2021 Spring Coding Bowl – Official Solutions

Problem 1 – Counterfeit Detection

One solution for this problem is to simply take the number of '2' characters and subtract the number of '5' characters.

Python 3 Solution:

```
s = input()
print(s.count('2')-s.count('5'))
```

Problem 2 – Emerald Exchange

For this problem, loop through the list of values once while keeping a counter. If the currently-considered value is even, add that value to the counter. If the currently-considered value is odd, set the counter to 0. The answer will be the largest number that the counter achieves.

Python 3 Solution:

```
n = int(input())
emeralds = [int(i) for i in input().split()]
ans = 0
counter = 0
for i in range(n):
     if emeralds[i]%2 == 0:
         counter += emeralds[i]
        ans = max(ans, counter)
     else:
        counter = 0
print(ans)
```

Problem 3 – Long Pizza

For 4 partial marks, it suffices to simply store an array tracking the amount of cheese on each slice, and every time the machine is run, adding 1 to the amount of cheese in the machine's range. However, the worst-case time complexity of this approach is $O(NR)$, since for each of the R runs of the machine, you have to increment up to N values. As such, this solution is too slow and does not receive full marks.

Instead, to obtain full marks, each run of the machine should be processed in $O(1)$ time complexity. There are multiple ways in which this can be done. One way is using a *prefix difference array*, but there is also a potentially easier solution: for each run of the machine from l to r, calculate the size of the interval within l, r that overlaps with the desired range x, y, and simply add the interval sizes together to produce an answer.

Python 3 Solution

```
N = int(input())
x, y = map(int, input().split())
R = int(input())

answer = 0

for i in range(R):
       l, r = map(int, input().split())
       answer += max(0, min(r, y) - max(l, x) + 1)
       # the above expression evaluates to the size of the overlap between the
       # intervals [l, r] and [x, y].

print(answer)
```

Problem 4 – Trampoline Jump

This problem requires two steps – first, generating the sequence $a_1, a_2, \ldots$ and second, finding the shortest possible path to the destination house.

The Fibonacci sequence can be generated from the bottom-up (remembering to take mod 2021 after calculating every term to prevent integer overflow), and adding the $i \ \% \ 50$ term will produce the desired values.

In order to find the shortest path to the destination house, observe that the road can be considered as a directed graph, with each house as a node. From each node, there are up to four directed edges (representing the forward or reverse walk or jump). It suffices to run the Breadth-First Search (BFS) algorithm to find the shortest path to the destination.

C++ Solution

```
#include <bits/stdc++.h>
using namespace std;

int main() {

    int n; cin >> n;
    int a [n+1] = {0}, fib [n+1] = {0}, mod = 2021;

    // Generate Sequence
    for(int i = 1; i <= n; i++){
        if(i==1 or i==2) fib[i] = 1;
        else fib[i] = (fib[i-1]+fib[i-2])%2021;
        a[i] = fib[i]+i%50;
    }

    // Run BFS algorithm
    vector<int> visited (n+1);
    queue<int> q;
    q.push(1);
    while(!q.empty()){
        int v = q.front(); q.pop();
        if(v+1 <= n and !visited[v+1]){visited[v+1] = visited[v]+1; q.push(v+1);}
```

```
        if(v-1 >= 1 and !visited[v-1]){visited[v-1] = visited[v]+1; q.push(v-1);}
        if(v+a[v] <= n and !visited[v+a[v]]){visited[v+a[v]] = visited[v]+1; q.push(v+a[v]);}
        if(v-a[v] >= 1 and !visited[v-a[v]]){visited[v-a[v]] = visited[v]+1; q.push(v-a[v]);}
        if(v==n){
            cout << visited[v] << endl;
            return(0);
        }
    }
}
```

Problem 5 – Woodcutting Game

A recursive dynamic programming approach can be implemented for this problem. Note that a "winning position" means that the player who receives that position has a strategy to guarantee a win, and a "losing position" means that the player who receives that position can always be forced to lose. We simply need to output "W" if the starting position is a winning position, and "L" if the starting position is a losing position.

The array `dp[][][][]` will store the DP state. In particular, `dp[h1][w1][h2][w2]` will store the state of one h1 by `w1` and one h2 by `w2` rectangular board in the woodcutting game using the following values:

- `0` if the state has not yet been calculated
- `1` if the state has been calculated to be a winning position
- `-1` if the state has been calculated to be a losing position

The key observation for this problem and for similar game theory problems is that if at least one of the possible moves from a state leads to a losing position, that state is a winning position. Otherwise, if all of the possible moves from a state lead to a winning position, that state is a losing position.

Using the above observation, we can recursively fill out the DP table. The recursive base case is `dp[1][1][1][1] = -1`, since by definition, two 1 by 1 boards is a losing position. Since we have an array to memorize the recursive function, the time complexity of the program is sufficiently optimized.

C++ Solution

```
#include <bits/stdc++.h>
using namespace std;

int dp [3][301][3][301] = {0};  // Initialize DP array

int f (int h1, int w1, int h2, int w2) {
    // If state has already been calculated, return the previously calculated value
    if(dp[h1][w1][h2][w2]!=0) return(dp[h1][w1][h2][w2]);

    // Recursive base case
    if(h1==1 and w1==1 and h2==1 and w2==1) return dp[1][1][1][1]=-1;

    // If any of the possible moves from OPTION 1 lead to a losing position, this state is a
    // winning position
    for(int i = 1; h1-i > 0; i++)
        if(f(i, w1, h1-i, w1)==-1) return dp[h1][w1][h2][w2] = 1;
    for(int i = 1; w1-i > 0; i++)
        if(f(h1, i, h1, w1-i)==-1) return dp[h1][w1][h2][w2] = 1;
```

```cpp
    for(int i = 1; h2-i > 0; i++)
        if(f(i, w2, h2-i, w2)==-1) return dp[h1][w1][h2][w2] = 1;
    for(int i = 1; w2-i > 0; i++)
        if(f(h2, i, h2, w2-i)==-1) return dp[h1][w1][h2][w2] = 1;

    // If any of the possible moves from OPTION 2 lead to a losing position, this state is a
    // winning position
    if(h1==2)
        if(f(1, w1, h2, w2)==-1) return dp[h1][w1][h2][w2] = 1;
    if(h2==2)
        if(f(h1, w1, 1, w2)==-1) return dp[h1][w1][h2][w2] = 1;
    for(int k = 1; k <= 10 and w1-k >= 1; k++)
        if(f(h1, w1-k, h2, w2)==-1) return dp[h1][w1][h2][w2] = 1;
    for(int k = 1; k <= 10 and w2-k >= 1; k++)
        if(f(h1, w1, h2, w2-k)==-1) return dp[h1][w1][h2][w2] = 1;

    // Otherwise, this state is a losing position.
    return dp[h1][w1][h2][w2] = -1;
}

int main() {
    int h1,w1,h2,w2; cin >> h1 >> w1 >> h2 >> w2;

    // Call the recursive function to find the answer.
    if(f(h1,w1,h2,w2)==1)cout << 'W' << endl;
    else cout << 'L'<< endl;
}
```