Sorting

Arranging of element in ordered fashion.

Some well-known standard techniques for sorting techniques :-

$O(n^2)$

(1) Bubble Sort

```
void bubble_Sort(int arr[], int n){
  for (int i=0 ; i<n-1 ; i++) {
     for(int j=0 ; j<n-i-1 ; j++ {
          if (arr[j] > arr[j+1]) {
             swap(arr[j], arr[j+1]);
          }
     }
  }
```

(2) Selection Sort

```
void selection_Sort(int arr[], int n) {
  for(int i=0 ; i<n ; i++) {
          int K = i;
    for(int j=i+1 ; j<n ; j++) {
      if(arr[K] < arr[j]) {
             K = j
      }
   swap (arr[K], arr[i]);
  }
}
```

③ Insertion Sort

```
void insertion_Sort(int arr[], int n){
    for(int i=1, i<n ;i++){
        int K = arr[i];
      for(int j=i-1 ; j>=0 && arr[j] >K ;j--){
            arr[j+1] = arr[j];
      }
    arr[j+1] = K;
}
```

O(nlogn)

- Merge Sort
- Quick Sort
- Randomized Quick Sort
- Heap Sort

Merge Sort

An divide & conquer sorting algorithm. It breaks the array in multiple smaller parts and then merge them.

```
void mergeSort(int arr[], int low, int high){
    if(low < high){
        int mid = (low+high)/2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid+1, high);
        merger(arr, low, mid, high);
    }
}
void merger(int arr[], int low, int mid, int high){
 int p=low; int q=mid+1; int n=high-low+1;
 int temp[n]; int i=0;
while(p<=mid && q<=high){
    if(arr[p] <arr[q]){
        temp[i++] = arr[p++];
    }else{
        temp[i++] = arr[q++];
    }
}
while(p<=mid){
   temp[i++] = arr[p++];
}
while(q<=high){
   temp[i++] = arr[q++];
}
  for(int i=low; i<=high; i++){
        arr[i] = temp[i-low];
  }
}
```

Quick Sort

```
int partition(int a[], int lb, int ub){
  int pivot = a[lb], i=lb, j=ub+1;
 do{
    do{
      i++;
    }while(i<=ub && a[i]<pivot);
     do{
       j--;
     }while(a[j] > pivot);
     if(i<j){
       swap(a[i], a[j]);
     }
   }while(i<j);
   if(lb!=j){
    swap(a[lb], a[j]);
   }
   return j;
}
 void quick_Sort(int a[], int lb, int ub){
    int pivot;
    if(lb<ub){
     pivot = partition(a, lb, ub);
     quick_Sort(a, lb, pivot-1);
     quick_Sort(a, pivot+1, ub);
   }
}
```

Heap Sort

```
void heapify(int a[], int i, int n){
    int left = 2*i+1;
    int right = 2*i+2;
    int largest = i;
    if(left<n && a[left] > a[largest]){
        largest = left;
    }
    if(right<n && a[right] > a[largest]){
        largest = right;
    }
    if(i!=largest){
        swap(&a[i], &a[largest]);
        heapify(a, largest, n);
    }
}

void heap_Sort(int a[], int n){
    for(int i=(n/2)-1; i>=0; i--){
        heapify(a, i, n);
    }
    for(int i=n-1; i>=0; i--){
        swap(&a[0], &a[i]);
        n--;
        heapify(a, 0, n);
    }
}
```

O(n)

Count Sort

Step 1 :- Find the max element in array.

Step 2 :- Make an count array of size = max element + 1

Step 3 :- Calculate cumulative sum of count array.

Step 4 :- Traverse the array from last and for each element get it position from count array

arr[] = {2, 3, 4, 1, 2, 2}

0	1	3	1	1

⇒ Step 2

0	1	4	5	6

⇒ Step 3

```
for (int i=n; i>=0; i--) {
    Output[count[a[i]] -1] = a[i];
    -- count[a[i]];
}
```

Radix Sort

Similar to count sort, but here count array is of fixed size.
So these thing we need to add to convert count sort into radix sort.

(i)

```
void radix_Sort (int arr[], int n){
   int maximum = INT_MIN;
   for(int i=0; i<n; i++){
      maximum = max(maximum, arr[i]);
   }
  for(int exp=1; (maximum/exp)>0; exp*=10){
      Count_Sort(arr, n, exp);
   }
}
```

(ii)

Change,

arr[i] ⟹ (arr[i] / exp) % 10

Sorting in C++

Increasing Order

```
sort(v.begin(),v.end()); //vector,string
sort(a,a+n); //array
```

Decreasing Order

```
sort(v.rbegin(),v.rend()); //vector
```

We can also sort a pair & tuple by using inbuilt sort.

1st element → 2nd element → 3rd element

pair

tuple

Searching

i) Linear Search O(n)
ii) Binary Search (logn)

For performing Binary Search, Array must be sorted.

```
int binarySearch(int a[],int l,int r,int x){
    if(r>=l){
      int mid = l + (r-l)/2;
       if(a[mid]==x){
          return 1;
       }
       if(a[mid]<x){
        return binarySearch(a,mid+1,r,x);
       }
      return binarySearch(a,l,mid-1,x);
   }
 return -1;
}
```

Iterative approach can be done similarly.

Binary Search C++ STL

binary_search(a, a+n, x)

↳ returns true or false

Some functions based on it :-

- lower_bound(a, a+n, x) ⇒ pointer to atleast x
- upper_bound(a, a+n, x) ⇒ pointer to element greater than x

Store them in auto a & b

b - a = count of element whose value is x

A better way,

```
auto r = equal_range(a, a+n, x);
cout << r.second - r.first << endl;
```