Heap

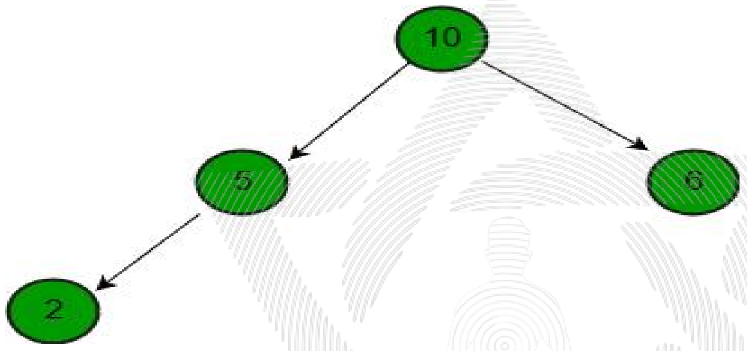
Lesson-5

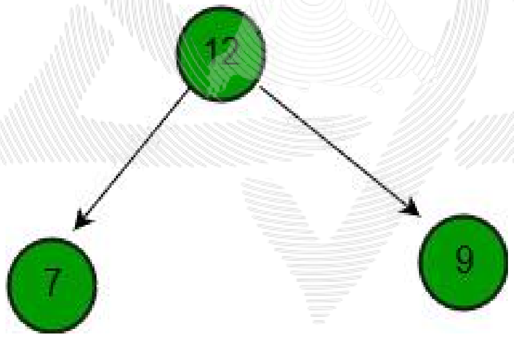
Merge two binary Max Heaps

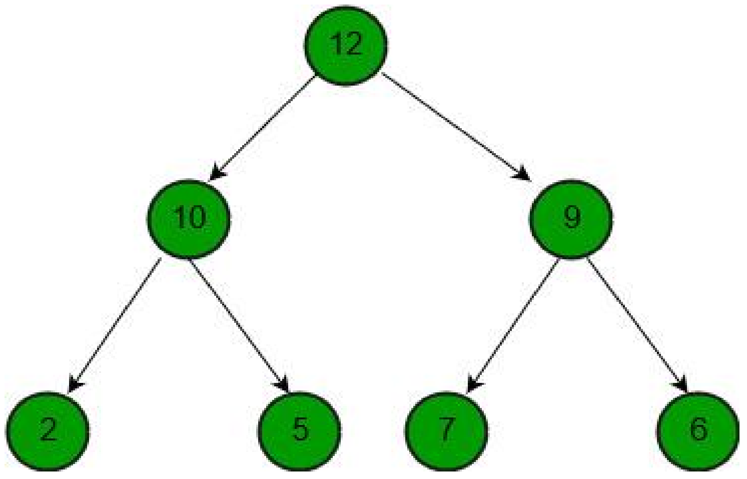
Given two binary max heaps as arrays, merge the given heaps.

Examples :

```
Input  : a = {10, 5, 6, 2},
         b = {12, 7, 9}
Output : {12, 10, 9, 2, 5, 7, 6}
```







The idea is simple. We create an array to store result. We copy both given arrays one by one to result. Once we have copied all elements, we call standard build heap to construct full merged max heap.

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

#include <bits/stdc++.h>
using namespace std;

// Standard heapify function to heapify a
// subtree rooted under idx. It assumes
// that subtrees of node are already heapified.
 void maxHeapify(int arr[], int n, int idx)
 {
     // Find largest of node and its children
     if (idx >= n)
         return;
     int l = 2 * idx + 1;
     int r = 2 * idx + 2;
     int max;
     if (l < n && arr[l] > arr[idx])
         max = l;
     else
         max = idx;
     if (r < n && arr[r] > arr[max])
         max = r;

     // Put maximum value at root and
     // recur for the child with the
     // maximum value
     if (max != idx) {
         swap(arr[max], arr[idx]);
         maxHeapify(arr, n, max);
     }
 }
```

```

// Builds a max heap of given arr[0..n-1]
void buildMaxHeap(int arr[], int n)
{
    // building the heap from first non-leaf
    // node by calling max heapify function
    for (int i = n / 2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);
}

// Merges max heaps a[] and b[] into merged[]
void mergeHeaps(int merged[], int a[], int b[],
                int n, int m)
{
    // Copy elements of a[] and b[] one by one
    // to merged[]
    for (int i = 0; i < n; i++)
        merged[i] = a[i];
    for (int i = 0; i < m; i++)
        merged[n + i] = b[i];

    // build heap for the modified array of
    // size n+m
    buildMaxHeap(merged, n + m);
}
```

Time Complexity:

Since time complexity for building the heap from an array of n elements is O(n). The complexity of merging the heaps is equal to O(n + m).

K-th Largest Sum Contiguous Subarray

Given an array of integers. Write a program to find the K-th largest sum of contiguous subarray within the array of numbers which has negative and positive numbers.

Examples:

```
Input: a[] = {20, -5, -1}
         k = 3
Output: 14
Explanation: All sum of contiguous
subarrays are (20, 15, 14, -5, -6, -1)
so the 3rd largest sum is 14.

Input: a[] = {10, -10, 20, -40}
         k = 6
Output: -10
Explanation: The 6th largest sum among
sum of all contiguous subarrays is -10.
```

A **brute force approach** approach is to store all the contiguous sums in another array and sort it, and print the k-th largest. But in case of number of elements being large, the array in which we store the contiguous sums will run out of memory as the number of contiguous subarrays will be large (quadratic order)

An **efficient approach** is store the pre-sum of the array in a sum[] array. We can find sum of contiguous subarray from index i to j as sum[j]-sum[i-1]

Now for storing the Kth largest sum, use a min heap (priority queue) in which we push the contiguous sums till we get K elements, once we have our K elements, check if the element is greater than the Kth element it is inserted to the min heap with popping out the top element in the min-heap, else not inserted . At the end the top element in the min-heap will be your answer.

```
// function to calculate kth largest element
// in contiguous subarray sum
int kthLargestSum(int arr[], int n, int k)
{
    // array to store predix sums
    int sum[n + 1];
    sum[0] = 0;
    sum[1] = arr[0];
    for (int i = 2; i <= n; i++)
        sum[i] = sum[i - 1] + arr[i - 1];

    // priority_queue of min heap
    priority_queue<int, vector<int>, greater<int> > Q;

    // loop to calculate the contigous subarray
    // sum position-wise
    for (int i = 1; i <= n; i++)
    {

        // loop to traverse all positions that
        // form contiguous subarray
        for (int j = i; j <= n; j++)
        {
            // calculates the contiguous subarray
            // sum from j to i index
            int x = sum[j] - sum[i - 1];

            // if queue has less then k elements,
            // then simply push it
            if (Q.size() < k)
                Q.push(x);

            else
            {
                // it the min heap has equal to
                // k elements then just check
                // if the largest kth element is
                // smaller than x then insert
                // else its of no use
                if (Q.top() < x)
                {
                    Q.pop();
                    Q.push(x);
                }
            }
        }
    }

    // the top element will be then kth
    // largest element
    return Q.top();
    }
```

Time complexity: O(n^2 log (k))

Auxiliary Space : O(k) for min-heap and we can store the sum array in the array itself as it is of no use.

Rearrange characters in a string such that no two adjacent are same

Given a string with repeated characters, the task is to rearrange characters in a string so that no two adjacent characters are the same.

Note : It may be assumed that the string has only lowercase English alphabets.

Examples:

```
Input: aaabc
Output: abaca

Input: aaabb
Output: ababa

Input: aa
Output: Not Possible

Input: aaaabc
Output: Not Possible
```

The idea is to put the highest frequency character first (a greedy approach). We use a priority queue (Or Binary Max Heap) and put all characters and orders by their frequencies (highest frequency character at root). We one by one take the highest frequency character from the heap and add it to the result. After we add, we decrease the frequency of the character and we temporarily move this character out of the priority queue so that it is not picked next time.
We have to follow the step to solve this problem, they are:
1. Build a Priority_queue or max_heap, **pq** that stores characters and their frequencies.

…… Priority_queue or max_heap is built on the basis of the frequency of character.
2. Create a temporary Key that will be used as the previously visited element (the previous element in the resultant string. Initialize it { char = ‘#’ , freq = ‘-1’ }
3. While **pq** is not empty.
….. Pop an element and add it to the result.
….. Decrease frequency of the popped element by ‘1’
….. Push the previous element back into the priority_queue if it’s frequency > ‘0’
….. Make the current element as the previous element for the next iteration.
4. If the length of the resultant string and original string is not equal, print “not possible”. Else print result.

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX_CHAR = 26;

struct Key {
    int freq; // store frequency of character
    char ch;

    // function for priority_queue to store Key
    // according to freq
    bool operator<(const Key& k) const
    {
        return freq < k.freq;
    }
};

// Function to rearrange character of a string
// so that no char repeat twice
void rearrangeString(string str)
{
    int n = str.length();

    // Store frequencies of all characters in string
    int count[MAX_CHAR] = { 0 };
    for (int i = 0; i < n; i++)
        count[str[i] - 'a']++;

    // Insert all characters with their frequencies
    // into a priority_queue
    priority_queue<Key> pq;
    for (char c = 'a'; c <= 'z'; c++)
        if (count)
            pq.push(Key{ count, c });

    // 'str' that will store resultant value
    str = "";

    // work as the previous visited element
    // initial previous element be. ( '#' and
    // it's frequency '-1' )
    Key prev{ -1, '#' };

    // traverse queue
    while (!pq.empty()) {
        // pop top element from queue and add it
        // to string.
        Key k = pq.top();
        pq.pop();
        str = str + k.ch;

        // IF frequency of previous character is less
        // than zero that means it is useless, we
        // need not to push it
        if (prev.freq > 0)
            pq.push(prev);

        // make current character as the previous 'char'
        // decrease frequency by 'one'
        (k.freq)--;
        prev = k;
    }

    // If length of the resultant string and original
    // string is not same then string is not valid
    if (n != str.length())
        cout << " Not valid String " << endl;

    else // valid string
        cout << str << endl;
}
```

Time complexity : O(nlog(n))

Another approach :

Another approach is to fill all the even positions of the result string first, with the highest frequency character. If there are still some even positions remaining, fill them first. Once even positions are done, then fill the odd positions. This way, we can ensure that no two adjacent characters are the same.

```
#include <bits/stdc++.h>
using namespace std;

char getMaxCountChar(const vector<int>& count)
{
    int max = 0;
    char ch;
    for (int i = 0; i < 26; i++) {
        if (count[i] > max) {
                max = count[i];
                ch = 'a' + i;
            }
```

```cpp
    }

    return ch;
}

string rearrangeString(string S)
{

    int n = S.size();
    if (!n)
        return "";

    vector<int> count(26, 0);
    for (auto ch : S)
        count[ch - 'a']++;

    char ch_max = getMaxCountChar(count);
    int maxCount = count[ch_max - 'a'];

    // check if the result is possible or not
    if (maxCount > (n + 1) / 2)
        return "";

    string res(n, ' ');

    int ind = 0;
    // filling the most frequently occuring char in the even
    // indices
    while (maxCount) {
        res[ind] = ch_max;
        ind = ind + 2;
        maxCount--;
    }
    count[ch_max - 'a'] = 0;

    // now filling the other Chars, first filling the even
    // positions and then the odd positions
    for (int i = 0; i < 26; i++) {
        while (count[i] > 0) {
            ind = (ind >= n) ? 1 : ind;
            res[ind] = 'a' + i;
            ind += 2;
            count[i]--;
        }
    }
    return res;
 }
```

Time complexity : O(n)

Space complexity : O(n+26) where 26 is the size of the vocabulary.

Sum of all elements between k1’th and k2’th smallest elements

Given an array of integers and two numbers k1 and k2. Find the sum of all elements between given two k1’th and k2’th smallest elements of the array. It may be assumed that (1 <= k1 < k2 <= n) and all elements of the array are distinct.

Examples :

Input : arr[] = {20, 8, 22, 4, 12, 10, 14}, k1 = 3, k2 = 6
Output : 26
    The 3rd smallest element is 10. 6th smallest element
    is 20. Sum of all element between k1 & k2 is
    12 + 14 = 26

Input : arr[] = {10, 2, 50, 12, 48, 13}, k1 = 2, k2 = 6
Output : 73

Method 1 (Sorting)

First sort the given array using a O(n log n) sorting algorithm like Merge Sort, Heap Sort, etc and return the sum of all elements between index k1 and k2 in the sorted array.

```
int sumBetweenTwoKth(int arr[], int n, int k1, int k2)
{
    // Sort the given array
    sort(arr, arr + n);

    /* Below code is equivalent to
```

```
      int result = 0;
      for (int i=k1; i<k2-1; i++)
       result += arr[i]; */
      return accumulate(arr + k1, arr + k2 - 1, 0);
  }
```

Time Complexity: O(n log n)

Method 2 (Using Min Heap)

We can optimize the above solution by using a min heap.

1) Create a min heap of all array elements. (This step takes O(n) time)

2) Do extract minimum k1 times (This step takes O(K1 Log n) time)

3) Do extract minimum k2 – k1 – 1 time and sum all extracted elements. (This step takes O ((K2 – k1) * Log n) time)

```
#include <bits/stdc++.h>
using namespace std;

int n = 7;

void minheapify(int a[], int index)
{

  int small = index;
    int l = 2 * index + 1;
    int r = 2 * index + 2;

    if (l < n && a[l] < a[small])
        small = l;

    if (r < n && a[r] < a[small])
        small = r;

    if (small != index) {
        swap(a[small], a[index]);
        minheapify(a, small);
    }
}

int main()
{
    int i = 0;
    int k1 = 3;
    int k2 = 6;

    int a[] = { 20, 8, 22, 4, 12, 10, 14 };

    int ans = 0;

    for (i = (n / 2) - 1; i >= 0; i--) {
        minheapify(a, i);
    }

    // decreasing value by 1 because we want min heapifying k times and it starts
    // from 0 so we have to decrease it 1 time
    k1--;
    k2--;

    // Step 1: Do extract minimum k1 times (This step takes O(K1 Log n) time)
    for (i = 0; i <= k1; i++) {
        // cout<<a[0]<<endl;
        a[0] = a[n - 1];
        n--;
        minheapify(a, 0);
    }

    /*Step 2: Do extract minimum k2 - k1 - 1 times and sum all
    extracted elements. (This step takes O ((K2 - k1) * Log n) time)*/
    for (i = k1 + 1; i < k2; i++) {
        // cout<<a[0]<<endl;
        ans += a[0];
        a[0] = a[n - 1];
        n--;
        minheapify(a, 0);
    }

    cout << ans;

    return 0;
}
```

Time Complexity:

Overall time complexity of this method is O(n + k2 Log n) which is better than sorting based method

K’th largest element in a stream

Given an infinite stream of integers, find the k’th largest element at any point of time.

Example:

Input:

stream[] = {10, 20, 11, 70, 50, 40, 100, 5, ...}

k = 3

Output: {_, _, 10, 11, 20, 40, 50, 50, ...}

Extra space allowed is O(k).

A **Simple Solution** is to keep an array of size k. The idea is to keep the array sorted so that the k’th largest element can be found in O(1) time (we just need to return first element of array if array is sorted in increasing order)

How to process a new element of a stream?

For every new element in the stream, check if the new element is smaller than the current k’th largest element. If yes, then ignore it. If not, then remove the smallest element from the array and insert the new element in sorted order. **Time complexity of processing a new element is O(k).**

A **Better Solution** is to use a Self Balancing Binary Search Tree of size k. The k’th largest element can be found in O(Logk) time.

How to process a new element of stream?

For every new element in the stream, check if the new element is smaller than the current k'th largest element. If yes, then ignore it. If no, then remove the smallest element from the tree and insert a new element. **Time complexity of processing a new element is O(Logk).**

An **Efficient Solution** is to use Min Heap of size k to store k largest elements of stream. The k'th largest element is always at root and can be found in O(1) time.

How to process a new element of stream?

Compare the new element with the root of the heap. If a new element is smaller, then ignore it. Otherwise replace root with new element and call heapify for the root of the modified heap. **Time complexity of finding the k'th largest element is O(Logk).**

```
void kthLargest(int k)
{
    // count is total no. of elements in stream seen so far
    int count = 0, x; // x is for new element

    // Create a min heap of size k
    int* arr = new int[k];
    MinHeap mh(arr, k);

      while (1) {
          // Take next element from stream
          cout << "Enter next element of stream ";
          cin >> x;

          // Nothing much to do for first k-1 elements
          if (count < k - 1) {
              arr[count] = x;
              count++;
          }

          else {
              // If this is k'th element, then store it
```

```
                // and build the heap created above
                if (count == k - 1) {
                    arr[count] = x;
                    mh.buildHeap();
                }

                else {
                    // If next element is greater than
                    // k'th largest, then replace the root
                    if (x > mh.getMin())
                        mh.replaceMin(x); // replaceMin calls
                                          // heapify()
                }

                // Root of heap is k'th largest element
                cout << "K'th largest element is "
                     << mh.getMin() << endl;
                count++;
            }
        }
    }
```

Implementation using Priority Queue:

```
vector<int> kthLargest(int k, int arr[], int n)
{
    vector<int> ans(n);

    // Creating a min-heap using priority queue
    priority_queue<int, vector<int>, greater<int> > pq;

    // Iterating through each element
    for (int i = 0; i < n; i++) {
        // If size of priority
        // queue is less than k
        if (pq.size() < k)
            pq.push(arr[i]);
        else {
            if (arr[i] > pq.top()) {
                pq.pop();
                pq.push(arr[i]);
            }
        }

        // If size is less than k
        if (pq.size() < k)
            ans[i] = -1;
        else
            ans[i] = pq.top();
    }

    return ans;
}
```