****

**Demux Summer Hackathon -6**

Solution

<https://www.hackerrank.com/demux-summer-hackathon-6>

**1.Find the Running Median**

This problem can be solved using two **heaps**.

Let's say we are taking input for the  **ith** number. If somehow we had the previous (**i-1)**  numbers sorted, then we can easily add the **ith**  number in  0(n) time to the appropriate place in our list so that the list remains sorted and find the median in 0(1 ) time. But we cannot afford to add every number with 0(n) complexity.

So let's say we had two sorted arrays. The first array holds the smaller half of the numbers in decreasing order. The second array contains the larger half of the numbers in increasing order. Now, after taking input for the **ith**  number, we can easily decide which half the **ith** number belongs to and add it there in the appropriate place. If any of the arrays becomes much larger than the other array, we can remove the first element from that array and add it to the appropriate place of the other array. We have reduced time complexity by some constant factor. But obviously, that is not enough.

Now let's look at the indices of the array where we need to access at any moment for getting the median. If the total number of elements is odd, then we need the first element of the array with the higher number of elements. If the total number of elements is even, then we need the average of the first elements of both the arrays. So the only indices we need to access while getting the median and adding the elements are the first index of both of the arrays. So which data structure can help in storing data in sorted order and accessing the top element efficiently? The answer is a **heap**.

We will use a **max heap** for storing data of the smaller half of the numbers and a **min heap** for storing data of the larger half of the numbers. Now let's see how we will add the numbers and get the medians.

# Adding the Ith number:

While adding the **ith** number we will check the following conditions and add accordingly:

1. If the **ith** number is greater than or equal to the max element of the max heap then it surely belongs to the larger half of the numbers i.e. the min heap. So we will add it there.
2. Otherwise, we will add it to the max heap.

Now it may happen that one of the heap becomes much larger than the other if we add the numbers in this way. So to stop this situation from taking place, we need to check the size of the heaps after adding every number. If the difference between the number of elements of the two heaps becomes more than one, then we need to pop the top element from the heap with more elements and push that element to the other heap. If we work in this way, the difference can never be more than one.

# Getting the median:

To get the median after adding the **ith** number we will check if i is odd or even. If i is odd, then surely one of the heap has one more element than the other. The median will be the top element of that heap then. If **i** is even, then the two heaps must have the same number of elements. So the median will be the average of the top elements of the two heaps.

#include<bits/stdc++.h>

using namespace std;

priority\_queue<int, vector<int>, greater <int> > min\_heap;

priority\_queue<int> max\_heap;

void add(int a)

{

if( max\_heap.size() && a >= max\_heap.top())

min\_heap.push(a);

else

max\_heap.push(a);

if(abs(max\_heap.size() - min\_heap.size()) > 1)

{

if(max\_heap.size() > min\_heap.size())

{

int temp = max\_heap.top();

max\_heap.pop();

min\_heap.push(temp);

}

else

{

int temp = min\_heap.top();

min\_heap.pop();

max\_heap.push(temp);

}

}

}

double get\_median()

{

int total = min\_heap.size() + max\_heap.size();

double ret;

if(total%2 == 1)

{

if(max\_heap.size() > min\_heap.size())

ret = max\_heap.top();

else

ret = min\_heap.top();

}

else

{

ret = 0;

if(max\_heap.empty() == false)

ret += max\_heap.top();

if(min\_heap.empty() == false)

ret += min\_heap.top();

ret/=2;

}

return ret;

}

int main()

{

cout << setprecision(1) << fixed;

int n, a;

cin >> n;

for(int i = 1; i<=n; i++)

{

cin >> a;

add(a);

cout << get\_median() << endl;

}

}

# 2. Jesse and Cookies

This problem can be solved using a min heap. Initially, we add all the cookies to the heap. We repeatedly pop**2** cookies with the least sweetness and combine them and add the resulting sweetness (1\* least sweet cookie + 2\* 2nd least sweet cookie) to the heap till the sweetness of minimum becomes **>=K**.

#include<iostream>

#include<vector>

#include<cstdio>

#include<algorithm>

#include<utility>

#include<set>

#include<map>

#include<cstring>

#include<cmath>

#include<string>

#include<cstdlib>

#include<queue>

using namespace std;

int main()

{

#define int long long

int n,k;

cin>>n>>k;

priority\_queue<int, std::vector<int>, std::greater<int> > pq;

for(int i=0;i<n;i++)

{

int val;

cin>>val;

pq.push(val);

}

int count=0;

bool ans=true;

while(1)

{

if(pq.empty())

{

ans=false;

break;

}

int a1=pq.top();

pq.pop();

if(a1>=k)

{

break;

}

if(pq.empty())

{

if(a1<k)

{

ans=false;

}

break;

}

int a2=pq.top();

pq.pop();

int nv=a1+2\*a2;

count++;

pq.push(nv);

}

if(ans)

cout<<count;

else

cout<<"-1";

cout<<endl;

}

# 3. Is This a Binary Search Tree?

### **C++**

bool checkBST(Node\* root, int minValue, int maxValue) {

if (root == NULL) {

return true;

}

if (root->data < minValue || root->data > maxValue) {

return false;

}

return ( checkBST(root->left, minValue, root->data - 1)

&& checkBST(root->right, root->data + 1, maxValue)

);

}

bool checkBST(Node\* root) {

return checkBST(root, 0, 10000);

}

### **Java**

boolean checkBST(Node root, int minValue, int maxValue) {

if (root == null) {

return true;

}

if (root.data < minValue || root.data > maxValue) {

return false;

}

return ( checkBST(root.left, minValue, root.data - 1)

&& checkBST(root.right, root.data + 1, maxValue)

);

}

boolean checkBST(Node root) {

return checkBST(root, 0, 10000);

}

# 4. Tree : Top View

def top\_view(root, m, hd,level):

if not root:

return

if hd in m:

if level < m[hd][1]:

m.update( {hd : [root.info,level] })

else:

m[hd] = [root.info,level]

top\_view(root.left, m, hd-1,level+1)

top\_view(root.right,m, hd+1, level+1)

def topView(root):

m={}

top\_view(root, m, 0,0)

mn = 100000

mx = -100000

for key,value in m.items():

if mx < key:

mx = key

if mn > key:

mn = key

i = mn

while i <= mx:

print (m[i][0],end = " ")

i = i+1

# 5. Sherlock and Anagrams

Two string are anagrams if and only if for every letter occurring in any of them the number of its occurrences is equal in both the strings.

This definition is crucial and will lead to the solution. Since the only allowed letters are lowercase English letters, from a to z, the alphabet size is constant and its size is 26 . This allows us to assign a constant size signature to each of the substring of **s** .

A signature of some string **w** will be a tuple of  26 elements where the **i-th** element denotes the number of occurrences of the i-th letter of the alphabet in w.

So, for example, if w = ”mom” then its signature is [0, 0, 0, 0, 0,0, 0, 0, 0,0, 0, 0, 2,0, 1, 0,0, 0, 0,0, 0, 0, 0, 0, 0, 0], so the only non-zero elements are the ones corresponding to letter m with value of 2 and letter o with value of 1.

Notice, that any string that is an anagram of ”mom” will have the same signature as "mom”, and every string that is not an anagram of ”mom” will definitely have a different signature.

This concept of signatures allows the following approach.

Let's iterate over all substrings of s and for each fixed substring let's compute its signature and add that signature to signatures hashmap, where signatures [sig] denotes the number of substrings of s with a signature sig.

Finally, the only remaining thing to do is to get the number of pairs of substrings of s that are anagrams. It's easy to do having our hashmap. Notice that if there are n substrings of s with signature sig, then they can formn- (n — 1)/2 pairs of substrings with signature sig, so we can just iterate over all values in the hashmap and for each value n add n .(n — 1)/2 to the final result.

The below, commented code, in Python, illustrates this exact approach.

The time complexity is O(|s|3) since we iterate over all O(|s|2) substrings of s and for each substring we compute its

signature in O(|s|) time. It's worth to mention that each operation on hashmap has constant running time since our signatures have a constant size, i.e. 26 which is the size of our alphabet. Otherwise, if the alphabet size is not constant, this approach will have O(|s|3 - ALPHABET \_SIZE) time complexity.

**Code:**

import string

q = int(raw\_input())

ALPHABET = string.ascii\_lowercase

for \_ in xrange(q):

s = raw\_input()

signatures = {}

signature = [0 for \_ in ALPHABET]

for letter in s:

signature[ord(letter)-ord(ALPHABET[0])] += 1

# iterate over all substrings of s

for start in range(len(s)):

for finish in range(start, len(s)):

# initialize substring signature

signature = [0 for \_ in ALPHABET]

for letter in s[start:finish+1]:

signature[ord(letter)-ord(ALPHABET[0])] += 1

# tuples are hashable in contrast to lists

signature = tuple(signature)

signatures[signature] = signatures.get(signature, 0) + 1

res = 0

for count in signatures.values():

res += count\*(count-1)/2

print res

# 6. Substring Searching

**Scala:**

object Solution {

def compute(pat: Array[Char], M: Int, TF: Array[Array[Int]]) {

var i: Int = 0

var lps = 0

var x: Int = 0

x = 0

while (x < 256) {

TF(0)(x) = 0

x = x + 1

}

TF(0)(pat(0)) = 1

i = 1

while (i < M) {

x = 0

while (x < 256)

{

TF(i)(x) = TF(lps)(x)

x = x + 1

}

TF(i)(pat(i)) = i + 1

if (i < M)

lps = TF(lps)(pat(i))

i = i + 1

}

}

def kmp(pat: Array[Char], txt: Array[Char]) {

val M = pat.length()

val N = txt.length()

var flag = 0

val TF = Array.ofDim[Int](M + 1, 256)

compute(pat, M, TF)

var i: Int = 0

var j = 0

i = 0

while (i < N) {

j = TF(j)(txt(i))

if (j == M) {

flag = 1

//break

}

i = i + 1

}

if (flag == 1) println("YES") else println("NO")

}

def main(args: Array[String])

{

var test: Int = 0

test = readInt()

//var txt = Array.ofDim[Char](100001)

//var pat = Array.ofDim[Char](100001)

while (test > 0)

{

val str1 = readLine()

val str2 = readLine()

var txt = str1.toCharArray()

var pat = str2.toCharArray()

kmp(pat, txt)

test = test - 1

}

}

}