2020 UCC Coding Competition – Solutions and Answer Key

Answer Key – The test data is still available on the main contest website.

	Test Case 1 (1/10)	Test Case 2 (4/10)	Test Case 3 (5/10)
Problem 1	3	241	236
Problem 2	82	84	266
Problem 3	24	760	245
Problem 4	529	426	181
Problem 5	4	69	57

Problem 1 – Snowstorm

This problem can be solved by looping through the two strings simultaneously character by character, and keeping a counter of how many times both the strings have a '0' in the same spot.

Python 3 Implementation

```
n = int(input(''))                          # Take the necessary inputs
a = input('')
b = input('')
ctr = 0                                     # Start a counter
for i in range(n):                          # Loop through the strings
      if a[i] == '0' and b[i] == '0':       # If the character in both strings
            ctr += 1                        # are both '0', increment a counter

print(ctr)                                  # Print the counter value
```

Problem 2 – Optimal Skiing

To solve this problem, we must calculate the time required for each of the ski lift routes, and find the minimum.

To do this, we loop through all of the ski lift routes. For each of these ski lift routes, we can use another loop to find the total time required for that route.

While checking each route, store the duration of the shortest route found so far. By the end, this will be our answer.

Python 3 Implementation

```
n = int(input(''))

minRoute = 1000000                  # Initialize to a sufficiently high value

for i in range(n):
      l = [int (i) for i in input().split()]    # Input a line into a list
                                                # and cast everything to int
      currentRoute = 0

      for j in range(l[0]):         # l[0] is number of lifts in route
            currentRoute += l[j+1]  # l[1] to l[j] are lengths of each lift

      if currentRoute < minRoute:
            minRoute = currentRoute

if minRoute>=1000:            # This prints the last 3 digits when over 1000.
      print(minRoute%1000)
else:
      print(minRoute)
```

Problem 3 – Farmer Bob

This is a three-part problem:

1. Find the largest gap in the trees
2. Find the largest tractor that can fit through this gap
3. Calculate how many trips are needed

For the first part, we can loop through the given string and keep a counter as follows: If a character is a '0' or an 'X', we add 1 to the counter. If a character is a '1', we reset the counter to zero. Observe what happens with a few examples:

```
String:           000XX1
Counter value:    123450

String:           10X0111X0
Counter value:    012300012

String:           0X0100X101010XXXX1X1    (from Test Case 1)
Counter value:    12301230101012345010
```

Notice that the largest value achieved by the counter overall represents the length of the largest consecutive block of '0's and 'X's, which is the largest gap in the trees that we want.

For the second part, finding the largest tractor that can fit through the gap is made relatively straightforward as the list of tractors was sorted. We can just loop through the list of tractors and store the value of the largest tractor that works so far, until we hit a tractor that is too big, at which point we can break out of the loop.

For the third part, notice that the number of trips required was just the amount of hay divided by the largest usable tractor, rounding up. The rounding upwards can be done using the ceiling function, which returns the smallest integer greater than a value.

Python 3 Implementation

```
# STEP 0: Taking Inputs and Declaring Variables

import math
h = int(input(''))
t = int(input(''))
tractors = [int(i) for i in input().split()]
m = int(input(''))
s = input('')
maxConsecutive = 0
ctr = 0

# STEP 1: Finding largest gap

for i in range(m):
      if(s[i] == '0' or s[i] == 'X'):
            ctr += 1
            maxConsecutive = max(maxConsecutive, ctr)
      else:
            ctr = 0

# STEP 2: Finding largest tractor that fits

biggestTractor = 0
for i in range(t):
      if(tractors[i] <= maxConsecutive):
            biggestTractor = tractors[i]
      else:
            break

# STEP 3: Calculating trips required and printing answer

answer = math.ceil(h/biggestTractor)

if answer >= 1000:
      print(answer%1000)
else:
      print(answer)
```

Problems 4 and 5 are more sophisticated and were intended for contestants with more knowledge and experience with computer algorithms.

Problem 4 – Bubble Tea

This problem can be solved using Dynamic Programming. We can build an array, `dp[]`, where `dp[i]` stores the minimum cost of purchasing the first `i` bubble teas. Let the array `c[]` store the cost of each bubble tea, in order. By definition:

```
dp[0] = 0
dp[1] = c[1]
dp[2] = c[1] + c[2] – 0.25*min(c[1], c[2])
```

Now, the key observation is that no matter what, the minimum cost of purchasing the first *k* bubble teas is equivalent to the minimum of three options:

- The cost of buying the *k*th bubble tea plus the minimum cost of purchasing the first *k-1* bubble teas,
- The cost of buying the *k*th and *(k-1)*th bubble teas together, applying the 25% discount, plus the minimum cost of purchasing the first *k-2* bubble teas, and
- The cost of buying the *k*th, *(k-1)*th and *(k-2)*th bubble teas together, applying the 50% discount, plus the minimum cost of purchasing the first *k-3* bubble teas.

Applying this observation to calculate the values of `dp[3]` to `dp[n]` will give us the answer of the minimum cost to buy all of (the first n) bubble teas.

Pseudocode Implementation

```
Input n, and Input the next n integers into c[]

Declare array dp[] of size n+1

dp[0] = 0
dp[1] = c[1]
dp[2] = c[1] + c[2] – 0.25*min(c[1], c[2])

for i from 3 to n:
      dp[i] = min(
            c[i] + dp[i-1],
            c[i] + c[i-1] – 0.25*min(c[i], c[i-1]) + dp[i-2],
            c[i] + c[i-1] + c[i-2] – 0.5*min(c[i], c[i-1], c[i-2]) + dp[i-3]
      )

if dp[n] >= 1000:
      print(dp[n]%1000)
else:
      print(dp[n])
```

Problem 5 – Public Transport

This is a graph problem. Each subway line represents a node, and each escalator represents a directed edge connecting a pair of nodes. We are looking for the distance between two nodes on this graph.

One way to do this is with a well-known algorithm called Breadth-First Search (BFS). *If unfamiliar with BFS, it is recommended to do some reading on it before attempting the problem or reading the below implementation.*

<u>Pseudocode Implementation</u>

```
Input the integers L, start, end, N

Declare an array of dynamically-sized integer arrays, adj[][], with size L

For i from 1 to N:
      Input a pair of integers, a and b
      Add b to the dynamic array adj[a]

Declare an integer array, dist[], of size N+1, with all values 0
Declare an empty First-In-First-Out Queue of integers, q

Add start to q

While q is not empty:
      cur = front of q
      for nxt in adj[cur]:
            if dist[nxt] == 0:
                  add nxt to back of q
                  dist[nxt] = dist[cur] + 1

print(dist[end])
```