27/05/2023

Q1 Remove Stones to minimize the total

i/p →

5	8	11	12
0	1	2	3

K = 3

We have to perform operation only k times. The operation that we can perform is reduce the stones by 1/2.
As we have to minimize the total, we need to perform the operation on maximum stones. Hence we need to have max heap.

Dry run

k = 1

Heap →

~~12~~ 6
11
8
5

→ Heapify →

~~11~~ 6
8
6
5

k = 2

Heapify ↓

~~8~~ 4
6
6
5

k = 3

4 + 6 + 5 + 5 = 21 } minimized total

Code

```
int minStoneSum (vector<int>& piles, int k){
    // Creation of max heap
    priority_queue <int> maxHeap;
    // Insert array in max heap
    for (int i=0; i<piles.size(); i++){
            maxHeap.push(piles[i]);
    }
    // Perform operation k times
    while (k--) {
        // Fetch max element
        int topElement = maxHeap.top();
        // Remove from heap
        maxHeap.pop();
        // Perform operation
        topElement = topElement - topElement/2;
        // Push updated element
        maxHeap.push(topElement);
    }
    // Find sum of elements in heap
    int sum =0;
    while (!maxHeap.empty()) {
        int e = maxHeap.top();
        maxHeap.pop();
            sum = sum+e;
    }
    return sum;
}
```

TC = O(n + k log n)

↱ Same pattern as minimum cost of cutting ropes

Q2 Reorganize string. Given a string, rearrange the character s, so that any 2 adjacent characters are not same.

i/p → a a b
o/p → a b a

1) Creating mapping of characters with the frequency.

a → 2
b → 1

2) Now higher frequency characters need to be consumed so that they don't repeat

(a, ~~2~~) 1
(b, 1) 0

o

ans = a
ans = ab
ans = aba

We need to fetch starting 2 characters from heap so as to avoid keeping the same characters adjacent. Once the frequency of character becomes 0, remove from the heap.

Code

```
class node {
    public :
        char data;
        int count;
        node (char d, int c) {
            data = d;
            count = c;
```

}

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

String reorganizeString (string s) {
    // Store frequency of each character
    int freq[26] = {0};
    for (int i=0; i<s.length(); i++){
        char ch = s[i];
        freq[ch - 'a']++;
    }
    // Create a max heap
    priority_queue <node, vector <node>,
    compare > maxHeap;
    // Insert elements in max heap
    for (int i=0; i<26; i++) {
        if (freq[i] != 0) {
            node temp (i + 'a', freq[i]);
            maxHeap.push (temp);
        }
    }
    String ans = "";
    While (maxHeap.size() > 1) {
        // Fetch top 2 elements
        node first = maxHeap.top();
        maxHeap.pop();
        node second = maxHeap.top();
        maxHeap.pop();
```

```
ans += first.data;
ans += second.data;
first.count--;
second.count--;
// Only push in heap if frequency is not 0
if (first.count != 0)
        maxHeap.push(first);
if (second.count != 0)
        maxHeap.push(second);
}
if (maxHeap.size() == 1) {
        node temp = maxHeap.top();
        maxHeap.pop();
        if (temp.count == 1)
                ans = ans + temp.data;
        }
        else { // Here adjacent can come & hence
                ans = ""; return empty string
        }
}
return ans;
}
```

Q3 Longest happy string. (Leetcode 1405)

i/p → a = 1, b = 1, c = 7

o/p → ccaccbcc

We have to find longest possible happy string.

A string is happy if :

(a) S contains letter only a, b and c.

(b) S does not contain any of aaa, bbb or ccc.

(c) S contains atmost a occurences of letter a.

(d) s contains atmost b occurences of 'b'.
(e) s contains atmost c occurences of 'c'.

Code

```
string longestDiverseString (int a, int b, int c){
  // Create a max heap
  priority_queue <node, vector<node>, compare>
  maxHeap;
  // Insert enteries in max heap
  if (a>0) { node temp ('a', a);
      maxHeap.push (temp);
  }
  if (b>0) {
      node temp ('b', b);
      maxHeap.push (temp);
  }
  if (c>0) {
      node temp ('c', c);
      maxHeap.push (temp);
  }
  string ans = " ";
  // Processing in heap
  while (maxHeap.size() >1) {
      node first = maxHeap.top();
      maxHeap.pop();
      node second = maxHeap.top();
      maxHeap.pop();
      // Try to get 2 characters (same pattern)
      if (first.count >=2) {
          ans += first.data;
          ans += first.data;
```

```
        first.count -= 2;
    }
    else { // Count is 1 for sure
        ans += first.data;
        first.count--;
    }
    // 2 characters from second          Important
    if (second.count >= 2 && second.count >= first.
                                                count){

        ans += second.data;
        ans += second.data;
        second.count -= 2;
    }
    // Count is 1 for sure
    else {
        ans += second.data;
        second.count--;
    }
    // Push in heap if count is greater than 0.
    if (first.count > 0)
        maxHeap.push(first);
    if (second.count > 0)
        maxHeap.push(second);
} // Came out of while loop & hence size would be 1
if (maxHeap.size() == 1) {
    node temp = maxHeap.top();
    maxHeap.pop();
    // 3 consecutive characters can't be inserted
    if (temp.count >= 2) {
        ans += temp.data;
        ans += temp.data;
        temp.count -= 2;
    }
```

```
        //Only 1 character left
        else {
            ans += temp.data;
            temp.count--;
        }
    }
    return ans;
}
```

↱ This can happen as first was decremented

Why second.count >= first.count condition was added ?

a = 0 , b = 8 , c = 11

* c = 11
b = 8

c c b b

* c = 9
b = 6

c c b b c c b b

* c = 7
b = 4

c c b b c c b b c c b b

* c = 5
b = 2

c c b b c c b b c c b b c c b b

* c = 3
b = 0 (not in heap)

c c b b c c b b c c b b c c b b c c } not the longest

* c = 1
b = 0

The longest happy string possible is

cc(b)cc b b c c bb cc b b cc b c } All c consumed
↳ Better to insert single b as it would lead to longer string.

Q4 Median in a stream. (Leetcode 295)
Median can be defined as the middle element in a sorted stream.

2, 3, 5, (10), 11, 12, 13
↳ median = 10

2, 3, (5, 10), 11, 12
↳ median = $\frac{5+10}{2}$ = 7.5

But in the question, the stream is not sorted.

Approach-1
1) Create a vector.
2) As a new element arrives, push in vector.
3) Sort the vector.
4) Return the middle element.

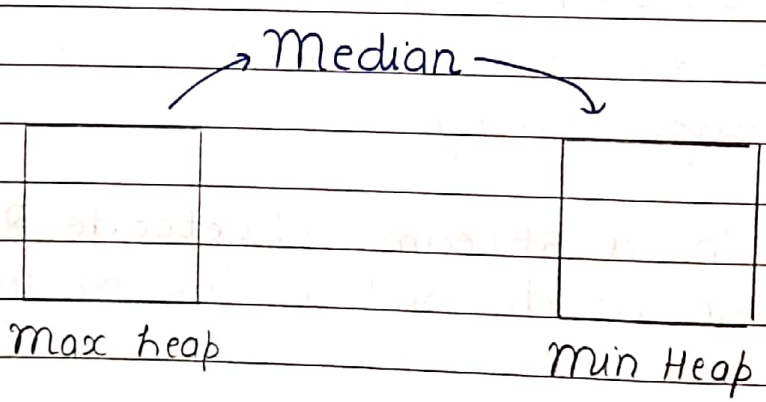
This is very bad approach.

Approach - 2

1) Create a BST of incoming elements.
2) Find inorder traversal as it is sorted.
3) Now return the middle element in the inorder traversal.

Approach - 3

In this we will be using both max heap and min heap.



First we have used max-heap as elements will be pushed here & just after the top element, median will be lying.

Cases

1) Both heaps have same size.

This means even number of elements and hence

$$\text{median} = \frac{(\text{top})\text{min} + (\text{top})\text{max}}{2}$$

2) Min-heap has size 1 greater than max-heap

and hence median lies at top of the min heap.

3) Min-heap has size 1 lesser than max-heap and hence median is the top element of the max-heap

Element to be inserted in which heap?
This will be dependent on the median. If new element is greater than median, then it will be inserted in min-heap else in max-heap.

1) Initially both heaps were having same size. Now after inserting new element either max-heap has size 1 greater than min-heap or vice versa which we have seen in cases 2 or 3.

size =

2) Now if min-heap has size + 1 and max-heap has size = size. Now if new element is greater than median & hence we need to insert in min-heap but it's size would become size = size + 2 and we have not discussed this case. So we need to remove one element from min-heap and insert in max-heap and then insert in min-heap. Now both will have size = size + 1 and this is case 1.

$$\text{new-median} = \frac{(\text{top})\text{min} + (\text{top})\text{max}}{2}$$

Here if new element < median, then simply insert in max-heap leading to equal size i.e case 1.

3) Similarly we can handle the case when max-heap has size = size + 1 & new element < median. When new element > median, simply insert in min-heap.

Code

```
int signum (int a, int b) {
    if (a == b)
        return 0;
    if (a > b)
        return 1;
    return -1;
}

void callMedian (double & median,
priority_queue <int> & maxHeap,
priority_queue <int, vector <int>,
greater <int>> & minHeap, int element) {
    switch (signum (minHeap.size(),
    maxHeap.size())) {
    // Both heaps have equal size
    case 0:
        if (element > median) {
            // Push in min heap
            minHeap.push (element);
            median = minHeap.top();
        }
        else { // Push in max heap
```

```
        maxHeap.push(element);
        median = maxHeap.top();
    }
    break;
    case 1: //minHeap has larger size
        if(element > median){
            //Remove from minHeap & push to maxHeap
            int minTop = minHeap.top();
            minHeap.pop();
            maxHeap.push(minTop);
            //Push new element to min heap
            minHeap.push(element);
            //Both heaps have equal size
            median = ((maxHeap.top())+(minHeap.top())/2.0
        }
        else{
            //Push in max heap
            maxHeap.push(element);
            median = ((maxHeap.top())+(minHeap.top())/2.0;
    }
    break;
    case -1: //max heap has larger size
        if(element > median){
            //Push in min heap
            minHeap.push(element);
            median = ((maxHeap.top())+(minHeap.top())/2.0;
        }
        else{
            //Remove from max heap & push to min heap
            int maxTop = maxHeap.top();
            maxHeap.pop();
            //Push new element to max heap
```

```
            maxHeap.push(element);
            // Both heaps have equal size now
            median = ((maxHeap.top()) + (minHeap.top()))/2.0;
        }
        break;
    }
}
```