26/12/2023
Tuesday

* **Remove stones to minimize the total**

Given an array piles, where piles[i] represents the number of stones in the ith pile and an integer k.
Choose any piles[i] and remove floor(piles[i]/2) stones from it. This can be done k times.
Return the minimum possible total number of stones.

I/P- piles = [4, 3, 6, 7], K = 3

O/P- 12

Explanation-

1st operation- piles = [4, 3, 6, 4], K = 2
IInd operation- piles = [4, 3, 3, 4], K = 1
IIIrd operation- piles = [2, 3, 3, 4], K = 0

Total number of minimum stones in [2, 3, 3, 4] is 12.

Approach-

- Make a max heap of array piles. Max heap because total ko minimize krna hai and it can be done by performing operations on maximum number of piles array.
- while (K != 0), top element i.e. max ko max -= max/2 kro.
- Now, jo bhi elements left hai max heap mei, unka sum return krdo and that's the answer.

Code-

```
class Solution{
public:

int minStoneSum(vector<int>&piles , int K){

    priority_queue<int>maxHeap;

    for(int i=0; i< piles.size(); i++){
            maxHeap.push(piles[i]);
        }

    while(K--){
            int top = maxHeap.top();
            maxHeap.pop();
            top -= top/2;
            maxHeap.push(top);
        }

    int sum=0;
    while(!maxHeap.empty()){
            int top= maxHeap.top();
            maxHeap.pop();
            sum+=top;
        }

    return sum;
    }
};
```

* Reorganize String

Given a string s, rearrange the characters of s so that any two adjacent characters are not the same.
Return any possible rearrangement of s or return "" if not possible.

I/P - s = "aaaaaabbbccc"

O/P - "abababacacac"

Approach -

= Calculate frequency of all characters in string.

= Push those characters in max heap whose frequency > 0.

= while(maxHeap.size() > 1)
Push top 2 elements in ans string.
Update frequency.
If count of character is remaining, then push those characters in string again.

= As we are processing two characters everytime, it may be possible for odd size string that one character remains for processing.
For that, vo case alg se handle kro.

= Now, if heap me jo 1 character remaining tha, agr uska count abhi bhi 0 nhi hua, it means rearrangement isn't possible. Return empty string.

Note -

(1) Max Heap isliye bnayi because starting mein humare pas number of blocks or space jyada hai, so us character ko utilise krna hai jiski frequency maximum hai so that no two similar characters come adjacent in string.

(2) Everytime we are picking 2 characters for processing. This is because possibility hai agr 1-1 krke kre to 2 adjacent characters jo similar hai ek saath aa jaye.

for instance, if frequency of a=4, b=2

Now, output string mein push hoga (a) and its frequency reduces to 3. At this instance, frequency (a) ki abhi bhi (b) ki frequency se greater hai. So, again output string mein (a) push ho jayega which we don't want. That's why everytime, we are procesing top 2 characters with maximum frequency in max heap.

Code-

```
class Info{
public:
char ch;
int count;

Info(char ch, int count){
        this->ch= ch;
        this->count = count;
      }
};

class compare{
public:
bool operator()(Info* a, Info* b){
       return a->count < b->count;
      }
};

class Solution{
public:

string reorganizeString(string s){

       vector<int> freq(26,0);

// calculate frequency of all characters in string
       for(int i=0; i<s.length(); i++){
              char ch= s[i];
              freq[ch-'a']++;
            }
```

```
priority_queue<Info*, vector<Info*>, compare> maxHeap;

//push those characters in max heap whose frequency>0
    for (int i=0; i<freq.size(); i++){
        if (freq[i] > 0) {
        Info* temp = new Info(i+'a', freq[i]);
        maxHeap.push(temp);
    }

string ans = "";

while (maxHeap.size() > 1) {
  Info * first = maxHeap.top();
  maxHeap.pop();
  Info * second= maxHeap.top();
  maxHeap.pop();

  ans.push_back(first -> ch);
  first -> count --;
  ans.push_back(second -> ch);
  second -> count --;

// if first or second ka count remaining hai, then
// again push krdo max heap mein

  if (first -> count > 0) maxHeap.push(first);
  if (second -> count > 0) maxHeap.push(second);
}
```

```
// possibility hai heap mein 1 character bcha ho
// use alg se handle kro
    if (maxHeap.size()==1){
      Info* first = maxHeap.top();
      maxHeap.pop();

      ans.push_back(first->ch);
      first->count--;

      // if this, it means ans isn't possible
      if(first->count != 0) return "";
    }

    return ans;
  }
};
```

* **Longest Happy String -**

A string s is called happy if it satisfies the following conditions -

- s contains the letters 'a', 'b' and 'c'.
- s does not contain any of "aaa", "bbb" or "ccc" as a substring.
- s contains at most (a) occurrences of letter 'a', atmost (b) occurrences of letter 'b' and atmost (c) occurrences of letter 'c'.

I/P - a = 5 , b = 3, c = 2
O/P - "aabbaaccab"

Approach -

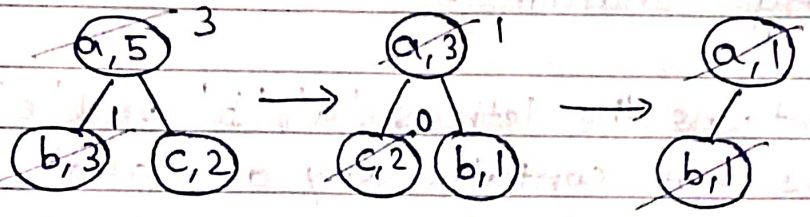
= Push 'a', 'b' and 'c' with their frequencies in max heap.
= while (maxHeap.size() > 1)
- Fetch top two elements in heap.
- Process for the answer.
- Insert those elements in heap again if their count is remaining.

= Handle the case separately when - (maxHeap.size == 1)

Note - Everytime, we are processing top two elements of heap. Therefore, the condition inside while is (maxHeap.size() > 1). It means jb heap me 2 characters honge tbhi while loop chlegi.

For instance,
a = 5 , b = 3 and c = 2

Output - "aabbaaccab"



Code -

```
class Info {
public:
char ch;
int count;

Info (char ch, int count) {
      this -> ch = ch;
      this -> count = count;
    }
};

class compare{
public:
bool operator () (Info a, Info b) {
      return a.count < b.count;
    }
};
```

```
class Solution {
public:

string longestDiverseString(int a, int b, int c){

 priority_queue<Info, vector<Info>, compare> maxHeap;

  if (a > 0) {
    Info temp ('a', a);
    maxHeap.push(temp);
   }
  if (b > 0) {
    Info temp ('b', b);
    maxHeap.push(temp);
   }
  if (c > 0) {
    Info temp ('c', c);
    maxHeap.push (temp);
   }

  string ans = " ";

  while (maxHeap.size() > 1) {
//fetch  Info first = maxHeap.top();
    maxHeap.pop();
    Info second = maxHeap.top();
    maxHeap.pop();
   // process the ans string
    if (first.count >= 2) {
        ans.push_back (first.ch);
        ans.push_back (first.ch);
        first.count -= 2;
     }
```

```
        else {
            ans.push_back(first.ch);
            first.count--;
        }

        if (second.count >=2 && second.count >=first.count){
            ans.push_back(second.ch);
            ans.push_back(second.ch);
            second.count -=2;
        }
        else{
            ans.push_back(second.ch);
            second.count--;
        }
        //insert if count is remaining
        if(first.count >0) maxHeap.push(first);
        if(second.count >0) maxHeap.push(second);
    }
    // Handle separately when heap size is 1.
    if(maxHeap.size()==1){
        Info first = maxHeap.top();
        maxHeap.pop();

        if(first.count >=2){
            ans.push_back(first.ch);
            ans.push_back(first.ch);
            first.count-=2;
        }
        else{
            ans.push_back(first.ch);
            first.count--;
        } }
    return ans; } };
```

* Median in a stream -

The median is the middle value in a sorted list. If the size of list is even, there is no middle value and the median is the mean of two middle values. If the size of list is odd, median is the middle value.

I/P - arr[] = [5, 15, 1, 3, 2, 8]

O/P - 5 10 5 4 3 4

Explanation -

Median when stream is -

5	→	5				
5	15	→	10			
5	15	1	→	5		
5	15	1	3	→	4	
5	15	1	3	2	→	3
5	15	1	3	2	8	→ 4

Approach -

= Make a max heap and min heap.
= We will find the median for every stream.
= It depends whether we are inserting the incoming element in max heap or min heap.
= We are assuming the left of median as max Heap and right part as min Heap.

= We will take care of that the absolute difference between sizes of max heap and min heap is atmost 1.

= On the basis of that, we are dividing the whole into 3 possible cases.

Observations –

1. maxHeap.size() == minHeap.size()

   Median = (maxHeap.top() + minHeap.top()) / 2

2. maxHeap.size() = minHeap.size() + 1

   Median = maxHeap.top();

3. minHeap.size() = maxHeap.size() + 1

   Median = minHeap.top();

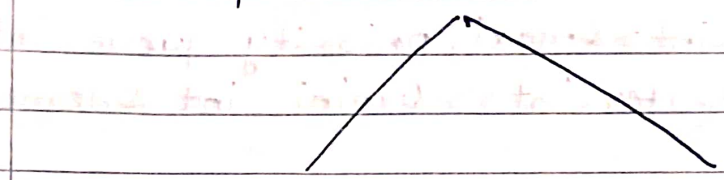
Insertion (whether maxHeap or minHeap)

1. maxHeap.size() == minHeap.size()

   (element > median) → minHeap.push(element)

   (element < median) → maxHeap.push(element)

2. maxHeap.size() == minHeap.size() + 1

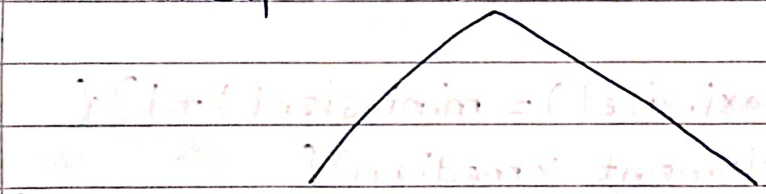


(element > median) (element < median)

minHeap.push(element)

(1) Push maxHeap.top() into minHeap.
(2) maxHeap.push(element)

3. minHeap.size() == maxHeap.size() + 1



(element > median) (element < median)

(1) Push minHeap.top() into maxHeap.
(2) minHeap.push(element)

maxHeap.push(element)

Code -

```
void medianInAStream (vector<int>&v, double &median,
priority_queue<int> &maxi, priority_queue <int,
vector<int>, greater<int>> &mini , int &element)
{
    if (maxi.size() == mini.size()) {
        if (element > median){
            mini.push (element);
            median= mini.top();
        }
        else{
            maxi.push(element);
            median = maxi.top();
        }
    }
    else if (maxi.size() = mini.size()+1){
        if(element > median){
            mini.push(element);
        }
        else{
            int maxTop = maxi.top();
            maxi.pop();
            mini.push (maxTop);
            maxi.push(element);
        }
    median = (maxi.top() + mini.top()) / 2;
}
```

```
else if (mini.size() = maxi.size() + 1){
      if (element > median){
        int minTop = mini.top();
        mini.pop();
        maxi.push(minTop);
        mini.push(element);
      }
      else{
        maxi.push(element);
      }
  median= (maxi.top() + mini.top()) / 2;
    }
}

int main(){
    vector<int> v {5, 15, 1, 3, 2, 8};
    double median = 0;
    priority_queue <int> maxi;
    priority_queue <int, vector<int>, greater<int>> mini;

    for (int i= 0; i< v.size(); i++){
        int element = v[i];
        medianInAStream(v, median, maxi, mini, element)
        cout << median << " ";
    }
}
```