Name Kher Akshay M.

Standard Division Roll

Subject DSA & Placement

INDEX

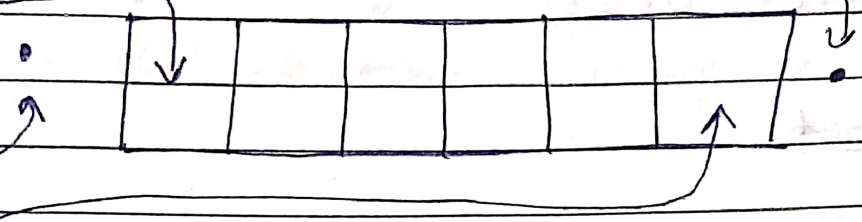
Serial No	Date	Title	Page No	Teacher's Signature/Remarks
1.		C++ STL	1-3	[take U forward]
2.		Time & space complexity	4	[LB video]
3.		Array & string	5	[LB sheet cpy.]
4.		Sorting	6-9	[GFG Aditya]
5.		Searching	10-19	[Aditya verma]
6.		Heap (Priority queue	20-27	[Aditya verma]
7.		Stack & queue	28-33	[Aditya verma]
8.		Hashing	34-37	[Hello world]
9.		Recursion & Backtracking	38-42	[Striver]
10.		Linkedlist	43-46	[Pareas salmi]

Date
Page 1

C++ STL

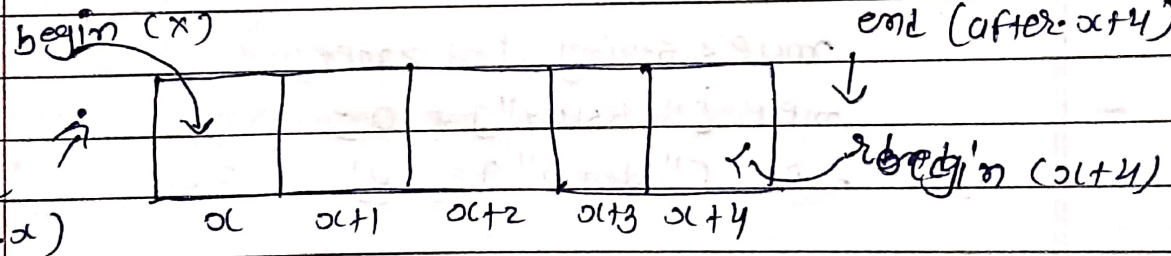
[take U Forward yt vide

- begin() → start on the array
- end() → end of the Array after the point



- rbegin() → end of the Array the point
- rend() → start of the array the point

end : right after the last element
rend : right before the start element
rbegin : Array of the last element
begin : Array of the start element



size : size of the array
front : Point first element
back : Point last element

★ set

→ set store unique element on Ascending order
O(log N)

☆ unorder set

→ element store in any order
→ every time random order store
→ unique O(1).

☆ multiset

→ You can add all type element in sorted order. Not unique (log N)

☆ map (container)

→ map only stores unique key maintains a linearly increasing order

```
map<string, int> mpp;
mpp["akshay"] = 10;
mpp["demo"] = 20;

for (auto it : mpp) {
    cout << it.first << " " << it.second << "\n";
}
```

☆ unorder map

→ does not stores in any order
→ O(1) in almost all cases & O(N) worst case
→ unorder-map not define a pair

☆ Stack & queue

→ stack means LIFO (Last in first out)
→ queue means FIFO (First in first out)
→ queue all operation O(1) but only pop is to time taken O(n)

☆ Priority queue

→ store all in sorted order & does all operation perform in (log N)
→ duplicated values are allow

☆ List

→ List are used to get element all side like front, back, top... etc...
→ It make doubly linked list

Date
Page 4

Time & Space Complexity

[LB yt video]

→ in here three type of complexity available
1) Big O notation — Upper Bound
2) ~~*~~ Theta Θ — Avg - case
3) Omega Ω — lower bound

- constant time - O(1)
- linear time - O(n)
- logarithmic time - O(log n)
- quadratic time - O(n²)
- cubic time - O(n³)

O(N!)
O(2^N)
O(N³)
O(N²)
O(N log N)
O(N)
O(log N)
O(1)

★ If stuck in TLE.

→ 10^8 operation rule → most of the modern machine perform 10^8 operations

≤ [10 ... 12] - O(n!)
≤ [15 ... 18] - O(2^n or n²)
≤ 100 - O(n⁴)
≤ 400 - O(n³)
≤ 2000 - O(n² log n)
≤ 10^4 - O(n²)
≤ 10^6 - O(n log n)
≤ 10^8 - O(n), O(log n)

☆ Arrays ☆

[Arrays - LB sheet
String - LB sheet]

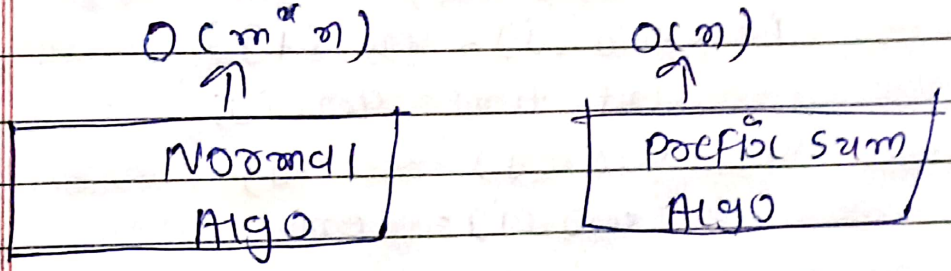
→ some important Algorithms in Arrays

1. Prefix sum Algo

→ when ever you can sum whole Arrays then you can use prefix sum Algo.

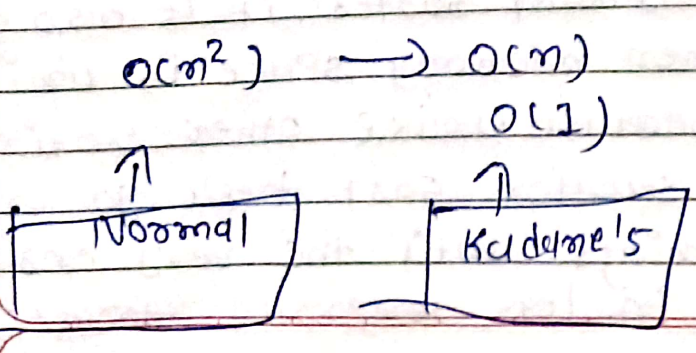
→ A normal Algorithms take (O m*n) time to perform m number of queries to find range sum on n size array

→ Prefix sum Algorithm take O(n) time




```
For (int i=1; i<n; i++)
    A[i] = A[i] + A[i-1];
```

(2) Kadane's Algorithm (maximum sum subArray)



☆ Sorting

[GFG Article]

Sorting Algorithms

1. selection sort
2. insertion sort
3. Bubble sort
4. merge sort
5. Quick sort
6. counting sort

selection Sort

```
for (int i=0; i<n-1; i++)
    for (int j=i+1; j<n; j++)
        if (arr[j] < arr[i])
            int temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
```

Time: $O(N^2)$
space: $O(1)$

why we use selection sort ?
selection sort can be good at checking if everything is already sorted. it is also good to use when memory space is limited. This is because unlike other sorting algorithms, selection sort does't go around swapping things until the very end, resulting in less temporary storage space

(2) Insertion Sort

```
void Insertion (int arr[], int n) {
    int key;

    for (int i=1; i<n; i++) {
        key = arr[i];
        j = i-1;

        while (j>=0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key
    }
}
```

Time : $O(N^2)$
Space : $O(1)$

→ Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big Array

(3) Bubble sort

```
void buble (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]);
}
```

Time : $O(N^2)$
Space : $O(1)$

Best case: $O(N)$, $O(1)$

→ Due to it's simplicity, bubble sort is often used to introduce the concept of a sorting algorithms

→ In computer graphics, it is popular for it's capability to detect a very small error (like a swap of just two elements) in almost - sorted arrays and fix it with just linear compile

(4) merge sort

```
void merge(int arr[], int start, int end)
{
    if(start < end){
        int mid = (start+end)/2;
        mergesort(arr, start, mid);
        mergesort(arr, mid+1, end);
        merge(arr, start, mid, end);
    }
}
```

time: O(n log n)
space: O(n)

→ it is based on the divide and conquer strategy merge sort continuously cuts down a list in to multiple sublists until each has only one item, then merges those sublists in to sorted list

(5) Quick Sort

```
void quick(int arr[], int low, int high){
    if(low < high){
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi-1);
        quicksort(arr, pi+1, high);
```

	Best	Average	Worst
Time:	$n \log n$	$n \log n$	$O(n^2)$

(6) Counting Sort

Time: $O(n+k)$

Space: $O(n+k)$

Date
Page 10

☆ Searching

① Linear Search

```
int Linear (int arr[], int n, int key)
{
    for(int i=0; i<n; i++){
        if(arr[i]==key)
            return i;
    }
    return -1;
}
```

Time: O(n)
Space: O(1)

→ Linear search is usually very simple to implement & is practical when the list has only a few elements or when performing a single search in an unordered list

→ when many value have to be searched in the same list, it often pays to pre-process the list in order to use a faster method

② Binary Search

[Ability version]

```
int Binary (int arr[], int n, int key){
    int start=0;
    int end=n;

    while (start<end){
        int mid=(start+end)/2;

    if(arr[mid]==key)
        return mid;
```

```
else if (arr[mid] > key)
        end = mid - 1;
    else
        start = mid + 1
return -1;
```

Best : O(1)
Ave : O(log n)
worst : O(log n)
space : O(1)

* Some questions on Binary search

(1) Binary search on Reverse sorted array

	0	1	2	3	4	5	6
arr[]:	20	17	15	14	13	12	8

search = 13

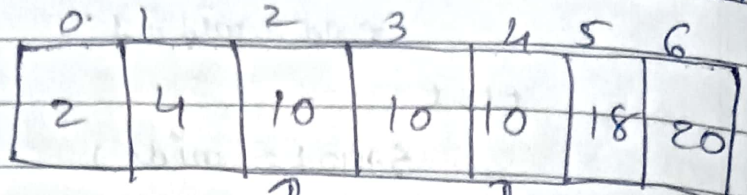
```
if (arr[mid] == key)
    return mid;
else if (arr[mid] < key)
    end = mid - 1;
else
    start = mid + 1;
```

(2) Order Not known search

→ You can check array are sorted accending or decending order

```
if (arr[i] < arr[i+1])
    accending
else
```

(3) First / last occurence of an element

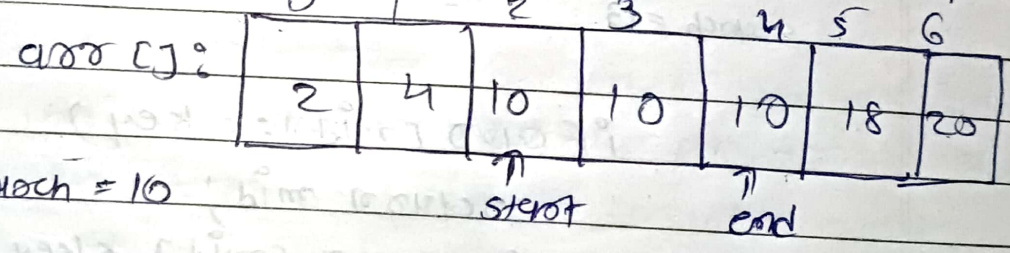


search = 10

First —
```
ans = mid;
end = mid - 1;
```

last
```
ans = mid;
start = mid + 1;
```

(4) Count of an element in a sorted array

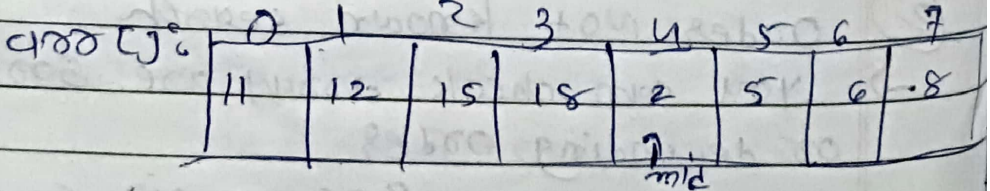


Search = 10

```
return (end - start) + 1
```

(4 - 2) + 1 = 2 + 1 = 3

(5) Number of Times a Sorted array is Rotated



→ Number of Times array rotated by depends on Index of minimum elements.

11 12 15 — sorted

18 2 5 6 8 — unsorted

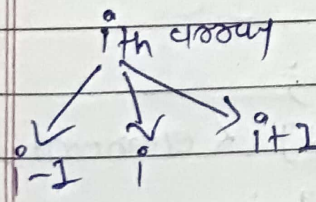
1. Find mid
2. Rotate & move

```
int prev = (mid+n-1) % n;
int next = (mid+1) % n;

if (nums[mid] < nums[prev] && nums[mid] < nums[next])
    return nums[mid];
else if (nums[mid] > nums[end])
    start = mid+1;
else
    end = mid-1;
```

(6) ~~nearest~~ ~~sorted~~ Array

arr[]:

5	10	30	20	40




```
if (element == arr[mid] && element == arr[mid-1]
        && element == arr[mid+1])
    return mid;
```

ceil 8

7.8

floor 7

(7) Find floor of an element in a sorted array

arr[] : | 1 | 2 | 3 | 4 | 8 | 10 | 10 | 12 |

floor = 5

floor = greatest element smaller then 5

output : 4

```
int res = -1;
if (arr[mid] < element)
    res = mid / arr[mid];   <- index / element
    start = mid+1;
else if (arr[mid] > element)
    end = mid-1;
```

(8) Find ceil of an element in a sorted array

output : 8

```
int res = -1;
if (arr[mid] > element)
    res = mid;
    end = mid-1;
else if (arr[mid] < element)
    start = mid+1;

return res;
```

(9) Next Alphabetical element

I/P. arr[]: | a | c | F | h | key = F

O/P :- h

```
char res = arr[start]
if(arr[mid] == key]
    start = mid+1;
else if (arr[mid] > key)
    res = key[mid];
    end = mid-1;
else
    start = mid+1;

return res;
```

(10) Find position of an element in an infinite [Sorted array]

	0	1	2	3	4	...
I/P arr[]:	1	2	3	4	5	.... ∞
	start	end				

key = 7

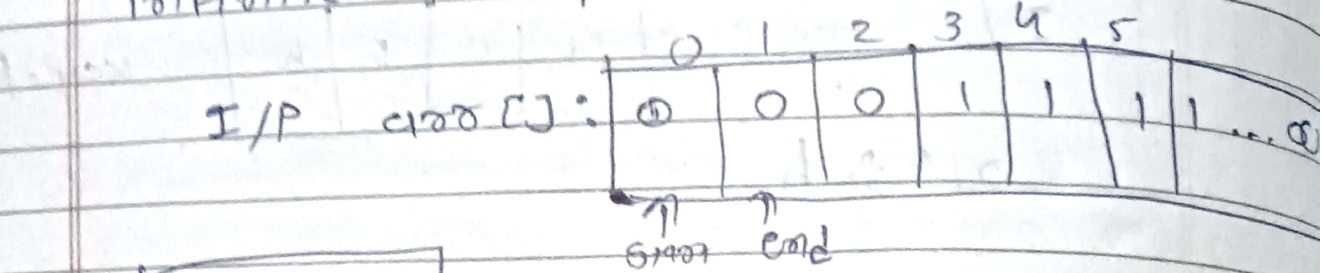
```
int start = 0;
int end = 1;

while(key > arr[end]){
    low = high;
    high = high * 2;
}
```

low = start
high = end

after BS (arr, start, end)

→ 0, 1

(11) Index of first 1 in a Binary sorted Infinite Array



O/P :- 3

⇒ Approach :- First occurrence + Infinite sorted

this question

```
while (key > arr[end]){
    end = mid + 1;
}
ans = mid;
end = mid - 1;
```

(12) minimum Difference element in sorted Arr

I/P arr[] : 4 6 10

key = 7

6

Output : 4-7 = 3
6-7 = 1
10-7 = 3

⇒ If key present then return key otherwise go to (Floor Approach)

⇒ Other Approach :-

```
while (start <= end) {
    int mid = start + (end - start)/2;
    if (key == arr[mid]
        return key;
    else if
        abs (arr [start - key]);
        abs ( arr [end - key]);
```

because of this type to stop integer overflow

(13) Binary search on Answer concept

If Sorted Array → BS

If Unsorted Array → BS (How ?) → see all next question,

(14) Peak element

	0	1	2	3	
I/P arr[] :	5	10	20	15	O/P :- 2

→ Peak means both of side element are smaller then current element

10 (20) 15 — this is peak element

```
int start = 0;
int end = size - 1;
```

```
while (start <= end) {
    int mid = start + (end - start)/2;

  if (mid > 0 && mid < n) {
      if (arr[mid] > arr[mid-1] &&
                arr[mid] > arr[mid+1])
          return mid;

    else if (arr[mid-1] > arr[mid])
          end = mid-1;

      else
          start = mid+1;
  }
  else if (mid == 0) {  <- first element
      if(arr[0] > arr[1]
          return arr[0];

      else
          return arr[1];
  }

  else if (mid == size-1) {
      if(arr[size-1] > arr[size-2]
          return size-1;
```

→ checks last element

```
      else
          return size-2;
  }
}
```

(15) Find maximum element in Bitonic Array

	0	1	2	3	4	5
I/P arr[]:	1	3	8	12	4	2

O/P :- 12

→ Bitonic Array means : monotonical increasing & decreasing

5 10 15 20

→ Approach :- same like previous question (peak element)

(16) Search An element in Bitonic Array

	0	1	2	3	4	5
I/P arr[]:	1	3	8	12	4	2

key = 4

O/P :- 4 (index of 4)

1 3 8 — increasing order

12 4 2 — Decreasing order

→ Approach :- First Find max element such like prev. question max = 12

→ after BS(arr, 0, index-1, asc);
BS(arr, index, size-1, des);

→ one BS return -1
→ one BS return Index & return it
→ if not found index then return -1

Date
Page 20

☆ Heap (priority queue)

- How to Identify heap question

1) In here two keyword find in heap question

kth smallest / largest / greater

- why we need to heap data structure ?

→ Because in this question given array after you can use merge sort (log n) complexity But we reduce by using heap sort (log k)

☆ Two Types of heap

1) max heap ⟶ (kth smallest)
2) min heap ⟶ (kth largest)

→ First think about sorting (log k). But in here question k are given so you can try to heap with (log k) complexity

☆ Observation

k + smallest / largest	→	smallest → max heap largest → min heap	→	n log k

(1) max heap (K th smallest / closest / lowest)

```
priority_queue <int> pq;

while (pq.empty() == false)
{
    cout << pq.top() << " ";
    pq.pop();
}
```

(2) min heap (largest / greatest / TOP th K)

```
priority_queue <int, vector<int>, greater <int>> pq;

while (pq.empty() == false)
{
    cout << pq.top() << " ";
    pq.pop();
}
```

(3) Kth smallest element

I/P arr[]:

7	10	4	3	20	15

k = 3

O/P: 7

1st Approach:
Sorting (n log n)

	0	1	2	3	4	5
sort	3	4	7	10	15	20

return arr[k-1]; return arr[k-1];

2nd Approach :- max heap O(n log k)

```
int findKth(vector<int>& nums, int k)
{
    int val;
priority_queue<int> pq;
int n = nums.size();

for(int i=0; i<n; i++)
{
  pq.push(nums[i]);
  if(pq.size() > k)
     pq.pop();
}
  return pq.top();

}
```

(4) Kth largest elements in array

I/P arr[] :

7	10	4	3	20	15

K = 3

O/P :- 10 20 15

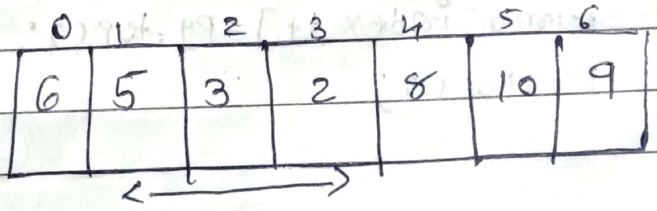
1st Approach :- Sorting O(n log n)

```
sort(arr, arr+n);
return arr[n-k];
```

2nd Approach :- min heap O(n log k)

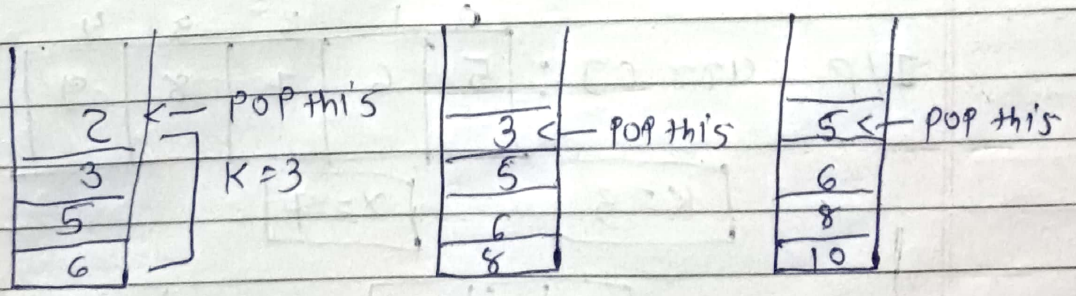
```
int FindKth (vector<int>& nums, int k)
{
    priority_queue<int, vector<int>, greater<int>> mp;
    int n = nums.size();

    for (int i=0; i<n; i++)
    {
        mp.push(nums[i]);
        if (mp.size() > k)
            mp.pop();
    }
    return mp.top();
}
```

(5) sort a k sorted array || sort nearly sorted array



K = 3

→ index zero value move & sort under k^{th} element means 6 are move to up to 3rd index



min heap Output :- 2 3 5 6 8 9 10

```
void sortKth(vector<int>& nums, int k) {

    priority_queue<int, vector<int>, greater<int>> pq;

    for(int i=0; i<=k; i++) {
        pq.push(nums[i]);
    }
    int index = 0;

techiedost code
    for(int i=k+1; i<nums.size(); i++)
    {
        nums[index++] = pq.top();
        pq.pop();
        pq.push(nums[i]);
    }
    while(!pq.empty())
    {
        nums[index++] = pq.top();
        pq.pop();
    }
}
```

// Time: O(n log k)
Space: O(k)

(6) k^{th} closest numbers

I/P arr[]:

0	1	2	3	4
5	6	7	8	9

K = 3 X = 7

O/P: 6 7 8

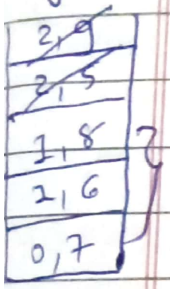
→ we need to here pair Because in stl c/s not only key store with. key & absolute different store

→ we use max heap Because closest element need

priority pairs code

```
void KthClosest (int arr[], int n, int x, int k)
{
    Priority_queue <Pair <int, int>> pq;
    For (int i=0; i<k; i++)
        pq.Push ({abs(arr[i]-x), i});

    For (int i=k; i<n; i++){
        int diff = abs(arr[i]-x);
    if (diff > pq.top().first)
        continue;

    pq.POP();
    pq.Push ({diff, i});
}

while (pq.empty() == False)
{
    cout<<arr[pq.top().second]<<" ";
    pq.POP();
}

}
```

for this



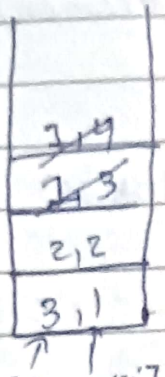
→ largest, greatest, top

(7) Top K^{th} Frequent Numbers (min heap)

	0	1	2	3	4	5	6
I/P arr[] :	1	1	1	3	2	2	4

K = 2

1 → 3 time present in array
3 → 1 time present in array
2 → 2 time present in array
4 → 1 time present in array



→ we need to top 2 most frequent Numbers

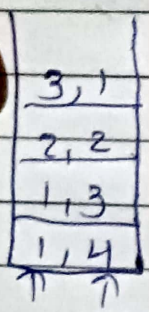
O/P : [1, 2]

1. Hashing (unordered_map)
2. After min heap

(8) Frequency Sort

	0	1	2	3	4	5	6
I/P arr[] :	1	1	1	3	2	2	4

O/P :- [1 1 1 2 2 3 4]



1st Approach :- compare function in STL (++)
2nd Approach :- Heap (max heap)

```
while (maxheap.size() > 0)
{
    int fre = maxheap.top().first;
    int ele = maxheap.top().second;
```

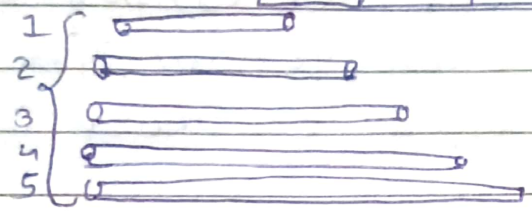
```
for (int i=1; i<=k; i++)
    cout << elem;
    maxheap.pop();
}
```

(9) k^{th} closest point To origin

(10) connect ropes

I/P :- arr[] :

0	1	2	3	4
1	2	3	4	5



O/P :-

(11) sum of element between k_1 smallest & k_2 smallest numbers

I/P arr[] :

1	3	12	5	15	11

$k_1 = 3$, $k_2 = 6$

sort →

1	3	5	11	12	15

3rd + 6th

11 + 12 = 23 return

Date
Page 28

☆ Stack

① How to identify stack question

→ if you have implemented Brute force ($O(n^2)$) and after you not decide ki how to optimize this then check this observation

j → 0 to i j++
j → i to 0 j--
j → i to n j++
j → n to i j--

→ when in Brute force 2nd (J) loop depends on i (first loop) then you can think about Stack and in optimize under O(n) time

☆ Some stack question

① Nearest greater to right || Next (largest ele. NGR

	0	1	2	3
I/P arr[]:	1	3	2	4

O/P	3	4	4	-1

1st Approach :- Brute force $O(n^2)$

```
For (int i=0; i<n-1; i++)
    For (int j= i+1; j<n; j++
```

→ in here 2nd loop (j) depend on 1st loop (i) so we think about stack data structure

2nd Approach :- stack Time : O(n) , space : O(n)

```
vector <int> v;  // creating a vector for store result
stack <int> s;

for(int i=n-1; i>=0; i--){
    if(s.size()==0)
        v.push_back(-1);
    else if(s.size()>0 && s.top()>arr[i])
        v.push_back(s.top());
    else if(s.size()>0 && s.top()<=arr[i])
    {
        while(s.size()>0 && s.top()<=arr[i])
            s.pop();
        if(s.size()==0)
            v.push_back(-1);
        else
            v.push_back(s.top());
    }
    s.push(arr[i]);
}
reverse(v.begin(),v.end());
return v;
```

(2) Nearest greater to left (NGL)

	0	1	2	3
I/P:	1	3	2	4

O/P:	-1	-1	3	-1

1st Approach :- $O(n^2)$

```
For(int i=0; i<n; i++
    For(int j=i-1; j>=0; j--)
```

→ we clearly see 2nd loop depend on 1st so we think about optimize steps for

2nd Approach :

→ left to right } same like pre. code
→ No reverse required

(3) Nearest smaller to left (NSL)

I/P : arr[] :

	0	1	2	3	4
	4	5	2	10	8

O/P :	-1	4	-1	2	2

1st Brute force : $O(n^2)$

```
For(int i=0; i<n; i++)
    For(int j=i-1; j>=0; j--)
```

2nd Approach : O(n) Stacks

1. left to right
2. No reverse required
3. greter than sign to lessthan && less to greter

(4) Nearest smaller to Right (NSR)

	0	1	2	3	4
I/P : arr[] :	4	5	2	10	8

O/P :	2	2	-1	8	-1

- 1st Brute Force Approach : $O(n^2)$

```
for (int i=0; i<n; i++
    for (int j=i+1; j<n; j++)
```

- 2nd Approach : Stack $O(n), O(n)$

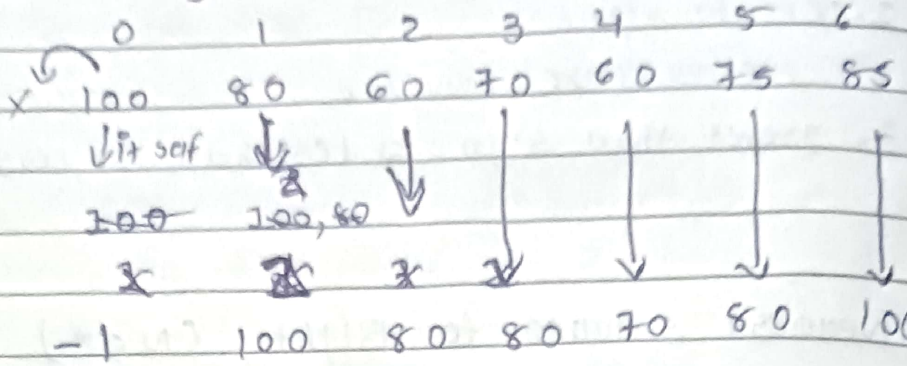
1. Right to left
2. smaller
3. vector Reverse

(5) Stock Span Problem

I/P arr[] :	100	80	60	70	60	75	8

O/P :	1	1	1	2	1	4	6

Approach :- you can think about nearest greter element to left



arr[i] index - nearest greter element index

1 | 1-0=1 | 2-1=1 | 3-1=2 | 4-3=1 | 5-1=4 | 6-0=6

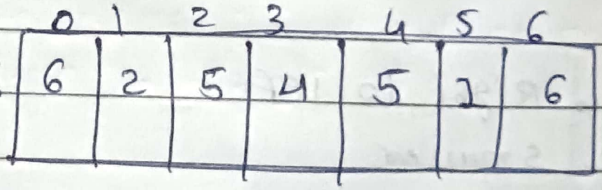
Actual output: { 1, 1, 1, 2, 1, 4, 6 }

stack <pair <int, int >> s;

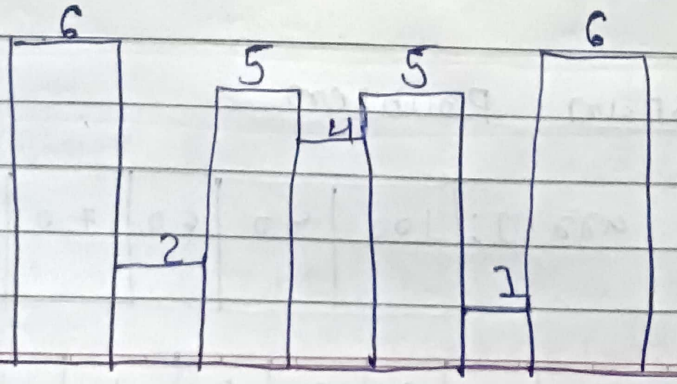
[store NGL] [store index]

Other code same like NGL

(6) maximum Area Histogram II (mAH)



O/P: 1 5 1 3 1 7 1



O/P: 12

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

Approach:-

NSR index	1	5	3	5	5	7	7
NSL index	-1	-1	1	1	3	-1	5

width = (NSR - NSL) - 1

arr	6	2	5	4	5	1	6
width :	1	5	1	3	1	7	1

Area of :	6	10	5	12	5	7	6
Veerd				↑			

max Area

(7) max Area Rectangle in Binary matrix

(8) Rain water Trapping

(9) minimum element in stack with extra space (min stacks)
O(n), O(n)

(10) minimum Element in stack with extra space (min stack) optimize space

Time: O(n)
Space: O(1)

☆ Hashing

(Hello world YT)

- hashing is a technique that is used to uniquely identify a specific object from a group of similar objects
- in hashing large key are converted into small keys by using hash functions

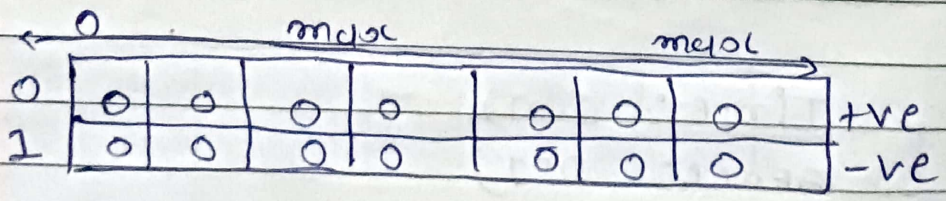
  eg:- WD53AFK32 ⟶ #32

- the value are then stored in a data structure it's called hash table
- the idea of hashing is to distribute entries (key/values pairs) uniformly across an array
- By using that key you can access the element in O(1) time.

☆ How to handle negative number in hashTable

→ The idea is to use a 2D array of size

```
hashTable[max+1][2]
```



- in general try to first think about Brute Force O(n) After think about O(1) using hashing

Code of Hashing Implementation

```
#define MAX 1000
bool hashTable[MAX+1][2];

bool search(int x){
    if(x>=0){
        if(hashTable[x][0]==1){
            return true;
        else
            return false;
    }else{
        x=abs(x);
        if(hashTable[x][1]==1)
            return true;
        else
            return false;
    }
}

void insert(int arr[], int n){
    for(int i=0; i<n; i++){
        if(arr[i]>=0)
            hashTable[a[i]][0]=1;
        else
            hashtable[abs(a[i])][1]=1;
    }
}
```

what is unordered_set

→ set → keys are stored in order fashion
→ unordered_set → keys are stored in unordered fashion

→ set → is internal implementation in Red Black tree
→ unordered_set → is internal implement in hashing

→ set → operation O(log n)
→ unorder_set → O(1) (insert, search, delete)

what is unordered_map

→ they store <key, value> pair other all are same unordered_set

some question

(1) Non Repeating Element

I/P arr[]: | -1 | 2 | -1 | 3 | 2 |

O/P :- 3

→ we stored here <key, value> pair Because in key we store arr[i] & in value store count of arr[i]

```
int first (int arr[], int n) {
    unordered_map<int, int> umap;
    For(int i=0; i<n; i++)
        umap[arr[i]]++;

    For (int i=0; i<n; i++) {
        int key = arr[i];
        auto temp = umap.find(key);
        if (temp -> second == 1)
            return key;
    }
```

(2) First Repeating Element

```
if (temp -> second > 1)
    return i+1;
```

(3) Intersection of two Arrays (unordered_set)

(4) key pair hashing problems (unordered_map)

(5) subarray with sum 0

(6) winner of an election

(7) pairs with positive negative values

(8) Relative sort Array. | sort an array according to the other

(9) group Anagrams | print Anagrams together

(10) sort Array by Increasing Frequency

(11) custom sort string

Date
Page 38

★ Recursion & Backtracking

[striver]

- **Recursion** :- when a function call itself until specify condition is met.

Basic example of Recursion

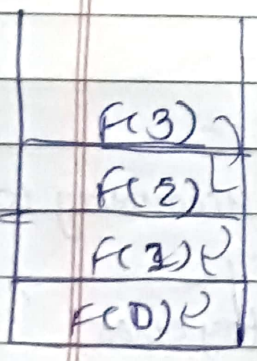
```
int count = 0;
void Total(){
    if(count == 3)
        return;
    cout << count << " ";
    count++;
    Total();
}
```

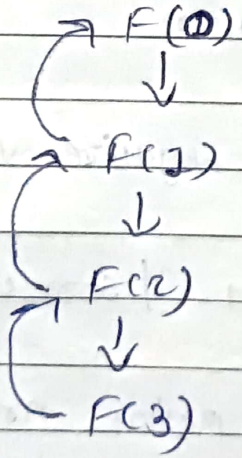
← Base condition to stop program

O/P : 0 1 2

Time : O(N)
space : O(N)

- **Recursion Tree**





← Base condition true

Some Basic Question on Recursion

[Try Itself]

1) Print Name 5 times
2) Print Linearly From 1 to N
3) Print Linearly From N to 1
4) Print Linearly From 1 to N (using Backtrack)
5) Print Linearly From N to 1 (using Backtrack)

6) Sum of First N number using
   1) Parameterise way
   2) Functional way

7) Factorial of First N Number
8) Fibonacci of First N Number

9) Reverse array using Recursion
   1) using two variable
   2) using single variable

10) Check if a string is Palindrome or not

★ Backtracking

(1) Subsets of Array

I/P :- arr[3] = {1, 2, 3}

O/P :- { {}, {1}, {2}, {3}, {1,2}, {2,3}, {1,3}, {1,2,3} }

$2^n = 2^3 = 8$

Approach :

```
class Solution {
  private:
    void solve(vector<int> nums, vector<int> output,
               int index, vector<vector<int>> &
  {
    if(index >= nums.size()){
      ans.push_back(output);
      return;
    }
    solve(nums, output, index+1, ans);

    int element = nums[index];
    output.push_back(element);
    solve(nums, output, index+1, ans);
  }
  public: vector<vector<int>> subsets(vector<int>&
    vector<vector<int>> ans;
    vector<int> output;
    int index=0;
    solve(nums, output, index, ans);
    return ans;
```

Base case →

(2) Phone Keypad Problem using Backtracking [Leetcode]

```
class Solution {
  private:

void solve (string digit, string output, int index, vector
                                <string> &ans, string mapping[])
{
  if (index >= digit.length()){
      ans.push_back(output);
      return;
  }
  int number = digit[index] - '0';
  string value = mapping[number];

  for(int i=0; i<value.length(); i++){
      output.push_back(value[i]);
      solve(digit, output, index+1, ans, mapping);
Backtrack ---> output.pop_back();
  }

public: vector<string> letter(string digits){
      vector<string> ans;
      if(digits.length()==0)
          return ans;

      string output, int index=0;

  string mapping[10] = { };
  solve(digits, output, index, ans, mapping);
  return ans;
}
```

- Some standard leetcode Backtracking que.

1) letter combinations of a phone no.
2) Generate Parentheses
3) Permutations
4) Permutations II
5) Combinations && combination sum && combination sum II
6) Subsets
7) Subsets II
8) N-queens [Imp]
9) Sudoku solver
10) word search
11) Palindrom Partitioning

★ LinkedList

→ Linked list is a linear data structure.

- Some Practical Application of LinkedList
  1) image viewer
  2) web browser
  3) music player

- How to create a LinkedList

→ You need to Data, & next then you can make a Linked list

(1) How to create & add Node in Linked list

```
Linked_List *add(int data, Linked_List *head)
{
    //case :- 1
    if(head == NULL){
        Linked_List *temp = new Linked_List(data);
        head = temp;
    }
    else{
        Linked_List *temp = head;
        while(temp->next != NULL)
            temp = temp->next;

        temp->next = new Linked_List(data);
    }
    return head;
}
```

② Print Linked List

```
void LinkedListPrint(LinkedList *head)
{
    auto temp=head;
    while(temp!=NULL){
        cout<<temp->data<<"->";
        temp=temp->next;
    } cout<<endl;
}
```

③ Find middle element in LinkedList

```
int middle_element(LinkedList *head)
{
    auto sp=head, fp=head;
    while(fp!=NULL && fp->next!=NULL)
    {
        sp=sp->next;
        fp=fp->next->next;
    }
    return sp->data;
}
```

④ delete middle element in linked list

```
LinkedList * deleteLink(LinkedList *head){
    auto sp=head, fp=head;
    auto pre=sp;
```

```
while (FP != NULL && FP->next != NULL) {
pre = SP;    SP = SP->next;
             FP = FP->next->next
}
    pre->next = SP->next;
    delete SP;
    return head;
}
```

(5) delete linked list

```
linked-list *delete_linked_list (auto head)
{
    auto pre = head;
    head = head->next;

    if (head == NULL)
        delete pre;
    while (head != NULL ){
        delete pre;
        pre = head;
        head = head->next;
    }
    return head;
}
```

Some Imp question on Linked List

1) Recursively Print Reverse of Linked List
2) Reverse Linked List
3) Palindrome Linked List
4) Delete Duplicate from Linked List (sorted & unsorted)
5) Detect Loop in LinkedList
6) Remove Loop
7) Intersection point in a Linked List
8) Odd Even Linked List
9) merge sort on Linked List