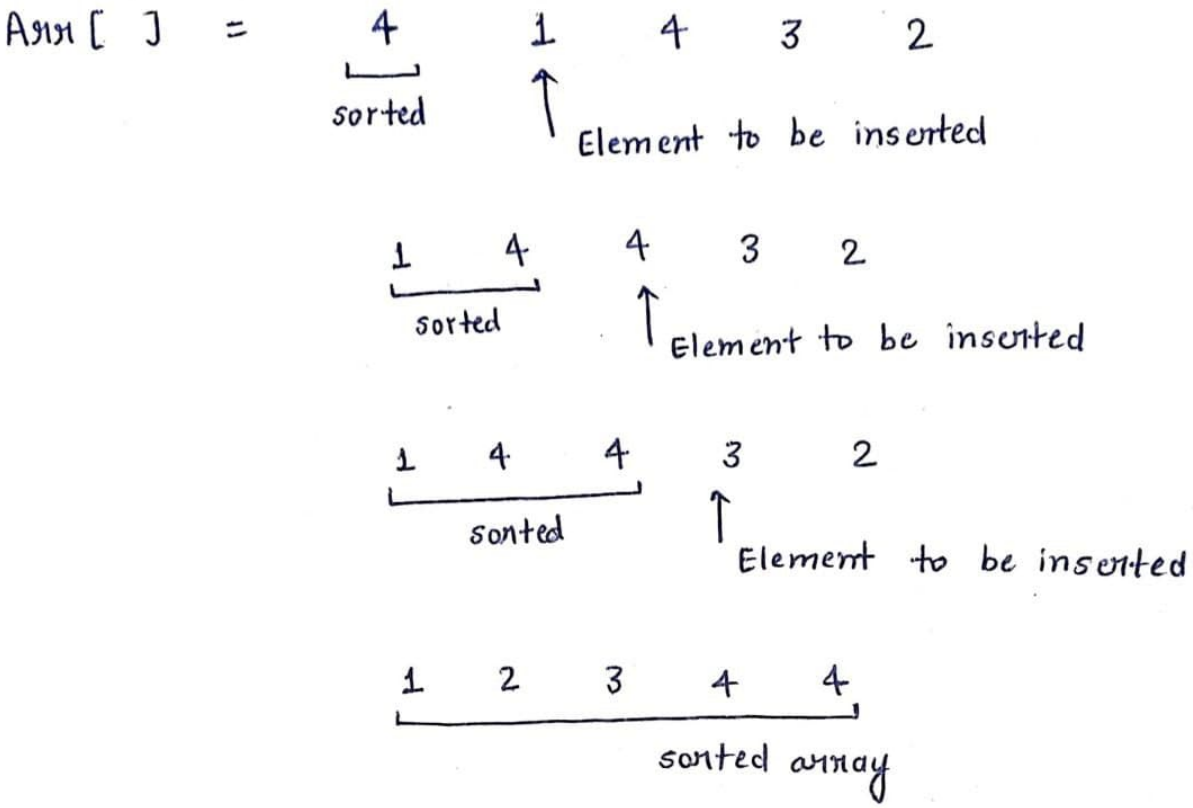
We can create an array - rank[26] to store the ranks of all 26 characters according to the string S.
For characters that do not occur in S, we can give them a rank of infinity i.e. INT_MAX.
We can initialise a vector of pairs containing characters of string T and their rankThe pairs can be passed to the custom comparator to define the sorting rule according to the given problem statement.

Insertion Sort

Sorting algorithms are mainly of two types:

Comparison based sorting - In this type of algorithm, we need to compare the array elements with each other in order to sort them. They are generally O(N2) or O(NlogN) algorithms. Eg. Insertion sort, Bubble sort etc.
Non-comparison based sorting - This type of sorting algorithm does not involve comparison between elements.

Insertion sort is a sorting algorithm where the array is virtually divided into two halves - sorted and unsorted. The values from the unsorted array are picked one by one and placed at their correct position in the sorted array. It is a stable algorithm as it maintains the order of duplicate elements.

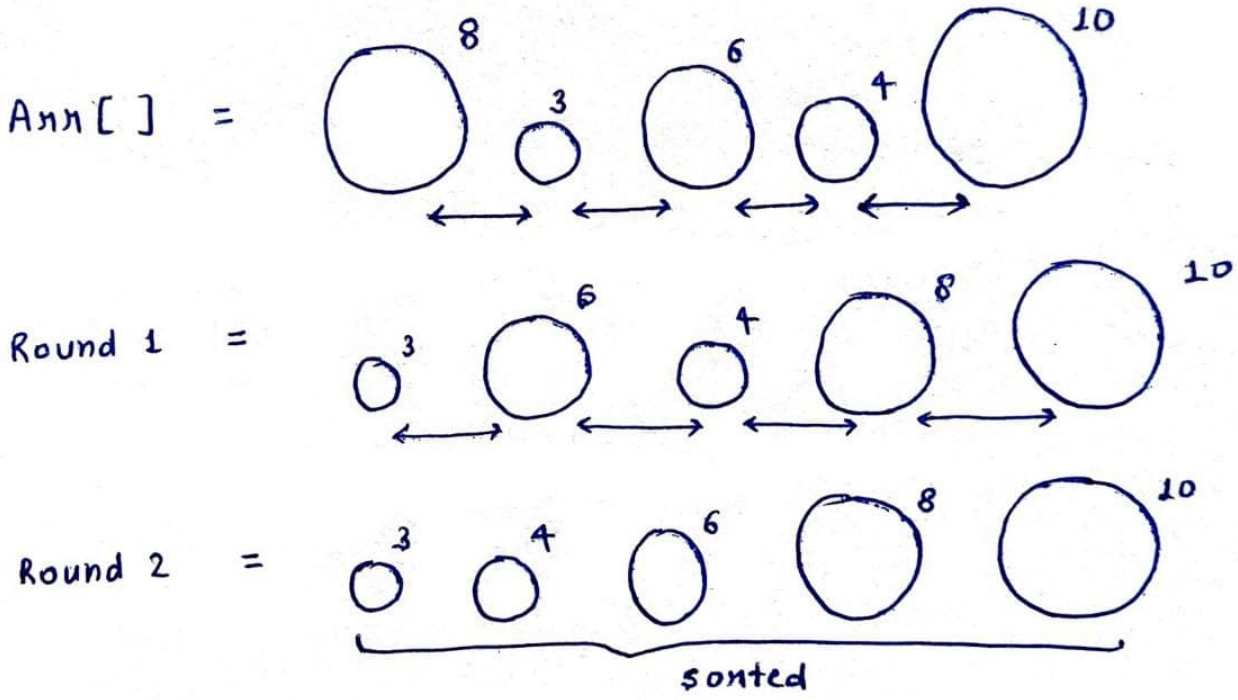


The time complexity of Insertion sort depends on the degree of unsortedness of the array. In the worst-case scenario, it is O(N^2).

Bubble Sort

Bubble sort is a simple sorting algorithm where the adjacent array elements are repeatedly swapped if they are in the wrong order until the array becomes sorted. We assume the array elements to be a bubble of different sizes(larger the element, larger the bubble) and perform multiple iterations and in each iteration, we get the largest element of the remaining array at its appropriate position.

Time complexity: O(N^2)
Worst case scenario: When the array is in descending order



Trouble sort: Bubble sort can also be explained by considering subarrays of size 2 which are reversed in each iteration if their order is incorrect.

Similarly, can we consider subarrays of size 3 in each iteration and revert them if the first and the third element are not in order to get a sorted array?

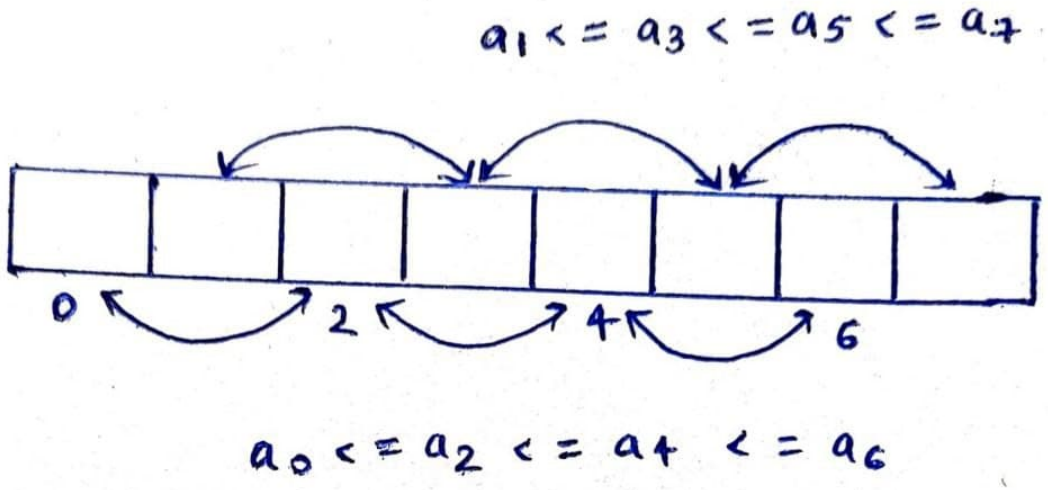
No, it will not give us a sorted array in each case.

Eg. Arr = [10, 11, 1, 7] -> [1, 7, 10, 11] (sorted output)

Arr = [4, 3, 2, 1] -> [2, 1, 4, 3] (unsorted output)

Can you predict the result of trouble sorting without actually implementing it?

On carefully observing, we note that the elements at even indices and at odd indices get sorted separately. Thus, we can create two temporary arrays to store the elements at even and odd indices. We can sort them separately with the help of the merge sort or inbuilt sort function and merge them to get the desired output.



Time complexity: O(Nlog2N)
Space complexity: O(N)

Dealing with Absolute Values

We have been given two arrays x[N] and y[N] containing the x and y coordinates of 'N' points lying on the coordinate axis. We have to find the sum of the Manhattan distances between all pairs of points.

Manhattan distance between two points (x1,y1) and (x2,y2) is equal to |x2-x1| + |y2-y1|.

Approach:

Brute Force - Consider all the NC2 pairs and calculate the sum of the Manhattan distances between them.
Time complexity: O(N^2)
Space complexity: O(1)
Using Sorting:
If we consider 5 points P0(x0, y0), P1(x1, y1), P2(x2, y2), P3(x3, y3), P4(x4,y4). Then the sum of Manhattan distance of point P0 from other points will be -
Sum(P0) = (|x1 - x0| + |y1 - y0|) + (|x2 - x0| + |y2 - y0|) + (|x3 - x0| + |y3 - y0|) + (|x4 - x0| + |y4 - y0|)
We know that the sum of absolute differences of x coordinates is independent of the sum of absolute differences of y coordinates. Therefore, we can calculate them individually i.e.
SumP0 = Xdiff(P0) + Ydiff(P0) where,
Xdiff(P0) = |x1 - x0| + |x2 - x0| + |x3 - x0| + |x4 - x0|
Ydiff(P0) = |y1 - y0| + |y2 - y0| + |y3 - y0| + |y4 - y0|
But in the above expression, we do not know whether the mod operator will open with a positive or a negative sign, therefore **we can sort x[N] and y[N] in ascending order to be able to process the mod operator**.
Suppose the x coordinates are sorted in ascending order then,
Xdiff(P0) = (x1 + x2 + x3 + x4) - 4x0
Similarly, Xdiff(P1) = (x2 + x3 + x4) - 3x1

We can generalise it as **Xdiff = ΣXdiff(Pi) = Σ(Σxi+1 - (N-i-1)*xi)**
Ydiff = ΣYdiff(Pi) = Σ(Σyi+1 - (N-i-1)*yi)
Time complexity: O(Nlog2N)
Space complexity: O(1)

Solve the Equation

We have been given an integer array Arr[N] containing distinct elements and we have to find the number of sextuples satisfying the equation - **(a*b+c)/d-e = f**. Here, one element can play the role of multiple variables.

Input: Arr[1] = {1}
Output: 1 i.e. {1, 1,1,1,1,1}

Approach:

Brute Force - We can run six nested loops to generate all the possible combinations and check if they satisfy the given equation or not.
Time complexity: O(N^6)
Note: We may encounter an edge case of 0/0, hence discard all combinations containing d=0.
We can rearrange the given equation to a*b+c = (f+e)*d. Now, we can separately find all the possible values of LHS and RHS and store them in a vector LHSArr[N^3] and RHSArr[N^3].
Time complexity: O(N^3) ∵ There are total N^3 combinations for both LHS and RHS

The equation will be satisfied only when LHS=RHS, thus we can iterate on the vector containing the values of LHS and use sorting and binary search to find the frequency of the value in RHS.

Time complexity for sorting RHSArr[N^3] = O(N^3logN)
Time complexity for iterating LHSArr[N^3] = O(N^3)
Time complexity for applying binary search on RHSArr[N^3] = O(log2N)

Overall Time complexity: O(N^3logN)
Space complexity: O(N^3)

Note: Do not forget to reject the values in RHS when d=0

Next Greater Permutation