→ Priority queue of pairs :-

- max heap on basis of first (base/default).-

```
typedef pair<int,int> pd;
int main(){
    priority queue<pd> pq;

    pq.push({3,23});
    pq.push({10,23});
    pq.push({1,103});
    pq.push({5,13});

    while(!pq.empty()){
        cout<<pq.top().first<<" "
            <<pq.top().second<<endl;
        pq.pop();
    }
    return 0;
}
```

Output:-

```
10  2
5   1
3   2
1   10
```

• min heap on basis of first :-

Use syntax of min heap.

```
typedef pair<int,int> pd;

int main(){
    priority_queue<pd, vector<pd>, greater<pd>> pq;
    pq.push({3,2});
    pq.push({10,2});
    pq.push({1,10});
    pq.push({5,1});

    while(!pq.empty()){
        cout<<pq.top().first<<" "
            <<pq.top().second<<endl;
            pq.pop();
    }
    return 0;
}
```

Output:-

```
1  10
3  2
5  1
10 2
```

• max heap on basis of second :-

Here, idea is to use mycomp structure with operator overloading operator to order pairs by it's second element in heap.

```
typedef pair<int,int> P;

Struct mycomp{

    bool operator()(pair<int,int> const& a,
      pair<int,int> const& b)
    {
        return a.second < b.second;
    }
};

int main(){
  priority_queue<P, vector<P>, mycomp> pq;

  pq.push({3,2});
  pq.push({10,2});
  pq.push({1,10});
  pq.push({5,1});
```

```
while(!pq.empty()){
    cout<<pq.top().first<<" "
        <<pq.top().second<<endl;
        pq.pop();
}
return 0;
}
```

Output:-

```
1 10
10 2
3 2
5 1
```

- min heap on basis of second :-

We will use same idea as previous one but will modify comparision structure and overloading method.

```
typedef pair<int,int> p;

struct myComp{
    bool operator()(pair<int,int> const& a,
        pair<int,int> const &b)
    {
        return a.second > b.second;
    }
};
```

```
int main(){
    priority_queue<P, vector<P>, myCmp> pq;
    pq.push({3,2});
    pq.push({10,2});
    pq.push({1,10});
    pq.push({5,1});

    while(!pq.empty()){
        cout<<pq.top().first<<" "
            <<pq.top().second<<endl;
    }
    return 0;
}
```

Output:-

```
5 1
3 2
10 2
1 10
```

→ priority queue of structure :-

we will make priority_queue of structure and make priority_queue on basis of it's any element.

```
Struct person{
    int age;
    float ht;
    person(int a, float h)
    {
        age = a;
        ht = h;
    }
};

Struct myComp{
    bool operator()(person const& p1, person const& p2)
    {
    return p1.ht < p2.ht;
    }
};

int main(){
    priority_queue<person, vector<person>, myComp> pq;
    pq.push({18, 160});
    pq.push({16, 165});
    pq.push({20, 200});
```

```
while(!pq.empty()){
    cout<<pq.top().age<<" "
        <<pq.top().ht<<endl;
    pq.pop();
}
return 0;
}
```

Output:-

```
20  200
16  165
18  160
```

ht is max heap.

→ Sorting array using priority_queue:-

In daily use we must use sort() function to sort any array, but here for exercise we will use priority_queue method-

```
void sortarr(int arr[], int n){
    priority_queue<int> pq(arr, arr+n);
    for(int i=n-1; i>=0; i--){
        arr[i] = pq.top();
    }    pq.pop();
}
```

```
int main()
{
    int arr = { 2, 7, 3 5, 12 };
    sortarr
    int n = sizeof(arr)/sizeof(arr[0]);
    sortarr(arr, n);
    return 0;
}
```

→ K largest element in an array :-

In this problem we have given an array and we have print first 'k' largest element.

Ex:-

arr[] = { 10, 5, 30, 2, 13, 9, 1 };

output:- 10, 13, 30

NOTE:- order of output is not considered here, it may change.

Method 1:-

In this method it is naive method in which we first sort the array and then print 'k' element of that array.

T.C. will be O(k + n log n).

O(n log n) for sorting the array
O(k) for printing 'k' elements.

code-

```
void printk(int arr[], int n, int k){
    sort(arr, arr+n);
    for(int i=n-k;
    sort(arr, arr+n, greater<int>);
    for(int i=0; i<k; i++)
        cout<<arr[i]<<" ";
}
```

Method 2:- Using max heap-

In this approach we will make max_heap using priority_queue of given array. and print first 'k' elements of it.

T.C. will be O(n + k log n)

O(n) for make max_heap using pq.

O(k log n) as k times popping

out of pq. and printing that element.

Code -

```
void printK(int arr[], int n, int k){
    priority_queue<int> pq(arr, arr+n);
    for(int i=0; i<k; i++){
        cout<<pq.top()<<" ";
        pq.pop();
    }
}
```

Method 3:- Using min heap -

It is more efficient than max heap.

In this approach we will follow this algo. -

- make min_heap of first 'k' element of array.
- Compair element from 'i=k' to 'i=n-1' to top of pq.

if arr[i] > pq.top() . remove top()
and insert arr[i] in pq.

else leave pq as it is.

- After loop ends 'k' element in pq. will be largest 'k' element of array.

NOTE:- Beauty of this approach is that, in this approach at any time pq gives 'k' largest element at ~~that point~~ any given point in array.

Here, we have used min_heap as we have to find 'k' largest element if we have to find 'k' smallest element, we can use max_heap with similar algorithm.

T.C of this approach is

$$O(K + (n-K)\log K)$$

O(K) due to making min_heap using pq.

$O((n-k)\log k)$ due to loop on pq.

Code -

```
void printK(int arr[], int n, int k){
    priority_queue<int, vector<int>, greater<int>> pq(arr, arr+k);
    for(int i=k; i<n; i++){
        if(arr[i] > pq.top()){
            pq.pop();
            pq.push(arr[i]);
        }
    }
    while(pq.empty == false){
        cout << pq.top() << " ";
        pq.pop();
    }
}
```

→ Purchasing maximum items:-

We have given an array with values of items, and a variable containing total money we have. We have to find the max. no of items we can buy using this money.

ex-

arr[] = {1, 30, 7, 5, 12, 39};
money = 20

result = 3 (1, 5 and 7)

method 1:-

In this method we sort the array and compare money we have to the ith element of array.

if (arr[i] < money) we can buy that item ⇒ result ++ and
money -= arr[i];

else,
return the result.

Code -

```
int purchasemax(int arr[], int n, int m){
    sort(arr, arr+n);
    int result=0;
    for(int i=0; i<n; i++){
        if(arr[i]<=m){
            result++;
            m=m-arr[i];
        }
        else
            break;
    }
    return result;
}
```

T.C. will be $O(n + n \log n)$ but we don't need extra space.

method 2:-

In this method we use min-heap, we make min-heap out of array and compare pq.top() with money we have.

```
if(money > pq.top())
    money -= pq.top();
    pq.pop();
    res++;
```

T.C. of this app. is

$O(n + n \log n)$

As, you can see T.C. looks similar but this T.C. occurs very rarely in min-heap, generally it is

$O(n + res * \log n)$ and

$res << n$ in many cases.

Code -

```
int purchaseMax(int arr[], int n, int m){
    priority_queue<int, vector<int>, greater<int>>
        pq(arr, arr+n);

    int res=0;
    while(m>=0 && pq.empty()==false &&
          pq.top()<=m)
    {
        m-=pq.top();
        pq.pop();
        res++;
    }
    return res;
}
```

→ Merge K sorted array -

We have given 'k' sorted array each of size 'n' in 2-D array form.

arr[k][n];

We have to print the sorted merge out.

ex -

K=3, n=4

arr[][] = {{1, 3, 5, 7},
{2, 4, 6, 8},
{0, 9, 10, 11}}

output = 0 1 2 3 4 5 6 7 8 9 10 11

Approach 1 -

It is a naive approach we store all input sorted array in one big output array one by one and then sort that output array.

but T.C and S.C will be high.

approach 2:-

We will use merge sort approach. Ie. merge those k array by first deviding them in half.

When devide a given 'k' array such that,

if only one array is left, return it.

if two array are left merge them

if more than two array are left call the devide function recursively one each half.

Code -

```
Void merge_array (int arr1[], int arr2[], int n,
                  int m, int out[]){
  int i=0, j=0, k=0;
  while(i<n && j<m)
  {
    if(arr1[i] <= arr2[j])
      out[k++] = arr1[i++];
    else
      out[k++] = arr2[j++];
  }
```

while(i<n1)
out[k++] = arr1[i++];

while(j<n2)
out[k++] = arr2[j++];

}
}

Void merge_array(int arr[][n], int i, int j,
int output[]){

if(i==j){ //one array left
for(int p=0; p<n; p++)
output[p] = arr[i][p];
return;
}

if(j-1==1){ //two arrays left

merge_array(arr[i], arr[j], n, n, output)
return;
}

int mid = (i+j)/2

int out1[n*(mid-i+1)], out2[n*(j-mid)
//two output array to store output array fr
//merge_array function.

```
merge_k_array(arr, i, mid, out1);
merge_k_array(arr, mid+1, j, out2);

merge_array(out1, out2, n*(mid-i+1), n*(j-mid),
            Output);
}

int main(){
  int arr[][n]={{2, 6, 12, 34},
                {1, 9, 20, 1000},
                {23, 34, 90, 2000}};
  int k=sizeof(arr)/sizeof(arr[0]);
  int output[n*k];
  merge_k_array(arr, 0, k-1, output);
}
```

Approach 3:- It is best approach to merge k sorted array. We use min-heap priority queue to merge k-sorted array.

We first make a priority queue of pairs.

typedef pair<int, pair<int, int>> ppi;

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

We do so, because we want to store not only value but also in which array that value is and at what index.

So.

Let

ppi curr = pq.top(); then

curr.first → value

curr.second.first → in which array that value is form.

curr.second.second → index of element in array.

now,

- Create an output array.
- Creat a min-heap of size k and insert 1st element in all the array into heap.

- Repeat following steps while priority queue is not empty.
  - Remove top/minimum element from heap and store it in output array.
  - insert next element from the array from which the element is extracted. If the array doesn't have any more elements, then do nothing.

Code -

```
#include<iostream>
#include<utility>
#include<queue>
#include<vector>

typedef pair<int, pair<int,int>> ppi;
```

```
vector<int> mergeKArray(vector<vector<int>>
                        arr){

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

for(int i=0; i<arr.size(); i++){
pq.push({arr[i][0], {i, 0}});
}

while(pq.empty() == false){
    ppi cur = pq.top();
    pq.pop();
    output.push_back(cur.first);
    int i = cur.second.first;
    int j = cur.second.second;
if(j+1 < arr[i].size())
    pq.push({arr[i][j+1], {i, j+1}});
}

return output;
}
```

```
int main(){
    vector<vector<int>> arr{ {2, 6, 12},
                             {1, 9, 30},
                             {23, 34, 1000, 11113} };

    Vector<int> Output = merge_k_array(arr);

    for (auto x : output)
        cout<<x<<" ";

    return 0;
}
```