Todays Content

1. Overriding Comperator
2. Given ar[N] sort elements based on freq
3. longest number in array
4. k closest Points in Origin
5. Sort in Decreasing Order

Intro: When we want our own sorting order, we use concept of compare function.

Compare function:

Define order for 2 ele, using that we can define order for all arr[] elements.

#Syntax:

```
bool f_name(Type s1, Type s2){  # Both s1 & s2 are same type.
    if we want s1 before s2: return True
        else if s2 before s1: return False;
}

int arr[];
sort(ar, arr+n, f_name); #Sort ar based on f_name

vector<int> v;
sort(v.begin(), v.end(), f_name); #Sort v based on f_name
```

TC: O(nlogn * Time taken for cmp function);

```
void mergeSort(int ar[], int s, int e){ TC: O(NlogN/2)  SC: O(N+logN/2)
    if(s==e){return;}
    int m=(s+e)/2;
    mergeSort(ar, s, m);
    mergeSort(ar, m+1, e);
    Merge(ar, s, m, e); # Merge both subarray [s..m] & [m+1..e]
}
```

ar[]: { .. s s+1 ... m | m+1 ... e .. }

tmp[ ]

ar[]: { .. s s+1 ... m | m+1 ... e .. }

```
void Merge(int ar[], int s, int m, int e){

    int c[e-s+1];
    int p1=s, p2=m+1, p3=0;

    while(p1<=m && p2<=e){ # 1st sub: [s..m]  2nd sub = [m+1..e]
        # s1        s2
        if(A[p1]<=A[p2]){
            c[p3]=A[p1]; p3++; p1++;
        }
        else{
            c[p3]=A[p2]; p3++; p2++;
        }
    }

    while(p1<=m){
        c[p3]=A[p1]; p3++; p1++;
    }
    while(p2<=e){
        c[p3]=A[p2]; p3++; p2++;
    }

    for(int i=s; i<=e; i++){
        A[i]=c[i-s];
    }
}
```

ar[]: { s .. p1 ... m | m+1 .... e

```
if(fnme(s1, s2)){
    # if function return True: s1
}
else{
    # if function return False: s2
}
```

Q: Given ar[N] elements sort elements in decreasing order.

Eg: ar[] = { 6 4 3 2 10 14 12 }

ar[] = { 14 12 10 6 4 3 2 }

```
bool dec(int s1, int s2){ TC: O(NlogN * 1) = O(NlogN)
    if(s1>s2){ #s1 come first
        return true;
    }
    else { #s2 come first
        return false;
    }
}

vector<int> sortDec(vector<int> & ar){
    sort(ar.begin(), ar.end(), dec); # sort ar in dec order
    return ar;
}
```

Given ar[N] elements sort elements in increasing order of frequency.

Note: if 2 elements have same freq, smaller element should come first

Ex:

	0	1	2	3	4	5	6	7
ar[] = {	7	2	9	2	3	3	2	5 }
ar[] = {	5	7	9	3	3	2	2	2 }

```
unordered_map<int, int> um;

bool freq(int s1, int s2){

    if( um[s1] < um[s2] ){ # s1 come first
        return true;
    }
    else if (um[s1] == um[s2]) {
        if(s1 < s2){ return true; }
        else { return false; }
    }
    else { #um[s1] > #um[s2];
        return false;
    }
}

vector<int> sortFreq(vector<int>& ar){
    # Complete erase um; # Reinitilization: TODO

    for(int i=0; i<ar.size(); i++){
        um[ar[i]]++;
    }

    return ar;
}
```

return s1 < s2;

s1

3 < 8 : True : s1

8 < 3 : False : s2

3B Given vector of pairs, each pair is representing 2D point
Sort points based on their distance to origin in increasing order

Note1: If 2 points have same distance to origin point with smaller x should come first

Note2: Distance between 2 points (x_1, y_1) $(x_2, y_2) = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$

Distance between origin $(0,0)$ & (x, y) : $\sqrt{(x-0)^2 + (y-0)^2}$

obs: if $\sqrt{x} > \sqrt{y}$: $x > y$. # d^2 from $(0,0)$ to $(x, y) = x^2 + y^2$

Ex:

V:	x y	#d	#d^2		x y
0	<2, 3>	$\sqrt{13}$	13	0	<2, 3>
1	<1, 4>	$\sqrt{17}$	17	1	<3, 2>
2	<5, 2>	$\sqrt{29}$	29	2	<1, 4>
3	<3, 3>	$\sqrt{18}$	18	3	<3, 3>
4	<4, -2>	$\sqrt{20}$	20	4	<4, -2>
5	<3, 2>	$\sqrt{13}$	13	5	<5, 2>

x_1 y_1 x_2 y_2

```
bool pdist(pair<int,int> p1, pair<int,int> p2){
    int x1 = p1.first, y1 = p1.second;
    int x2 = p2.first, y2 = p2.second
    int d1 = x1^2 + y1^2;  # Square distance from (0,0) to (x1, y1)
    int d2 = x2^2 + y2^2;  # Square distance from (0,0) to (x2, y2);
    if(d1 < d2){ return true;}
    else if(d1==d2){
        return x1 < x2;
    }
    else{ # d1 > d2
        return false;
    }
}
```

if $x_1 < x_2$: True : s_1 comes first
else : False : s_2 comes first

```
vector<pair<int,int>> sortDist(vector<pair<int,int>> &ar){

    sort(ar.begin(), ar.end(), pdist);

}
```

48 Largest Number:

Given an ar[N], arrange them in such a way that by concenating all of them from left to right it should from largest number.

Note: Result may be very large, so return a string.

Ex1: ar[] = { 2 3 9 0 } (0 1 2 3)
→ { 3 9 0 2 } = 3902
→ { 9 3 2 0 } = 9320

Ex2: ar[] = { 99 90 98 } → { 99 98 90 } = 999890

Ex3: ar[] = { 998 9 }
→ { 998 9 } = 9989
→ { 9 998 } = 9998

Ex4: ar[] = { 30 3 }

Idea:

1. Sort ar[] in descending order & concatenate *

2. #Hint: Take any 2 elements & fix order on them

ele1	ele2	ele1 + ele2	ele2 + ele1	order
89	8	898	889	ele1 first
90	9	909	990	ele2 first
98	9	989	998	ele2 first

Idea: Given 2 numbers ele1 & ele2

```
if( ele1+ele2 > ele2+ele1){ # ele1 comes first
    return true;
}
else{
    return false;
}
```

Note: When we compare strings we get dictnary order

to_string(i);

```
bool desc(int s1, int s2){
    string f = to_string(s1) + to_string(s2);
    string s = to_string(s2) + to_string(s1);
    if(f>s){ #s1 comes first
        return true;
    }
    else{
        return false;#s2 comes first
    }
}
```

s_1: 3 4 6
s_2: 3 4 7

$s_1 < s_2$ ✓

```
string sortDes(vector<int> &arr){

    sort(ar.begin(), ar.end(), desc);
    string ans="";
    for(int i=0; i<ar.size(); i++){
        ans += to_string(ar[i]);
    }
    return ans;
}
```

String concatenation in C++;

🔍 Time Complexity of Each

1. `s = s + "a";`

> ❌ **O(n)** (every time)

- `s + "a"` creates a **new string of length** `n + 1`
- Copies all `n` characters from `s`
- Appends `"a"`
- Assigns the new string back to `s`

🔷 So total cost = **O(n) copy + O(1) append**

🕒 **Time Complexity: O(n)** per operation
(`n` = current length of `s`)

2. `s += "a";`

> ✅ **Amortized O(1)** per operation

- Appends in-place, reusing capacity if available
- Only occasionally reallocates (like vector doubling)

So:

- Most of the time → **O(1)**
- When reallocating → **O(n)**

🧠 But over many appends (e.g., in loop), it's **amortized O(1)**

🚀 Summary Table:

Operation	Time Complexity	Notes
`s = s + "a"`	❌ O(n)	Creates temp + copies entire string
`s += "a"`	✅ Amortized O(1)	In-place append using existing memory