Guess Number Higher or Lower II

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to **n**. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.

However, when you guess a particular number x, and you guess wrong, you pay **$x**. You win the game when you guess the number I picked.

Example:

```
n = 10, I pick 8.

First round:  You guess 5, I tell you that it's higher. You pay $5.
Second round: You guess 7, I tell you that it's higher. You pay $7.
Third round:  You guess 9, I tell you that it's lower. You pay $9.

Game over. 8 is the number I picked.

You end up paying $5 + $7 + $9 = $21.
```

Given a particular **n ≥ 1**, find out how much money you need to have to guarantee a **win**.

1. The best strategy to play the game is to minimize the maximum loss you could possibly face. Another strategy is to minimize the expected loss. Here, we are interested in the **first** scenario.
2. Take a small example (n = 3). What do you end up paying in the worst case?
3. Check out this article if you're still stuck.
4. The purely recursive implementation of minimax would be worthless for even a small n. You MUST use dynamic programming.
5. As a follow-up, how would you modify your code to solve the problem of minimizing the expected loss, instead of the worst-case loss?

Credits:
Special thanks to @agave and @StefanPochmann for adding this problem and creating all test cases.

Solution 1

For each number x in range[i~j]
we do: result_when_pick_x = x + **max**{DP([i~x-1]), DP([x+1, j])}
--> *// the max means whenever you choose a number, the feedback is always bad and therefore leads you to a worse branch.*
then we get DP([i~j]) = **min**{xi, ... ,xj}
--> *// this min makes sure that you are minimizing your cost.*

```java
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] table = new int[n+1][n+1];
        return DP(table, 1, n);
    }

    int DP(int[][] t, int s, int e){
        if(s >= e) return 0;
        if(t[s][e] != 0) return t[s][e];
        int res = Integer.MAX_VALUE;
        for(int x=s; x<=e; x++){
            int tmp = x + Math.max(DP(t, s, x-1), DP(t, x+1, e));
            res = Math.min(res, tmp);
        }
        t[s][e] = res;
        return res;
    }
}
```

Here is a bottom up solution.

```java
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] table = new int[n+1][n+1];
        for(int j=2; j<=n; j++){
            for(int i=j-1; i>0; i--){
                int globalMin = Integer.MAX_VALUE;
                for(int k=i+1; k<j; k++){
                    int localMax = k + Math.max(table[i][k-1], table[k+1][j]);
                    globalMin = Math.min(globalMin, localMax);
                }
                table[i][j] = i+1==j?i:globalMin;
            }
        }
        return table[1][n];
    }
}
```

written by bbccyy1 original link here

Solution 2

Definition of `dp[i][j]`: minimum number of money to guarantee win for subproblem `[i, j]`.

Target: `dp[1][n]`

Corner case: `dp[i][i] = 0` (because the only element must be correct)

Equation: we can choose `k (i<=k<=j)` as our guess, and pay price `k`. After our guess, the problem is divided into two subproblems. Notice we do not need to pay the money for both subproblems. We only need to pay the worst case (because the system will tell us which side we should go) to guarantee win. So `dp[i][j] = min (i<=k<=j) { k + max(dp[i][k-1], dp[k+1][j]) }`

```
public class Solution {
    public int getMoneyAmount(int n) {
        if (n == 1) {
            return 0;
        }
        int[][] dp = new int[n + 1][n + 1];
        for (int jminusi = 1; jminusi < n; jminusi++) {
            for (int i = 0; i + jminusi <= n; i++) {
                int j = i + jminusi;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k <= j; k++) {
                    dp[i][j] = Math.min(dp[i][j],
                                        k + Math.max(k - 1 >= i ? dp[i][k - 1] :
0,
                                                     j >= k + 1 ? dp[k + 1][j] :
0));
                }
            }
        }
        return dp[1][n];
    }
}
```

written by dachuan.huang original link here

Solution 3

Big Idea: Given any n, we make a guess k. Then we break the interval [1,n] into [1,k - 1] and [k + 1,n]. The min of worst case cost can be calculated recursively as

cost[1,n] = k + max{cost[1,k - 1] + cost[k+1,n]}

Also, it takes a while for me to wrap my head around "min of max cost". My understand is that: you strategy is the best, but your luck is the worst. You only guess right when there is no possibilities to guess wrong.

```java
public class Solution {
    public int getMoneyAmount(int n) {
        // all intervals are inclusive
        // uninitialized cells are assured to be zero
        // the zero column and row will be uninitialized
        // the illegal cells will also be uninitialized
        // add 1 to the length just to make the index the same as numbers used
        int[][] dp = new int[n + 1][n + 1]; // dp[i][j] means the min cost in the worst case for numbers (i...j)

        // iterate the lengths of the intervals since the calculations of longer intervals rely on shorter ones
        for (int l = 2; l <= n; l++) {
            // iterate all the intervals with length l, the start of which is i. Hence the interval will be [i, i + (l - 1)]
            for (int i = 1; i <= n - (l - 1); i++) {
                dp[i][i + (l - 1)] = Integer.MAX_VALUE;
                // iterate all the first guesses g
                for (int g = i; g <= i + (l - 1); g++) {
                    int costForThisGuess;
                    // since if g is the last integer, g + 1 does not exist, we have to separate this case
                    // cost for [i, i + (l - 1)]: g (first guess) + max{the cost of left part [i, g - 1], the cost of right part [g + 1, i + (l - 1)]}
                    if (g == n) {
                        costForThisGuess = dp[i][g - 1] + g;
                    } else {
                        costForThisGuess = g + Math.max(dp[i][g - 1], dp[g + 1][i + (l - 1)]);
                    }
                    dp[i][i + (l - 1)] = Math.min(dp[i][i + (l - 1)], costForThisGuess); // keep track of the min cost among all first guesses
                }
            }
        }
        return dp[1][n];
    }
}
```

Any questions, suggestions & criticism welcomed!

written by myanonymos original link here