INSAT ACM Problem Solving Community

Vote: I like it **0** Vote: I do not like it

Private

Spectator

Add to favourites

→ About Group

Here we share all the competitions that were held by ACM INSAT. Either training contests or mirrors of official national contests held by ACM INSAT Club like CodeQuest and Winter Cup

---

Group website

→ Group ratings

- Insat CPC Season 2017 - 2018
- Insat CPC Season 2022-2023

→ Member management

You are not group member yet,

but can request group join.

Membership type: Participant

Join

CodeQuest 1.0 Editorial

By LorenzoRaed02, history, 12 months ago, In English

A. Stolen necklace

Solution

Explanation:

The number of potential thieves is the number of persons for whom there is no logical argument to be made to prove their innocence.

The key observation in this problem is that there will be a transition from a present necklace to an absent one **exactly one time** because the necklace was stolen only once, and no one can lie other than the thief.

We can conclude that the people we suspect represent a **substring** of the input string.

So, we need to find where (in the string) the crime was committed. Since we can also encounter "?", we find the index of the **leftmost** 0 (in case of absence, we take n) and mark it as f0, and the index of the **rightmost** 1 (in case of absence, we take -1) and mark it as l1.

Solution:

The number of indices between them (inclusive), because only one could be the thief. *f0 — l1 + 1*

- In fact, the thief cannot be to the left of l1, since the person of index l1 would then be lying which leads to a contradiction.
- Similarly, the thief cannot be to the right of f0, since the person of index f0 would then be lying which leads to a contradiction.
- The people between l1 and f0 are '?' which means that the real transition happened somewhere in that substring (inclusive) which means that we can suspect any one of them

C++ Solution
Java Solution
Python Solution

```
t = int(input())

for _ in range(t):
    n=int(input())
    s = input()
    ans = 0
    last_one = -1
    first_zero = n

    for i in range(n):
        if s[i] == '1':
            last_one = i
        if s[i] == '0':
            first_zero = i
            break

    if last_one != -1 and first_zero != n:
        ans = first_zero - last_one + 1
    elif last_one == -1 and first_zero == n:
        ans = first_zero
    else:
        ans = first_zero - last_one

    print(ans)
```

B. Luffy is a Foodie

Solution
Complexity
C++ Solution
Java Solution
Python Solution

C. Currensea

Solution

Key Observation :

Let f(n) be the function that associates a number n to the value 1111... (n times) For example, f(7) = 1111111

We can prove (using strong induction) that for all $(n \geq 3)$, there exists $a \geq 0$ and $b \geq 0$ such that $f(n) = af(2) + bf(3) = 11a + 111b$

Defining the Problem:

The problem is thus reduced to the following problem: can n be written:

$$n = 11a + 111b$$

where $a, b \geq 0$

Solutions: (all derive from this observation)

All the following solutions are based on this key observation

- **Solution 1: iterate over b**:
  - Since the maximum value for n is 10^9 we can safely iterate over the possible values of b and check if $(n - 111b)$ is divisible by 11.
  - This solution only works given the low constraints
- **Solution 2: extended Euclidean algorithm**:
  - By using the extended Euclidean algorithm, we calculate the solutions for the following equation $11x + 111y = 1$
  - The solutions are $x = -10 + 111k$ and $y = 1 - 11k$ where k is an integer
  - We can then solve for the equation $11x + 111y = n$
  - The solutions are $x = -10n + 111k$ and $y = n - 11k$
  - We need $x, y \geq 0$. The maximum value k can take while maintaining y positive is $n/11$
  - If x is also positive for this solution then an answer exists
- **Solution 3: euclidean division:**
  - $n = 11a + 111b = 11a + 111(11q + r) = 11(a + 111q) + 111r$ where $(0 \leq r \leq 10)$
  - We iterate over the possible values of r and check if $n - 111b$ is divisible by 11

Read more about the extended Euclidean algorithm

C++ Solution 1
C++ Solution 2
Java Solution
Python Solution

```
t = int(input())

for _ in range(t):
    n = int(input())
    test = False

    for r in range(11):
        if (n - 111 * r) >= 0 and (n - 111 * r) % 11 == 0:
            test = True
            break
    if test:
        print("YES")
    else:
        print("NO")
```

D. Celebration

Solution

Linus Torvalds: “Talk is cheap. Show me the code.”

Check the Code to understand the solution

C++ Solution
Java Solution
Python Solution

```
t = int(input())
for _ in range(t):
    x, y, k = map(int, input().split())
    res = 0
    if (k + y - 1) % x == 0:
        res = (k + y - 1) // x
    else:
        res = (k + y - 1) // x + 1
    print(res)
```

E. Bounty Challenge

Solution

Annotation :

Let's denote by $f(B)$ the value of the bounty after simulating the fights using an initial bounty of B

Brute Force Approach :

Linearly searching for the answer will have a time complexity of $O(n * L)$, which will yield TLE given the constraints

Solution :

Since for $B_1 < B_2$ we have $f(B_1) < f(B_2)$: in other words, f is a monotonous function, we can instead use binary search to look for the maximum initial bounty value * Our search space is $[0, L]$ * The check we perform in our binary search consists of simulating the process of fights * If $f(B) > L$, the solution can only figure in the $[\text{left}, \text{mid} - 1]$ space * If $f(B) \leq L$, the solution is valid but we look for a better solution in the $[\text{mid} + 1, \text{right}]$ space since we're looking to maximize the initial bounty B

Time Complexity :

The check is performed with time complexity of $O(n)$ => the overall time complexity of our algorithm is thus $O(n * \log(L))$

C++ Solution
Java Solution
Python Solution

```
n, l = map(int, input().split())
fights = list(map(int, input().split()))

min_bounty, max_bounty, ans = 0, l, 0
```

```
while min_bounty <= max_bounty:
    current_bounty = (min_bounty + max_bounty) // 2
    current_sum = 0

    for i in range(n):
        current_sum += 100 if current_bounty + current_sum > fights[i] else 0
        if current_bounty + current_sum > l:
            break

    if current_bounty + current_sum > l:
        max_bounty = current_bounty - 1
    else:
        ans = current_bounty
        min_bounty = current_bounty + 1

print(ans)
```

F. Blood Transfusion

Solution

The determining factor of the solution is the occurrence of each blood type in the two arrays so we should start by calculating that.

We know that injured people of blood type O can only receive from O donors

Intuitively, we can safely start by saving the first category of injured people: people with blood type O. If injuredO > donorO then we know that it's impossible to save all the injured people.

We update donorO accordingly.

We know that injured people of blood type A can receive from blood type A donors as well as blood type O donors Symmetrically, injured people of blood type B can receive from blood type B donors as well as blood type O donors

Starting by saving people with blood type A or people of blood type B does not matter. (Try all the possible cases)

We stop if the value of (donorO + donorA) < injuredA or if (donorO + donorB) < injuredB

Note: don't forget always to update the value of donorO

We know that people of blood type AB can receive from any donor

If O, A, B injured people are saved then the AB injured people will be saved since size(injured) = size(donors) and every donor left can donate to the AB injured people.

C++ Solution
Java Solution
Python Solution

```
t = int(input())
for _ in range(t):
    n = int(input())

    # the arrays count the respective occurrence of A, B, AB, and O
    injured = [0, 0, 0, 0]
    donors = [0, 0, 0, 0]

    injured_input = input().split()
    for s in injured_input:
        if s == "A":
            injured[0] += 1
        elif s == "B":
            injured[1] += 1
        elif s == "AB":
            injured[2] += 1
        elif s == "O":
            injured[3] += 1
    donors_input = input().split()
    for s in donors_input:
        if s == "A":
            donors[0] += 1
        elif s == "B":
            donors[1] += 1
        elif s == "AB":
            donors[2] += 1
        elif s == "O":
            donors[3] += 1

    # check if we can satisfy the injured of blood type O
    if injured[3] <= donors[3]:
        donors[3] -= injured[3]
    else:
        print("NO")
        continue

    # check if we can satisfy the injured of blood type B
    if injured[1] <= donors[1] + donors[3]:
        temp = donors[1]
        donors[1] = max(donors[1] - injured[1], 0)
        injured[1] = max(injured[1] - temp, 0)
        donors[3] = donors[3] - injured[1]
    else:
        print("NO")
        continue

    # check if we can satisfy the injured of blood type A
    if injured[0] <= donors[0] + donors[3]:
        temp = donors[0]
        donors[0] = max(donors[0] - injured[0], 0)
        injured[0] = max(injured[0] - temp, 0)
        donors[3] = donors[3] - injured[0]
    else:
        print("NO")
        continue

    # if A, B, O are all satisfied, then AB is satisfied
    print("YES")
```

G. Blood Transfusion (Hard version)

Solution

- The problem can be modelled using a **bipartite** graph:
  - The nodes represent the people
  - The first set of nodes represents the injured people
  - The second set of nodes represents the donors
  - Node I from the first set and D from the second edge have an edge between them means that D can donate to I.
- **First step: building the graph:**
  - A donor can donate blood to a person of the same blood type: for each donor and injured person of the same blood type, we add an edge
  - For each relation between blood type **x** and blood type **y**, we add an edge between all the donor nodes of blood type **x** and all the injured of blood type **y**.

- **Second step: apply a maximum bipartite matching algorithm:**

  - A **matching** in a Bipartite Graph is a set of the edges chosen in such a way that **no two edges share an endpoint**. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.
  - We use the **Hopcroft-Karp Algorithm** to find the maximum matching. This algorithm works by finding **augmenting paths** (paths used to maximise the current matching) until there are no more augmenting paths.
  - This article goes into depth about Bipartite Graphs
  - These two articles go into depth about the Hopcroft-Karp Algorithm and its implementation

  - The **time complexity** for this algorithm is $O(\sqrt{V} . E)$ where V is the number of nodes of our graph and E denotes the number of edges
  - This article goes into depth about another algorithm for finding the maximum bipartite matching by converting the problem to a maximum flow problem and then using the Ford-Fulkerson Algorithm
  - The **time complexity** for this algorithm is $O(V . E)$
- **Third step: finding the answer**
  - Given the maximum matching value (how many pairs of edges the maximum matching includes), if this value is equal to our n, then we can save everyone otherwise we can't

C++ Solution
Java Solution
Python Solution

```
from collections import deque


class BipGraph:
    NIL = 0
    INF = float('inf')

    def __init__(self, m, n):
        self.m = m
        self.n = n
        self.adj = [[] for _ in range(m + 1)]

    def add_edge(self, u, v):
        self.adj[u].append(v)

    def hopcroft_karp(self):
        self.pairU = [self.NIL] * (self.m + 1)
        self.pairV = [self.NIL] * (self.n + 1)
        self.dist = [0] * (self.m + 1)

        result = 0
        while self.bfs():
            for u in range(1, self.m + 1):
                if self.pairU[u] == self.NIL and self.dfs(u):
                    result += 1
        return result

    def bfs(self):
        Q = deque()

        for u in range(1, self.m + 1):
            if self.pairU[u] == self.NIL:
                self.dist[u] = 0
                Q.append(u)
            else:
                self.dist[u] = self.INF

        self.dist[self.NIL] = self.INF

        while Q:
            u = Q.popleft()

            if self.dist[u] < self.dist[self.NIL]:
                for v in self.adj[u]:
                    if self.dist[self.pairV[v]] == self.INF:
                        self.dist[self.pairV[v]] = self.dist[u] + 1
                        Q.append(self.pairV[v])

        return self.dist[self.NIL] != self.INF

    def dfs(self, u):
        if u != self.NIL:
            for v in self.adj[u]:
                if self.dist[self.pairV[v]] == self.dist[u] + 1:
                    if self.dfs(self.pairV[v]):
                        self.pairV[v] = u
                        self.pairU[u] = v
                        return True
            self.dist[u] = self.INF
            return False
        return True


if __name__ == "__main__":
    n, m = map(int, input().split())

    inj = [[] for _ in range(26)]
    don = [[] for _ in range(26)]

    x = input().split()
    for i in range(len(x)):
        inj[ord(x[i]) - ord('A')].append(i + 1)

    x = input().split()
    for i in range(len(x)):
        don[ord(x[i]) - ord('A')].append(i + 1)

    g = BipGraph(n, n)

    for i in range(m):
        x, y = input().split()
        ia = ord(y) - ord('A')
        ib = ord(x) - ord('A')

        for a in inj[ia]:
            for b in don[ib]:
                g.add_edge(a, b)

    for i in range(26):
        for a in inj[i]:
            for b in don[i]:
                g.add_edge(a, b)

    if g.hopcroft_karp() == n:
        print("YES")
    else:
        print("NO")
```

H. Clone Fruit

Solution

Annotations:

Let's suppose that initially, we have 4 people denoted by A, B, C, and D.

Brute Force Approach (TLE):

Simulating the process has a time complexity of $O(n)$ which will yield a TLE verdict given the constraints of our problem

Solution :

The sequence of people eating the fruits will be like the following ABCD-AABBCCDD-AAAABBBBCCCCDDDD-.....

We separated each **section** by a — to illustrate the solution more easily.

The regularity is clear: each section's size is double that of the previous section.

We first need to identify which section the n-th fruit will land on.

To do this we initialize a variable `curr` denoting the current fruit number and `nbr` denoting how many times each name is repeated in the current section.

We start adding whole sections $(m, 2m, 4m. . . .)$ until $(curr > n)$.

This operation costs $O(\log n)$ time complexity.

After identifying the section, we just need to find the name of the person at the n-th index. We calculate the index of the fruit in the last section `index`. Knowing how many times each name is repeated in the current section, we can easily deduce the answer in constant time.

C++ Solution

Java Solution

Python Solution

```
n = int(input())
nbM = int(input())
s = [input() for _ in range(nbM)]

i = 0
a = nbM
sum_val = 0

while (sum_val + a) <= n:
    sum_val += a
    a = a * 2
    i += 1

b = n - sum_val
x = 2 ** i
j = 0

if b != 0:
    j = b // x
    if b % x != 0:
        j += 1
    print(s[j - 1])
else:
    print(s[nbM - 1])
```

I. Hidden Treasures

Solution

Problem I — Hidden treasures

Brute Force Approach (TLE) :

In this problem, calculating the sum of integers in the upper part of the diagonal and the sum of integers in the lower part of the diagonal for each query has a time complexity of $O(\min(n, m).\,q)$ and will yield a TLE verdict.

Solution :

For this purpose, to solve the problem, it is required to precompute the prefix sum for each diagonal using the formula:

$$\text{prefixSum}[i][j] = \text{matrix}[i][j] + \text{prefixSum}[i-1][j-1]$$

Each diagonal in the newly created matrix is a prefix sum of the elements on that diagonal

Doing this **precomputation** will reduce the complexity of answering each query from a linear time complexity to a constant one $O(1)$

The answer for each query (**x**, **y**) will be a comparison between $\text{prefixSum}[x-1][y-1]$: **sum of elements above our element** and
$\text{prefixSum}[\min(n-x, m-y)+x][\min(n-x, m-y)+y] - \text{prefixSum}[x][y]$: **sum of elements below our element**

Note:

min(n — x, m — y) + x and **min(n — x, m — y) + y** denote the position of the last element in the diagonal associated to **(x, y)**.

Complexity:

Complexity=O(q + n.m)
C++ Solution
Java Solution
Python Solution

```
import sys

n, m, q = map(int, input().split())

matrix = [[0] * 2002 for _ in range(2002)]
prefixSum = [[0] * 2002 for _ in range(2002)]

for i in range(1, n + 1):
    matrix[i][1:m+1] = map(int, sys.stdin.readline().split())

for i in range(1, n + 1):
    for j in range(1, m + 1):
        prefixSum[i][j] = matrix[i][j] + prefixSum[i - 1][j - 1]

for _ in range(q):
    x, y = map(int, sys.stdin.readline().split())
    if prefixSum[x - 1][y - 1] == prefixSum[min(n - x, m - y) + x][min(n - x, m - y) + y] - prefixSum[x][y
        sys.stdout.write("YES\n")
    else:
        sys.stdout.write("NO\n")
```

J. The Turn Game

Solution

Solution:

A key observation is that the first player who has a turn in which $a_i = b_i$ wins because he will choose the move that will give him the turn in the next choice (the next index in which $a_i = b_i$). The player will repeat the process until he chooses the move that lets him have the last turn and wins.

We can just simulate the game and keep track of who has the turn. When we encounter the first $a_i = b_i$ we can just print the name of the player who has the i-th turn.

C++ Solution
Java Solution
Python Solution

```
n = int(input())
a = input()
b = input()
ai, bi = 0, 0
turn = True

for i in range(n):
    ai = int(a[i])
    bi = int(b[i])
    if ai < bi:
        turn = not turn
    elif ai == bi:
        break

print("Luffy" if turn else "Zoro")
```

K. Sanji The Gentleman

Solution

Linus Torvalds: “Talk is cheap. Show me the code.”

Check the Code to understand the solution
C++ Solution
Java Solution
Python Solution

```
N = int(input())
s = input()
msg = "NO"

for char in s:
    if char == 'W':
        msg = "YES"
        break

print(msg)
```

L. The Circle of Death

Solution

Prerequisite:

A triangle in a circle is a right triangle if and only if its hypotenuse is the diameter of the circle.

How to Approach the Problem:

Given the input, we need to find all the two points that form a diameter of the circle: all the pairs of points where the length of the arc between them is half of the circle's perimeter

For each such pair of points, there exist $n - 2$ unique right triangles because the third point can be any of the remaining points.

So the problem becomes: How should we find these pairs of points?

First, we build a **prefix sum array** of the arc lengths => this allows us to find the length of the arc between any two points in $O(1)$

Suppose b this array:

$$b[p] = \sum_{j=1}^{p} a[j-1] \quad \text{where} \quad 1 \le p \le n, b[0] = 0$$

Second, how to find the pairs?

We can perform a linear search for every value in the prefix sum array but the time complexity would be $O(n^2)$. This will yield a TLE verdict given our constraints

Solution 1 : two pointers technique:

We will fix two indexes i and j and we will move them until we find the distance.

i will start from 0 and j from 1. If $(b[j] - b[i] = (1/2)\text{perimeter})$ then we will add $n - 2$ to the answer and we will increment i and j else if $(b[j] - b[i] < (1/2)\text{perimeter})$ then we will increment j because we need to increase the distance between the two points else we will increment i because we need to decrease the distance between the two points.

Time complexity: $O(n)$ **.**

This solution has the same logic as the Two Sum problem

Solution 2 : hash table:

We put all the values of the prefix sum array into a **hash table**. For each value b[i] , we can search for $(b[i] + (1/2)perimeter)$ in the hash table in $O(1)$ time complexity

Time complexity: $O(n)$.

Solution 3 : binary search:

For each b[i], we search for the value $(b[i] + (1/2)perimeter)$ using binary search $O(logn)$

Time complexity: $O(n * logn)$.

C++ Solution

Java Solution

Python Solution

```
def solve():
    n = int(input())
    a = list(map(int, input().split()))
    b = [0] * (n + 1)
    b[0] = 0
    total_sum = 0

    for i in range(n):
        total_sum += a[i]
        b[i + 1] = total_sum

    if total_sum % 2 != 0:
        print("0")
        return

    target_sum = total_sum // 2
    ans = 0
    i, j = 0, 1

    while i < n - 1 and j < n:
        if b[j] - b[i] == target_sum:
            ans += n - 2
            i += 1
            j += 1
        elif b[j] - b[i] < target_sum:
            j += 1
        else:
            i += 1

    print(ans)


if __name__ == "__main__":
    solve()
```

- Vote: I like it
- **0**
- Vote: I do not like it

Add to favourites

- Author LorenzoRaed02
- Publication date 12 months ago
- Comments 1

Comments Comments (1)

Write comment?

12 months ago, # | Add to favourites

LorenzoRaed02 *Auto comment: topic has been updated by LorenzoRaed02 (previous revision, new revision, compare).*

→ Reply