Heap

Lesson-4

Maximum distinct elements after removing k elements

Given an array arr[] containing n elements. The problem is to find the maximum number of distinct elements (non-repeating) after removing k elements from the array.

Note: 1 <= k <= n.

Examples:

Input : arr[] = {5, 7, 5, 5, 1, 2, 2}, k = 3
Output : 4
Remove 2 occurrences of element 5 and
1 occurrence of element 2.

Input : arr[] = {1, 2, 3, 4, 5, 6, 7}, k = 5
Output : 2

Input : arr[] = {1, 2, 2, 2}, k = 1
Output : 1

Approach: Following are the steps:

1. Make a multi set from the given array.

2. During making this multiset check if the current element is present or not in the multiset, if it is already present then simply reduce the k value and do not insert in the multiset.

3. If k becomes 0 then simply just put values in the multiset.

4. After traversing the whole given array,

a) if k is not equal to zero then it means the multiset is consist of only unique elements and we have to remove any of the k elements from the multiset to make k=0, so in this case the answer will be the size of multiset minus k value at that time.

b) if k is equal to zero then it means there may be duplicate values present in the multiset so put all the values in a set and the size of this set will be the number of distinct elements after removing k elements

```
int maxDistinctNum(int a[], int n, int k)
{
 int i;
 multiset<int> s;
 // making multiset from given array
     for(i=0;i<n;i++){
        if(s.find(a[i])==s.end()||k==0)
        s.insert(a[i]);
        else
          {
             k--;
          }
       }

       if(k!=0)
       return s.size()-k;
       else{
            set<int> st;
            for(auto it:s){
                st.insert(it);
            }
            return st.size();
        }
}
```

Time Complexity: O(k*logd), where **d** is the number of distinct elements in the given array.

Maximum difference between two subsets of m elements

Given an array of n integers and a number m, find the maximum possible difference between two sets of m elements chosen from a given array.

Examples:

```
Input : arr[] = 1 2 3 4 5
            m = 4
Output : 4
The maximum four elements are 2, 3,
4 and 5. The minimum four elements are
1, 2, 3 and 4. The difference between
two sums is (2 + 3 + 4 + 5) - (1 + 2
+ 3 + 4) = 4

Input : arr[] = 5 8 11 40 15
           m = 2
Output : 42
The difference is (40 + 15) - (5  + 8)
```

Approach:
The idea is to first sort the array, then find the sum of first m elements and sum of last m elements. Finally return the difference between two sums.

```
int find_difference(int arr[], int n, int m)
{
    int max = 0, min = 0;

    // sort array
    sort(arr, arr + n);

    for (int i = 0, j = n - 1;
         i < m; i++, j--) {
        min += arr[i];
        max += arr[j];
    }

    return (max - min);
}
```

Median in a stream of integers (running integers)

Given that integers are being read from a data stream. Find the median of all the elements read so far starting from the first integer till the last integer. This is also called the Median of Running Integers. The data stream can be any source of data, for example, a file, an array of integers, input stream etc.

Input: 5 10 15
Output: 5, 7.5, 10
Explanation: Given the input stream as an array of integers [5,10,15]. Read integers one by one and print the median correspondingly. So, after reading first element 5,median is 5. After reading 10,median is 7.5 After reading 15 ,median is 10.
Input: 1, 2, 3, 4
Output: 1, 1.5, 2, 2.5
Explanation: Given the input stream as an array of integers [1, 2, 3, 4]. Read integers one by one and print the median correspondingly. So, after reading first element 1,median is 1. After reading 2,median is 1.5 After reading 3 ,median is 2.After reading 4 ,median is 2.5.

Method 1: Insertion Sort

If we can sort the data as it appears, we can easily locate the median element. *Insertion Sort* is one such online algorithm that sorts the data appeared so far. At any instance of sorting, say after sorting *i*-th elements, the first *i* elements of the array are sorted. The insertion sort doesn't depend on future data to sort data input till that point. In other words, insertion sort considers data sorted so far while inserting the next element. This is the key part of insertion sort that makes it an online algorithm.

However, insertion sort takes O(n2) time to sort *n* elements. Perhaps we can use *binary search* on *insertion sort* to find location of next element in O(log n) time. Yet, we can't do data movement in O(log n) time. No matter how efficient the implementation is, it takes polynomial time in case of insertion sort.

Interested readers can try implementation of Method 1.

Method 2: Augmented self balanced binary search tree (AVL, RB, etc…)

At every node of BST, maintain the number of elements in the subtree rooted at that node. We can use a node as the root of a simple binary tree, whose left child is self balancing BST with elements less than root and right child is self balancing BST with elements greater than root. The root element always holds an effective *median*.

If left and right subtrees contain the same number of elements, the root node holds the average of left and right subtree root data. Otherwise, root contains the same data as the root of subtree which is having more elements. After processing an incoming element, the left and right subtrees (BST) are differed utmost by 1.

Self balancing BST is costly in managing the balancing factor of BST. However, they provide sorted data which we don't need. We need a median only. The next method makes use of Heaps to trace median.

Method 3: Heaps

Similar to balancing BST in Method 2 above, we can use a max heap on the left side to represent elements that are less than the effective *median*, and a min heap on the right side to represent elements that are greater than *effective median*.

After processing an incoming element, the number of elements in heaps differ utmost by 1 element. When both heaps contain the same number of elements, we pick the average of heaps root data as an effective *median*. When the heaps are not balanced, we select an effective *median* from the root of the heap containing more elements.

```
#include <iostream>
using namespace std;

// Heap capacity
#define MAX_HEAP_SIZE (128)
#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

//// Utility functions

// exchange a and b
inline
void Exch(int &a, int &b)
{
    int aux = a;
    a = b;
    b = aux;
}

// Greater and Smaller are used as comparators
bool Greater(int a, int b)
{
    return a > b;
}

bool Smaller(int a, int b)
{
    return a < b;
}

int Average(int a, int b)
{
    return (a + b) / 2;
}

// Signum function
// = 0  if a == b  - heaps are balanced
// = -1 if a < b   - left contains less elements than right
// = 1  if a > b   - left contains more elements than right
int Signum(int a, int b)
{
    if( a == b )
        return 0;

    return a < b ? -1 : 1;
}

// Heap implementation
// The functionality is embedded into
// Heap abstract class to avoid code duplication
class Heap
{
public:
    // Initializes heap array and comparator required
    // in heapification
    Heap(int *b, bool (*c)(int, int)) : A(b), comp(c)
    {
        heapSize = -1;
    }

    // Frees up dynamic memory
    virtual ~Heap()
    {
        if( A )
        {
            delete[] A;
        }
    }

    // We need only these four interfaces of Heap ADT
    virtual bool Insert(int e) = 0;
    virtual int  GetTop() = 0;
    virtual int  ExtractTop() = 0;
    virtual int  GetCount() = 0;
```

```

protected:

    // We are also using location 0 of array
    int left(int i)
    {
        return 2 * i + 1;
    }

    int right(int i)
    {
        return 2 * (i + 1);
    }

    int parent(int i)
    {
        if( i <= 0 )
        {
            return -1;
        }

        return (i - 1)/2;
    }

    // Heap array
    int   *A;
    // Comparator
    bool  (*comp)(int, int);
    // Heap size
    int   heapSize;

    // Returns top element of heap data structure
    int top(void)
    {
        int max = -1;

        if( heapSize >= 0 )
        {
            max = A[0];
        }

        return max;
    }

    // Returns number of elements in heap
    int count()
    {
        return heapSize + 1;
    }

    // Heapification
    // Note that, for the current median tracing problem
    // we need to heapify only towards root, always
    void heapify(int i)
    {
        int p = parent(i);

        // comp - differentiate MaxHeap and MinHeap
        // percolates up
        if( p >= 0 && comp(A[i], A[p]) )
        {
            Exch(A[i], A[p]);
            heapify(p);
        }
    }

    // Deletes root of heap
    int deleteTop()
    {
        int del = -1;

        if( heapSize > -1)
        {
            del = A[0];

            Exch(A[0], A[heapSize]);
            heapSize--;
            heapify(parent(heapSize+1));
        }

        return del;
    }

    // Helper to insert key into Heap
    bool insertHelper(int key)
    {
        bool ret = false;

        if( heapSize < MAX_HEAP_SIZE )
        {
            ret = true;
            heapSize++;
            A[heapSize] = key;
            heapify(heapSize);
        }

        return ret;
    }
};

// Specilization of Heap to define MaxHeap
class MaxHeap : public Heap
{
private:

public:
    MaxHeap() : Heap(new int[MAX_HEAP_SIZE], &Greater)   {   }

    ~MaxHeap()   { }

    // Wrapper to return root of Max Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Max Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Max Heap
    int  GetCount()
    {
        return count();
```

```cpp
    }

    // Wrapper to insert into Max Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// Specilization of Heap to define MinHeap
class MinHeap : public Heap
{
private:

public:

    MinHeap() : Heap(new int[MAX_HEAP_SIZE], &Smaller) { }

    ~MinHeap() { }

    // Wrapper to return root of Min Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Min Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Min Heap
    int  GetCount()
    {
        return count();
    }

    // Wrapper to insert into Min Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// Function implementing algorithm to find median so far.
int getMedian(int e, int &m, Heap &l, Heap &r)
{
    // Are heaps balanced? If yes, sig will be 0
    int sig = Signum(l.GetCount(), r.GetCount());
    switch(sig)
    {
    case 1: // There are more elements in left (max) heap

        if( e < m ) // current element fits in left (max) heap
        {
            // Remore top element from left heap and
            // insert into right heap
            r.Insert(l.ExtractTop());

            // current element fits in left (max) heap
            l.Insert(e);
        }
        else
        {
            // current element fits in right (min) heap
            r.Insert(e);
        }

        // Both heaps are balanced
        m = Average(l.GetTop(), r.GetTop());

        break;

    case 0: // The left and right heaps contain same number of elements

        if( e < m ) // current element fits in left (max) heap
        {
            l.Insert(e);
            m = l.GetTop();
        }
        else
        {
            // current element fits in right (min) heap
            r.Insert(e);
            m = r.GetTop();
        }

        break;

    case -1: // There are more elements in right (min) heap

        if( e < m ) // current element fits in left (max) heap
        {
            l.Insert(e);
        }
        else
        {
            // Remove top element from right heap and
            // insert into left heap
            l.Insert(r.ExtractTop());

            // current element fits in right (min) heap
            r.Insert(e);
        }

        // Both heaps are balanced
        m = Average(l.GetTop(), r.GetTop());

        break;
    }

    // No need to return, m already updated
    return m;
}

void printMedian(int A[], int size)
{
    int m = 0; // effective median
    Heap *left  = new MaxHeap();
    Heap *right = new MinHeap();

    for(int i = 0; i < size; i++)
    {
        m = getMedian(A[i], m, *left, *right);

        cout << m << endl;
    }

    // C++ more flexible, ensure no leaks
    delete left;
    delete right;
}
```

Time Complexity: If we omit the way how stream was read, complexity of median finding is ***O(N log N)***, as we need to read the stream, and due to heap insertions/deletions.

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find the second best player?

We can use adversary arguments based on tournament trees (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and an internal node represents the winner. In a tournament tree every internal node contains the winner and every leaf node contains one player.

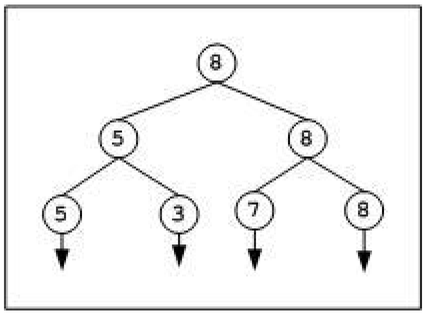
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put n = 2 in the equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need a minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is the number of internal nodes and E is the number of external nodes. It means to find the maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in **($N + \log_2 N – 2$)** comparisons.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of the loser *tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates who participated with the final winner, that leads to 7 as second best.

Median of Sorted Arrays

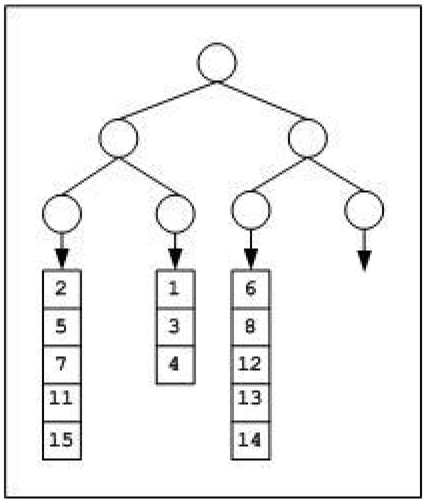
Tournament trees can effectively be used to find the median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2 M$)** to have at least M external nodes.

Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.{ 2, 5, 7, 11, 15 } ---- Array1

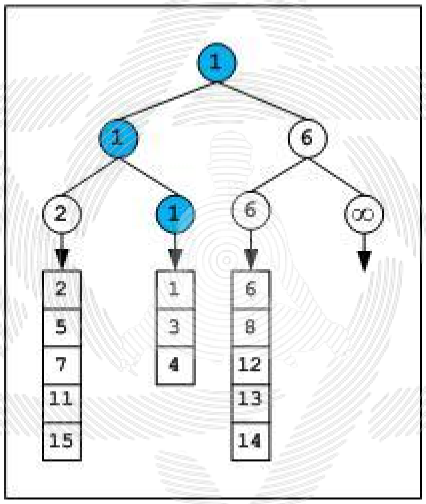
{1, 3, 4} ---- Array2

{6, 8, 12, 13, 14} ---- Array3

What should be the height of the tournament tree? We need to construct a tournament tree of height $\log_2 3$.= 1.585 = 2 rounded to the next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



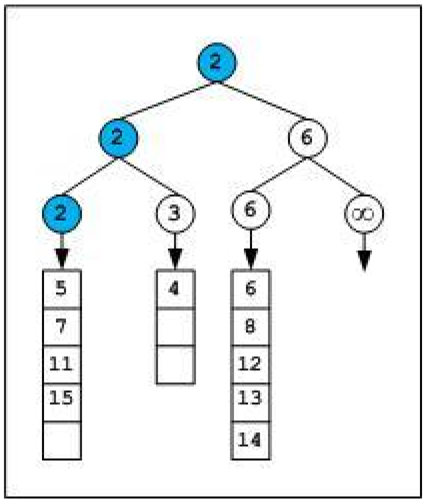
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of the previous tournament.

Note that infinity is used as a sentinel element. Based on data being held in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as a sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so the next element of the Array1 array which is 5 will dive-in to the next round, and the next tournament played along the path of 2.

The tournaments can be continued till we get a median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2 … L_m requires time complexity of ***$O((L_1 + L_2 + … + L_m) * \log M)$*** to merge all the arrays, and ***$O(m*\log M)$*** time to find median, where ***m*** is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select the first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use a tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree in a bottom-up manner. All the leaf nodes filled first. Start at the left extreme of the tree and fill along the breadth (i.e. from 2^{k-1} to 2^k – 1 where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. Implementation is discussed in below code

Given an array of integers, find the minimum (or maximum) element and the element just greater (or smaller) than that in less than 2n comparisons. The given array is not necessarily sorted. Extra space is allowed.

Examples:

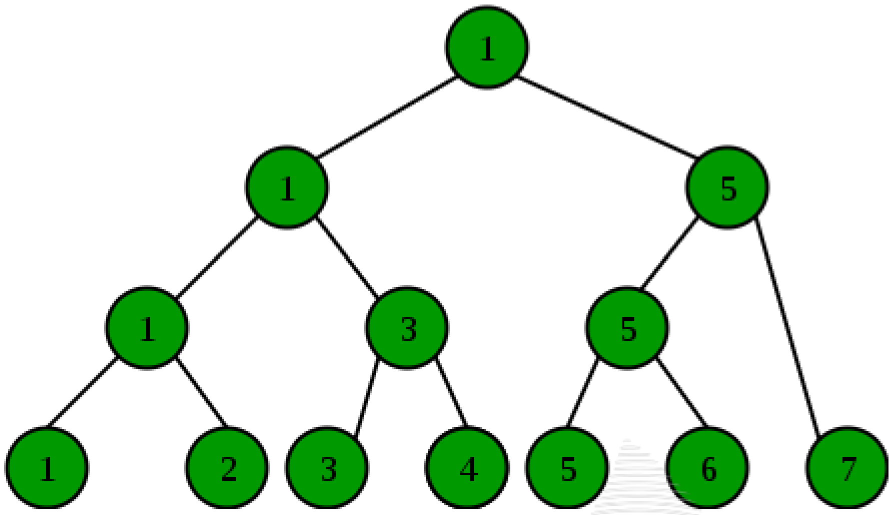
Input: {3, 6, 100, 9, 10, 12, 7, -1, 10}

Output: Minimum: -1, Second minimum: 3

Find the smallest and second smallest element in an array

Comparisons of array elements can be costly if array elements are of large size, for example large strings. We can minimize the number of comparisons used in the above approach.

The idea is based on a tournament tree. Consider each element in the given array as a leaf node.



First, we find the minimum element as explained by building a tournament tee. To build the tree, we compare all adjacent pairs of elements (leaf nodes) with each other. Now we have n/2 elements which are smaller than their counterparts (from each pair, the smaller element forms the level above that of the leaves). Again, find the smaller elements in each of the n/4 pairs. Continue this process until the root of the tree is created. The root is the minimum.

Next, we traverse the tree and while doing so, we discard the sub trees whose root is greater than the smallest element. Before discarding, we update the second smallest element, which will hold our result. The point to understand here is that the tree is height balanced and we have to spend only logn time to traverse all the elements that were ever compared to the minimum in step 1. Thus, the overall time complexity is **n + logn**. Auxiliary space needed is O(2n), as the number of leaf nodes will be approximately equal to the number of internal nodes.

```cpp
// C++ program to find minimum and second minimum
// using minimum number of comparisons
#include <bits/stdc++.h>
using namespace std;

// Tournament Tree node
struct Node
{
    int idx;
    Node *left, *right;
};

// Utility function to create a tournament tree node
Node *createNode(int idx)
{
    Node *t = new Node;
    t->left = t->right = NULL;
    t->idx = idx;
    return t;
}

// This function traverses tree across height to
// find second smallest element in tournament tree.
// Note that root is smallest element of tournament
// tree.
void traverseHeight(Node *root, int arr[], int &res)
{
    // Base case
    if (root == NULL || (root->left == NULL &&
                        root->right == NULL))
        return;

    // If left child is smaller than current result,
    // update result and recur for left subarray.
    if (res > arr[root->left->idx] &&
        root->left->idx != root->idx)
    {
        res = arr[root->left->idx];
        traverseHeight(root->right, arr, res);
    }

    // If right child is smaller than current result,
    // update result and recur for left subarray.
    else if (res > arr[root->right->idx] &&
             root->right->idx != root->idx)
    {
        res = arr[root->right->idx];
        traverseHeight(root->left, arr, res);
    }
}

// Prints minimum and second minimum in arr[0..n-1]
void findSecondMin(int arr[], int n)
{
    // Create a list to store nodes of current
    // level
    list<Node *> li;

    Node *root = NULL;
    for (int i = 0; i < n; i += 2)
    {
        Node *t1 = createNode(i);
        Node *t2 = NULL;
        if (i + 1 < n)
        {
            // Make a node for next element
            t2 = createNode(i + 1);

            // Make smaller of two as root
            root = (arr[i] < arr[i + 1])? createNode(i) :
createNode(i + 1);

            // Make two nodes as children of smaller
            root->left = t1;
            root->right = t2;

            // Add root
            li.push_back(root);
        }
        else
            li.push_back(t1);
    }

    int lsize = li.size();

    // Construct the complete tournament tree from above
    // prepared list of winners in first round.
    while (lsize != 1)
    {
        // Find index of last pair
        int last = (lsize & 1)? (lsize - 2) : (lsize - 1);

        // Process current list items in pair
        for (int i = 0; i < last; i += 2)
        {
            // Extract two nodes from list, make a new
            // node for winner of two
            Node *f1 = li.front();
            li.pop_front();

            Node *f2 = li.front();
            li.pop_front();
            root = (arr[f1->idx] < arr[f2->idx])?
                createNode(f1->idx) : createNode(f2->idx);

            // Make winner as parent of two
            root->left = f1;
            root->right = f2;

            // Add winner to list of next level
            li.push_back(root);
        }
        if (lsize & 1)
        {
            li.push_back(li.front());
            li.pop_front();
        }
        lsize = li.size();
    }

    // Traverse tree from root to find second minimum
    // Note that minimum is already known and root of
    // tournament tree.
    int res = INT_MAX;
    traverseHeight(root, arr, res);
    cout << "Minimum: " << arr[root->idx]
        << ", Second minimum: " << res << endl;
}
```

Find k numbers with most occurrences in the given array

Given an array of **n** numbers and a positive integer **k**. The problem is to find **k** numbers with most occurrences, i.e., the top **k** numbers having the maximum frequency. If two numbers have the same frequency then the larger number should be given preference. The numbers should be displayed in decreasing order of their frequencies. It is assumed that the array consists of **k** numbers with most occurrences.

Examples:

Input:

arr[] = {3, 1, 4, 4, 5, 2, 6, 1},

k = 2

Output: *4 1*

Explanation:

*Frequency of **4** = 2*

*Frequency of **1** = 2*

These two have the maximum frequency and

***4** is larger than **1**.*

Input :

arr[] = {7, 10, 11, 5, 2, 5, 5, 7, 11, 8, 9},

k = 4

Output: *5 11 7 10*

Explanation:

*Frequency of **5** = 3*

*Frequency of **11** = 2*

*Frequency of **7** = 2*

*Frequency of **10** = 1*

These four have the maximum frequency and

***5** is largest among rest.*

Method 1:

- **Approach:** The thought process should begin from creating a HashMap to store element-frequency pair in the HashMap. HashMap is used to perform insertion and updation in constant time. Then sort the element-frequency pair in decreasing order of frequency. This gives the information about each element and the number of times they are present in the array. To get k elements of the array, print the first k elements of the sorted array.

- **Hashmap:** HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. HashMap is known as HashMap because it uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing and faster searches. HashSet also uses HashMap internally. It internally uses a link list to store key-value pairs already explained in HashSet in detail and further articles.

  More on HashMap >>
- **Algorithm:**
  1. Create a Hashmap hm, to store key-value pair, i.e. element-frequency pair.
  2. Traverse the array from start to end.
  3. For every element in the array update hm[array[i]]++
  4. Store the element-frequency pair in a vector and sort the vector in decreasing order of frequency.
  5. Print the first k elements of sorted array

```
// comparison function to sort the 'freq_arr[]'
bool compare(pair<int, int> p1, pair<int, int> p2)
{
    // if frequencies of two elements are same
```

```
•       // then the larger number should come first
•       if (p1.second == p2.second)
•           return p1.first > p2.first;
•
•       // sort on the basis of decreasing order
•       // of frequencies
•       return p1.second > p2.second;
• }
•
• // funnction to print the k numbers with most
  occurrences
• void print_N_mostFrequentNumber(int arr[], int n, int k)
• {
•       // unordered_map 'um' implemented as frequency hash
  table
•       unordered_map<int, int> um;
•       for (int i = 0; i < n; i++)
•           um[arr[i]]++;
•
•       // store the elements of 'um' in the vector
  'freq_arr'
•       vector<pair<int, int> > freq_arr(um.begin(),
  um.end());
•
•       // sort the vector 'freq_arr' on the basis of the
•       // 'compare' function
•       sort(freq_arr.begin(), freq_arr.end(), compare);
•
•       // display the top k numbers
•       cout << k << " numbers with most occurrences
  are:\n";
•       for (int i = 0; i < k; i++)
•             cout << freq_arr[i].first << " ";
• }
```

Complexity Analysis:

- **Time Complexity:** O(d log d), where **d** is the count of distinct elements in the array. To sort the array O(d log d) time is needed.
- **Auxiliary Space:** O(d), where **d** is the count of distinct elements in the array. To store the elements in HashMap O(d) space complexity is needed.

Method 2:

- **Approach:** Create a HashMap to store element-frequency pair in the HashMap. HashMap is used to perform insertion and updation in constant time. Then use a priority queue to store the element-frequency pair (Max-Heap). This gives the element which has maximum frequency at the root of the Priority Queue. Remove the top or root of Priority Queue K times and print the element. To insert and delete the top of the priority queue *O(log n)* time is required.
  Priority Queue: Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non increasing order(hence we can see that each element of the queue has

a priority{fixed order}).

More Info about Priority Queue: C++ STL priority_queue

- **Algorithm :**

    1. Create a Hashmap *hm*, to store key-value pair, i.e. element-frequency pair.
    2. Traverse the array from start to end.
    3. For every element in the array update *hm[array[i]]++*
    4. Store the element-frequency pair in a Priority Queue and create the Priority Queue, this takes O(n) time.
    5. Run a loop k times, and in each iteration remove the top of the priority queue and print the element.

```
/// comparison function defined for the priority queue
```

```
struct compare {
    bool operator()(pair<int, int> p1, pair<int, int> p2)
    {
        // if frequencies of two elements are same
        // then the larger number should come first
        if (p1.second == p2.second)
            return p1.first < p2.first;

        // insert elements in the priority queue on the basis of
        // decreasing order of frequencies
        return p1.second < p2.second;
    }
};

// funnction to print the k numbers with most occurrences
void print_N_mostFrequentNumber(int arr[], int n, int k)
{
    // unordered_map 'um' implemented as frequency hash table
    unordered_map<int, int> um;
    for (int i = 0; i < n; i++)
        um[arr[i]]++;



    // priority queue 'pq' implemented as max heap on the basis
    // of the comparison operator 'compare'
    // element with the highest frequency is the root of 'pq'
    // in case of conflicts, larger element is the root
    priority_queue<pair<int, int>, vector<pair<int, int> >,
                   compare>
        pq(um.begin(), um.end());

    // display the top k numbers
    cout << k << " numbers with most occurrences are:\n";
    for (int i = 1; i <= k; i++) {
        cout << pq.top().first << " ";
        pq.pop();
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(k log d + d), where **d** is the count of distinct elements in the array.

  To remove the top of priority queue O(log d) time is required, so if k elements are removed then O(k log d) time is required and to traverse the distinct elements O(d) time is required.

- **Auxiliary Space:** O(d), where **d** is the count of distinct elements in the array. To store the elements in HashMap O(d) space complexity is needed.