Sorting

defined in #include <algorithm>

- Sort() provide sorting of the data stored in containers which allow random access. → accessing in constant time
  (Array, vector, deque)

- For Array → points to element after last index

→ sort(a, a+n)
  points to first index

→ sort(a, a+n, greater<int>())
  comparison function

- For vector

→ sort(v.begin(), v.end());
→ sort(v.begin(), v.end(), greater<int>()); → ये bracket भूलना नहीं

→ You can create your own Comparison function:

```
struct Point
{   int x,y;
};
bool mycomp(Point p1, Point p2)
{   return (p1.y > p2.y)
}
int main()
{
    Point a[] = {{3,10}, {2,8}, {5,4}};
    sort(a, a+3, mycomp);
    for(auto i:a)
        cout<< i.x<<" "<< i.y<< endl;
}
```

O/p → 3 10
2 8
5 4

जिसका y बड़ा वो ऊपर
(decreasing order of y value)

- Java sort function uses QuickSort, InsertionSort, MergeSort
- Python ———— Tim Sort (Merge Sort + Insertion Sort)

** Worst & Average case O(nlogn)
C++ uses IntroSort (Hybrid of QuickSort, HeapSort, InsertionSort)

- By default it uses Quick Sort, if QS is doing unfair partitioning and taking more than (nlogn) time it switches to HeapSort and if Array size is very small it switches to Insertion Sort.

→ Stability in Sorting Algorithm :
- if two element/item have same value, then they should appear in the same order as they appear in the original Array.
- Stability is important when we have elements like this
eg: {("Anil", 50), ("Ajay, 80), ("Pijush", 50), ("Ramesh", 80)}

~~{("Anil", 50), ("Ajay", 80)}~~, (

{("Anil", 50), ("Pijush", 50), ("Ajay", 80), ("Ramesh", 80)}

O/P of a stable sort algorithm

- Stability is check when we have more than one element in a object.

O/P → {("Pijush", 50), ("Anil", 50), ("Ajay", 80), ("Ramesh", 80)}

Unstable Algo

- unstable Algo doesn't care of the order

→ Eg of Stable Sorts: Bubble, Insertion, Merge

→ Eg of Unstable Sort: Selection, QuickSort, HeapSort

SORTING METHODS

(i) Bubble Sort:

In this sorting algorithm elements are compared with the next adjacent elements and if required we need to swap the elements as per the condition.

eg:

0	1	2	3	4	5
4	3	1	8	6	2

*sorting in ascending order

if (4>3) then swap

3	4	1	8	6	2

4>1 swap

3	1	4	8	6	2

4 ≯ 8 no swapping

8>6 swap

3	1	4	6	8	2

8>2 swap

3	1	4	6	2	8

→ further repetition 3>1 swap---

- एक बार जब अंदर वाला for loop complete होगा तो यहां तक पहुंचेगा।

*** एक बार जब अंदर वाला for loop complete हो जाएगा तो, जो सबसे largest number होगा वो आखरी में पहुंच जाएगा (here 8)

Programme

```
main()
{
        int A[100], i, j, n, t;
        pf("enter the size of array");
        scanf("%d", &n);
        pf("enter %d integers", n);
        for(i=0; i<n; i++)
        {
                scanf("%d", &A[i]);
        }
        for(i=0; i<n; i++)
        {
                for(j=0; j<n-1; j++)
                {
                        if(A[j]>A[j+1])
                        {
                                t = A[j];
                                A[j] = A[j+1];
                                A[j+1] = t;
                        }
                }
        }
        pf("Sorted array is \n");
        for(i=0; i<n; i++)
                pf("%d ", A[i]);
}
```

→ ①

→ To optimize more
j<n-i-1

If array is already sorted then we can optimize our code by using flag variable

```
for(——)
{
    flag = true;
    for( )
    {
    }
    if(flag == true)
        break;
}
```

Output

```
enter the size of array 6
enter 6 integers 1 5 2 0 6 4
Sorted array is
0 1 2 4 5 6
```

(11) Selection Sort

- The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning. or vice-versa

- The algorithm maintains two subarrays in a given array.
  1) The subarray which is already sorted.
  2) Remaining subarray which is unsorted.

- In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

8	5	1	6	2	0	7

after one iteration

0	5	1	6	2	8	7

after 2nd

0	1	5	6	2	8	7

subarray (sorted side) — subarray (unsorted side)

- एक बार जब for loop चल जाएगा अंदर वाला, तो minimum element 1st position पर आ जाएगा and so on

D. Selection Sort does less memory write compared to QuickSort, MergeSort, InsertionSort etc. But Cycle Sort is ahead in terms of memory write.
- Not Stable, In-place (doesn't use extra space)
- $O(n^2)$ → always
- Basic Idea for heap sort

Programme

```
main()
{
    int i, j, A[1000], n, t, min, k;
    same line → ①
    for(i=0; i<n-1; i++)
    {
        min = A[i];
        for(j=i+1; j<n; j++)
        {
            if(min > A[j])
            {
                k = j;
                min = A[j];
                A[k] = min;
            }
        }
        if(A[i] != min)  → swaping condition
        {
            t = A[i];
            A[i] = A[k];
            A[k] = t;
        }
    }
    printf("Sorted array is \n");
    for(i=0; i<n; i++)
        printf("%d  ", A[i]);
}
```

swap ~~condition~~ is done

Output: enter the size of array 8
enter 8 integers 6 5 7 4 8 3 9 1
Sorted array is
1 3 4 5 6 7 8 9

- in some case the elements are at the right place so don't need to swap. min
∴ we will swap only if element is not at right place.

(iii) **Insertion Sort**

- it is based on the principle that a new key K is inserted at its appropriate position in an already sorted sub list.

- say List $L = \{K_1, K_2, K_3 \text{ --- } K_{n-1}, K_n\}$

→ in first pass K_2 compared to K_1, if not at appropriate place swap.

→ in second pass K_3 compared to K_1. if condition true swap ~~then~~

say → $\{K_3, K_2, K_1 \text{ ---}\}$

and also K_1 compared to K_2, if condition true swap

say → $\{K_3, K_1, K_2 \text{ ---}\}$

and so on ---

eg: $\{16, 36, 4, 22, 100, 1, 54\}$

after 1st pass → $\{[16, 36], 4, 22, 100, 1, 54\}$ → it is compared

sorted side

after 2nd pass → $\{[4, 16, 36], 22, 100, 1, 54\}$

after 3rd pass → $\{[4, 16, 22, 36], 100, 1, 54\}$

after 4th " → $\{[4, 16, 22, 36, 100], 1, 54\}$

after 5th " → ~~{4,~~ $\{[1, 4, 16, 22, 36, 100], 54\}$

" 6th " → $\{1, 4, 16, 22, 36, 54, 100\}$ → sorted list

Bubble Sort → Adjacent element compare
Selection Sort → Subarray sorted & unsorted
Insertion Sort → Key को correct position

(I) Insertion Sort :

- it is in-place (no extra space)
- Stable
- ✓ It is most effective if array size is small
- Timsort (Merge Sort + Insertion Sort)

```
void insertionSort(int a[], int n)
{
    for(int i=0; i<n; i++)
    {
        int Key = a[i];
        int j = i-1;
        while (j>=0 && a[j] > Key)
        {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = Key;
    }
}
```

0 ——— i ——— n-1

Sorted	Key	Unsorted

Worst → $O(n^2)$, Best → $O(n)$

eg: 5, 20, 40, 60 | 10, 30 (indexes 0, 1, 2, 3 | 4, 5) Say i=4

← Sorted →

5, 20, 40, 60, 10, 30

← j i/Key

Merge Sort:

- it is an Divide & Conquer Algorithm
  (divide the array in two parts, sorts these two parts, then merge this two parts)
- Stable
- $O(n\log n) \leftarrow TC$ | $O(n) \leftarrow SC$ (not in-place)
- Block Merge Sort:

D is a sorting algo combining atleast two merge operations with an insertion sort to arrive at $O(n\log n)$ in-place stable sorting algo.

- For Array Quick Sort better than MS
- For linked list MS better uses $O(1)$ Aux space
- Well suited for external sorting (bring parts of array in RAM then sort)