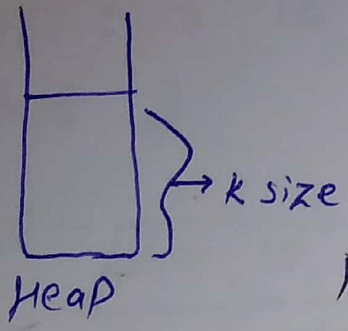
Heap and its identification

i) k
ii) Smallest / largest → Heap



Heap

- min Heap (k + largest)
  - at top min element present
- max Heap (k + smallest)
  - at top max element present

time Complexity reduce from $n \log n$ to $n \log k$.

Problem :- find k^{th} smallest Element

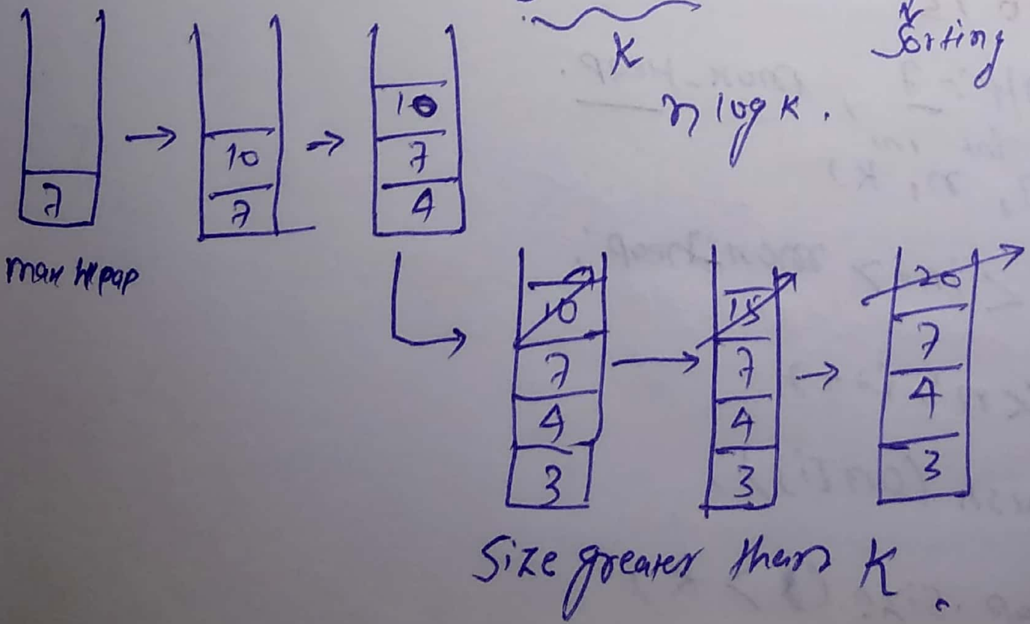
arr[] :

7	10	4	3	20	15

K = 3 output :- 7.

$\underbrace{3 \quad 4 \quad 7}_{k}$ $\boxed{10 \quad 15 \quad 20}$ → Sorting not used for this

$n \log k$.



return Heap top ans.

Size greater than K.

Code :-

Max Heap

```
priority_queue <int> max_Heap;
```

Min Heap

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

for pair :-

```
priority_queue < pair<int,int>, vector < pair<int,int> >,
          greater < pair<int, int > > min_heap;
```

or

```
type define  pair< int, int>  p ;
priority_queue < p  vector <p> ; greater<p> > min_heap.
```

kth Smallest Element

arr : 7 10 4 3 20 15

k = 3   o/p :- 7 , max_Heap.

```
int solve (int arr[], int n, int k)
{ priority_queue < int > max_heap;
  for (int i=0; i<n; i++)
  { max_heap.push (arr[i]);
    if (max_heap.size () > k)
        max_heap.pop ();
  } return max_heap.top ();
}
```

Return K^th largest element in array

arr[] : 7 10 4 3 20 15

K : 3

20 15 10 7 4 3

O/P :- $\boxed{20, 15, 10}$

```
int
void Solve (arr[], int n, int k)
{ priority_queue < int, vector<int>, greater <int> >
                                        min_heap;

  for( int i=0; i<n ; i++)
  { min_heap. Push (arr[i]) ;
    if (min_heap.size() > k)
          min_heap. pop();

  }
  while ( min_heap.size())
  { Cout << min_heap. top() << ' ' ;
        min_heap. pop();
  }
}
```

Sort a K-Sorted Array / Sort Nearly Sorted Array

arr[] :- 6 5 3 2 8 10 9

K = 3

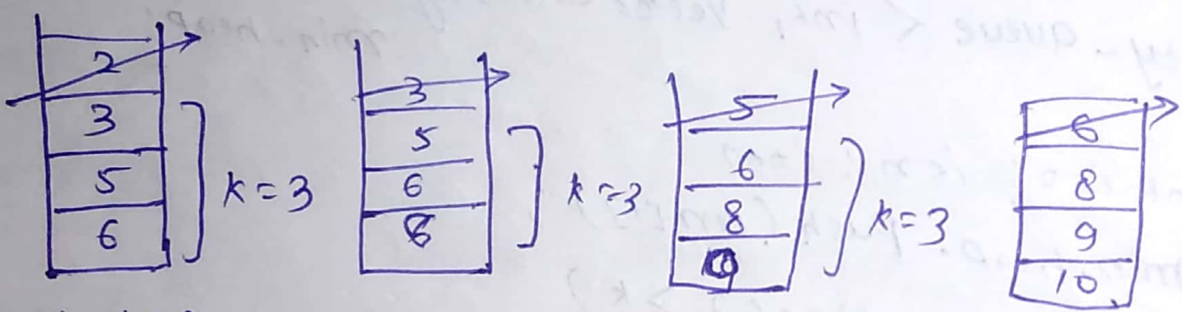
3

6 5 3 2 8 10 9

1 3 1 1

2 3 5 6 8 9 10

K = 3



min_heap

2 3 5 6 8 9 10

```
Void solve { int arr[], int n, int k)
{  Priority_queue <int, vector <int>, greater <int>>
                                          min_heap;

   for (int i=0 ; i<n ; i++)
   {  min_heap. Push ( arr[i]);
      if (min_heap. size() >k)
         { Cout << min_heap. top()<<" ";
           min_heap. pop() ;
         }
   }
 while (min_heap. size() )

   {  Cout << min_heap. top() ;
      min_heap. pop() ;
   } }
```

K Closest Number

input :- arr[] : 5 6 7 8 9

K = 3, x = 7

Output :-

5	6	7	8	9
7	7	7	7	7
2	1	0	1	2

↳ key

```
priority_queue < pair < int, int >> max_heap.
{ for (int i=0; i<n; i++)
   { max_heap.push ({ abs(x-arr[i]), arr[i] });
     if (max_heap.size() > k)
          max_heap.pop();

   }
 while (max_heap.size())
   { Cout << max_heap.top().second << ' ';
     max_heap.pop();

   }

}
```

Top K frequent Numbers

arr[]: 1 1 1 3 2 2 4

K = 2

O/p :- 1, 2

1 → 3
2 → 2
3 → 1
4 → 1

Largest
Greatest
top
↓
min-heap

Smallest
lowest
Closest
↓
max-heap

```
void Solve (int arr[], int n, int K)
{   unordered_map <int,int> mp;
    for (int i=0; i<n; i++)
        mp[arr[i]]++;
    priority_queue < pair<int,int>, vector<int,int>,
                       greater<int, int >> min_heap;
    for (auto i = mp.begin(); i != mp.end(); i++)
    {   min_heap.push({i->second, i->first});
        if (min_heap.size() > K)
            min_heap.pop();
    }
    while (min_heap.size())
    {   cout << min_heap.top().second << ' ';
        min_heap.pop();
}   }
```

Frequency Sort

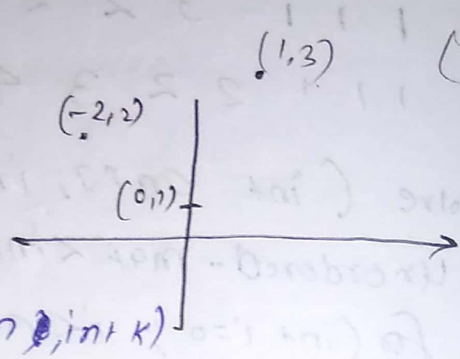
arr[] :- 1 1 1 3 2 2 4

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

```
Void Solve ( int arr[], int n)
{   unordered_map <int ,int> mp;
    for (int i=0 ; i<n ; i++)
        mp[ arr[i]] ++;
    priority_queue < pair <int , int >> max_heap ;
    for (auto i = mp.begin () ; i != mp.end() ; i++)
      { max_heap.push ({i->second, i->first});
      }
    While ( max_heap.size ())
    { int n = max_heap.top().first ;
      int m = max_heap.top().second ;
      while (n--)
       { Cout << m << " ";
       }
    }
 Cout << endl ;
}
```

K-Closest points to Origin

$$
arr[\,] = \begin{array}{c}
\begin{array}{cc} x & y \end{array} \\
\begin{bmatrix} 1 & 3 \\ -2 & 2 \\ 5 & 8 \\ 0 & 1 \end{bmatrix}
\end{array}
\begin{array}{c} \\ 10 \\ 8 \\ 89 \\ 1 \end{array}
\quad k=2
$$



O/P :- (0,1), (-2,2)

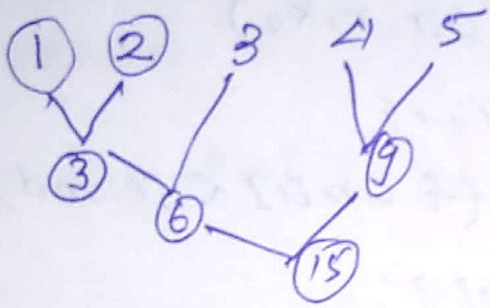
```
void solve ( int arr[][2], int n, int k)
{   priority_queue < pair< int, pair<int, int> > > max_heap;
for(int i=0 ; i<n ; i++)
  { max_heap.push ( { arr[i][0]*arr[i][0]+ arr[i][1]*arr[i][1]
                      { arr[i][0], arr[i][1] } } );
   if (max_heap.size() > k)
      { max_heap.pop();
      }
  }
while (max_heap.size())
{  cout <<'('<< max_heap.top().second.first << ', '
        << max_heap.top().second.second <<')'<<' ';
   max_heap.pop();

  }
}
```

Connect Ropes to minimise the Cost

arr[]: 1 2 3 4 5

Cost is if we connect 3 and 4 then
cost = 7.
if we connect 4,5 then cost = 9.



Cost = 3+6+9+15
= 33 ans.

heap,

[i][j]

```
int solve( int arr[], int n)
{ priority_queue <int, vector<int>, greater<int>>
                                        min-heap ;
  int sum=0;
for (int i=0; i<n; i++)
     min-heap.push( arr[i]);
while (min-heap.size()>=2)
 { int first = min-heap.top();
   min-heap.pop();
   int second = min-heap.top();
     min-heap.pop();
   sum += (first +second);
   min-heap.push( first + second);

 }
return sum;
}
```

Sum of elements

arr[] : 1 3 12 5 15 11

$K_1^{th} = 3$ Find sum between K_1^{th} and K_2^{th}.

$K_2^{th} = 6$

Smallest

```
int sum=0;
int first = solve (arr, n, k1)
int second = solve (arr, n, k2)

for (int i=0; i<n; i++)
    if ( arr[i] > first && arr[i] < second)
        sum += arr[i];

return sum;
```

Minimum difference Element in sorted array

arr[] :- 1 3 8 10 12 15

key :- 12

11 9 4 2 0 3

key :- 11

10 8 3 1 1 4

```
int binarySearch (int arr[], int n, int k)
{ int low=0;
  int high = n-1; int mid;
  while (low <= high)
  { mid = (low+high)/2;
    if (arr[mid] == k)
        return k;
```

Else if (arr[mid] > k)
    high = mid - 1;
Else low = mid + 1;

}
if (abs (k - arr[low]) > abs (k - arr[high]))
    return arr[high];
Else
    return arr[low];

}

Concept use if element not meet the low and high point to the neighbours of ~~element~~ key.

Ex:- 1 3 8 10 12 15

key = 11

they after while loop

1 3 8 10 (11) 12 15

high low

return (arr[low] => 12);

longest increasing subsequence

3 4 -1 0 6 2 3 = 4

2 5 1 8 3 = 3

```
int t[n+1]; int j; t[0]=0;
for (int i=0; i<n; i++)
      t[i+1]=1;
for( int i=1; i<n; i++)
{  j=0;
     for( ; j<i; j++)
       { if ( arr[i] >= arr[j] )
              t[i+1] = max( t[i+1], t[j+1]+1);
       }
}
return ~~t~~ max_element( t+1, t+n+1);
}
```

arr[i] = key;
}

Heap sort   O(n log n)

Heap sort, merge sort, Selection sort independent of data present in array.

Heap] → Average = Best = worst = O(n log n)
merge]

Selection sort] → Average = Best = worst = (O(n²))
Pick minm element take to prefix.

Stable

Bubble sort] → Average / worst = O(n²)
insertion sort] → Best O(n)

Quick sort] → Average / best = O(n logn)
worst = O(n²)

```
Void heapify (int arr[], int n, int i)
{   int largest = i;
    int l = 2*i+1; int r = 2*i+2;
    if (l<n && arr[l] > arr[largest])
            largest = l;
    if (r<n && arr[r] > arr[largest])
            largest = r;
    if (largest != i)
        {  swap ( arr[i], arr[largest])
           heapify ( arr[], n, largest);
        }
}
```

Page No.
Date :

```
Void heapsort ( int arr[], int n)
{
    for( int i = (n-1-1)/2 ; i>=0 ; i--)
        heapify ( arr, n, i);

    for( int i= n-1; i>0; i--)
    {
        swap( arr[0], arr[i]);
        heapify ( arr, i, 0);
    }                               O(n)
}
```

All time complexity
$O(n \log n)$

```
int main ()
{   heapsort (arr, n);
}
```