1/05/2023

Q1 First non-repeating character in a stream.

i/p → a a b c
o/p → a # b b

{a} → a
{a, a} → # (Demond of question)
{a, a, b} → b
{a, a, b, c} → b

Here we need to store the count / frequency of each character. This can be done via maintaining a count array or using map. Here will be using the count array.
Non-repeating → Can be concluded with the help of count array.
Here we have to find the first non-repeating

character and for this we need to keep the track.

Dry run / Algorithm
i/p → a a b c

1) We are at 0th index and arr [0] is a, first thing we need to increase the count in the count array. Second thing we need to do is push in the queue.

2) Check q. front () now ÷
  (i) Repeating → simply pop
  (ii) Non-repeating → Store in the ans.
  (iii) Empty queue → Store # in the ans.

Code

```
void solve (string &s){
    //Count array → 26 size as there are 26 alphabet
    int count [26] = {0};
    queue <char> q;
    string ans = " ";
    //Traverse whole string (i/p)
    for (int i=0 ; i<s.length () ; i++){
        char ch = s [i];
        //Increment frequency
        count [ch - 'a'] ++;
        //Push in queue
        q.push (ch);
        while (!q.empty()) {
            //Repeating then pop
```

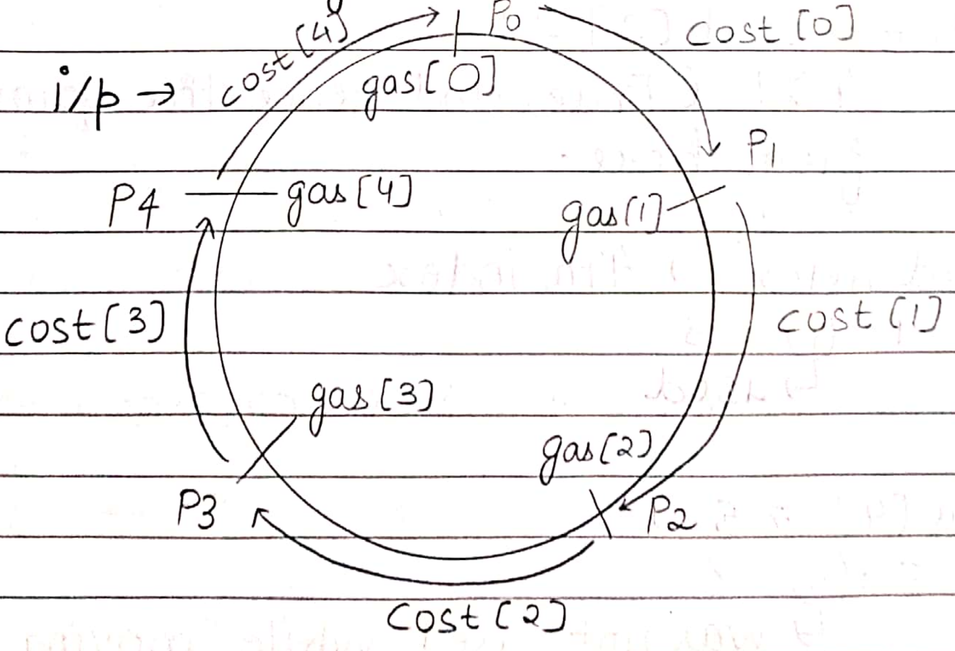
```
                    ↳ Typecasting
            if (count [q.front() - 'a'] > 1) {
                    q.pop();
            }
            // Non-repeating character found.
            else {
                ans.push_back (q.front());
                break; // Ans found & hence break
            }
        }
        // Non-repeating character not found & q was empty
        if (q.empty()){
            ans.push_back ("#"); // Demand of
        }                               question.
        cout << ans ;
}
```

Time complexity = O(n)
Space complexity = O(n)

↳ Very important question (Coding test / Interview)

Q2 Circular tour / Gas station. (Leetcode 134)

i/p →



O/p → Starting index such that circular route is completed.

gas →	1	2	3	4	5
cost →	3	4	5	1	2

Brute force

1) gas [0] = 1 (balance)
dist = cost [0] = 3
1 > 3 (False) & hence 0th index is not the answer.

2) gas [1] = 2 (balance)
dist = cost [1] = 4
2 > 4 (False) & hence 1st index is not the answer.

3) gas [2] = 3 (balance)
dist = cost [2] = 5
3 > 5 (False) & hence 2nd index is not the answer.

4) gas [3] = 4 (balance)
dist = cost [3] = 1
4 > 1 (True) and hence the game starts from here.

3rd index → 4th index

$4 - \boxed{1} = 3$

↳ used

5) gas [4] = 5

$5 + \boxed{3} = 8$

↳ was not used while moving from 3rd to 4th index cost [4] = 2

8 > 2 (True)

6) dist = cost [0] = 3

gas [0] = 1

$1 + \boxed{6} = 7$

↳ not used while moving from 4th - 0th index

7 > 3 (True)

7) dist = cost [1] = 4

gas [1] = 2

$2 + \boxed{4} = 6$

↳ not used while moving from 0th - 1st index

6 > 4 (True)

8) dist = cost [2] = 5

gas [2] = 3

$\boxed{2} + 3 = 5$

↳ not used while moving from 1st - 2nd index

5 >= 5 (True)

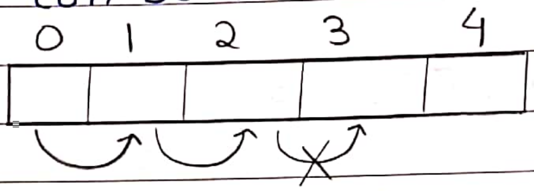
Hence we have reached the starting point & hence the answer = 3

Time complexity = $O(n^2)$

Better approach :-

gas →	1	2	3	4	5
cost →	3	4	5	1	2

The optimization that we can do in the previous approach can be :-



When we can't reach from 0^{th} index to 3rd index, then we don't have to check from 1st/2nd index to 3rd index as the contribution from previous petrol pump will be greater than or equal to 0 & won't be negative.

Use of queue

1) If we can go ahead, then move rear forward.
2) If movement not possible, simply make front = rear + 1 and then = front.

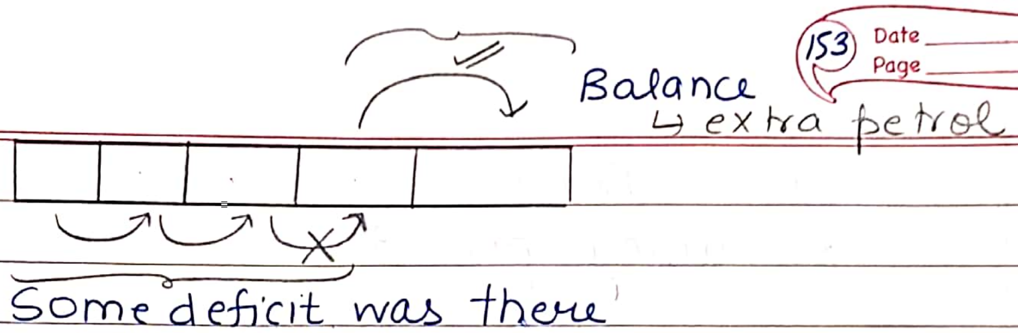
When front becomes equal to rear, then circular tour has been finished.

Better Approach - 2

gas → {1, 2, 3, 4, 5}
cost → {3, 4, 5, 1, 2}

When we can't reach a particular index, this means that petrol was less which means there is a deficit.

$$\text{deficit} = \text{abs}(\underset{\text{petrol}}{\boxed{p}} - \overset{\text{distance}}{\boxed{d}});$$



If balance >= deficit, answer will come and hence circular tour will be done.

Dry run

gas → {4, 6, 3, 4, 8}
cost → {3, 6, 7, 1, 3}

1) gas [0] = 4
cost [0] = 3
4 > 3 (True)
Remaining gas = 1

2) gas [1] = 6
6 + $\textcircled{1}$ = 7
↳ Remaining which we got from 1st
cost [1] = 6
7 > 6 (True)
Remaining gas = 1

3) gas [2] = 3
3 + $\textcircled{1}$ = 4
↳ Remaining which we got from 2nd
cost [2] = 7
4 > 7 (False)
Deficit = 7 − 4 = 3 } Short of 3 units of gas

4) Now we have to start again from 3rd index as discussed in better - approach 1.
gas [3] = 4
cost [3] = 1

4 > 1 (True)
Remaining gas = 3

5) cost[4] = 3
gas [4] = 8
8 + 3 = 11
↳ Remaining which we got from 4th
11 > 3 (True)
Remaining = 8 (Balance)

Now Balance >= deficit is true & hence we will surely complete circular tour.

Here in approach-2 (better approach), we eliminated going into the circular thing by maintaining the deficit.

Code

```
int solve (vector <int> &gas, vector <int>& cost)
{
        //Shortage of petrol
        int deficit = 0;
        //How much petrol is left
        int balance = 0;
        //Starting index
        int start = 0;
        //Traverse the gas array
        for (int i = 0 ; i < gas.size() ; i++){
            balance = balance + gas[i] - cost[i];
            // Balance is negative
            if(balance < 0     ){
```

→ Here we can do mistake

```
        //Consider all deficit
        deficit = deficit + abs(balance);
        //Better approach - 1
        Start = i+1;
        //Start again
        balance = 0;
      }
   }
   //Ans found
   if (balance >= deficit){
          return start;
   }
   //Answer not found
   return -1;
}
```

Q3 Sliding window maximum. In a window of size k, we have to find the maximum number in that window.

i/p → {1, 3, -1, -3, 5, 3, 6, 7}, k = 3
o/p → {3, 3, 5, 5, 6, 7}

The pattern of the question will be same as that of first negative integer in every window of size k.

1) Create a queue
2) Process 1st window of size k

{1, 3, -1}
(i) simply insert 0 into queue as it is empty.
↳ index of 1

(ii) Now we have 3 in the array and it is greater than element whose index is at q.front(). Hence simply remove 0 and insert index = 1

(iii) Now we have -1 in the array and it is not greater than element whose index is at q.front() but still push index of -1 as it might be possible answer for next windows.

Queue →	1	2	

3) Remove out of window elements for the next window we are going to process.

  1   2   3
{3, -1, -3}

No out of window in the queue & hence no need to pop.

Note → Whenever we a bigger element, then we pop out all the elements which are smaller than the element we are processing. This means on the left of the element there will be indexes of those elements greater than the current element we are processing. Hence we need to pop from the back as we need to find maximum element.

Code

```
void solve (vector <int> &nums, int k){
        deque<int>dq;
        vector <int> ans;
        // Process first window
        for (int i=0 ; i< k ; i++){
Difficult      //Remove smaller than current element
to understand  while (! dq.empty() && nums[i] > =
                        nums [dq.back ()]) {
                    dq.pop_back ();
            }
          //Push index so that we can process out of window
          dq.push_back (i);
    }
    //Store answer of 1st window of size k
    ans.push_back (nums [dq.front()]);
    // Remaining window
    for (int i = k ; i<nums.size () ; i++) {
Difficult      //Delete out of window from front
to understand  if (! dq.empty () && i-dq.front () > =k) {
                        dq.pop_front();
            }
          //Remove smaller than current element
          while (! dq.empty() && nums[i] > =nums [dq.back()]
          {
                dq.pop_back();
          }
          //Insert index → To detect out of window
          dq.push_back (i);
          // Store answer of current window
          ans.push - back (nums [dq.front()]);
    }
    //Print ans vector    for(auto:i) { cout << i<<" ";}
}
```