375. Guess Higher or Lower Number II

Dec. 6, 2016 | 21.6K views

Previous Next

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**.

Summary

Given a number n, we have to find the worst case cost of guessing a number chosen from the range $(1, n)$, assuming that the guesses are made intelligently(minimize the total cost). The cost is incremented by i for every wrong guess i.

For example:

```
n=5
1 2 3 4 5
```

If we start with 3 as the initial guess, the next guess would certainly be 4 as in the worst case required number is 5. Total Cost $= 4 + 3 = 7$.

But if we start with 4 as the initial guess, our next guess would be 2 as in the worst case required number is 3 or 1. Total Cost $= 4 + 2 = 6$ which is the minimum cost.

```
n=8
1 2 3 4 5 6 7 8
```

In this case we have to guess 5 followed by 7. Total Cost $= 5 + 7 = 12$. If we choose 4 as our initial guess. Total Cost $= 4 + 5 + 7 = 16$.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

Firstly, we need to be aware of the fact that out of the range $(1, n)$, we have to guess the numbers intelligently in order to minimize the cost. But, along with that we have to take into account the worst case scenario possible, that is we have to assume that the original number chosen is such that it will try to maximize the overall cost.

In Brute Force, we can pick up any number i in the range $(1, n)$. Assuming it is a wrong guess(worst case scenario), we have to minimize the cost of reaching the required number. Now, the required number could be lying either to the right or left of the number picked(i). But to cover the possibility of the worst case number chosen, we need to take the maximum cost out of the cost of reaching the worst number out of the right and left segments of i. Thus, if we pick up i as the pivot, the overall minimum cost for the worst required number will be:

$$\text{cost}(1, n) = i + \max\left(\text{cost}(i, i-1), \text{cost}(i+1, n)\right)$$

For every segment, we can further choose another pivot and repeat the same process for calculating the minimum cost.

By using the above procedure, we found out the cost of reaching the required number starting with i as the pivot. In the same way, we iterate over all the numbers in the range $(1, n)$, choosing them as the pivot, calculating the cost of every pivot chosen and thus, we can find the minimum cost out of those.

Java

```
public class Solution {
    public int getMoneyAmount(int low, int high) {
        if (low >= high)
            return 0;
        int minres = Integer.MAX_VALUE;
        for (int i = low; i <= high; i++) {
            int res = i + Math.max(calcCost(1 + 1, high), calcCost(low, i - 1));
            minres = Math.min(res, minres);
        }

        return minres;
    }

    public int getMoneyAmount(int n) {
        return calcCost(1, n);
    }
}
```

Complexity Analysis

- Time complexity : $O(n!)$. We choose a number as pivot and repeat the pivoting process further n times $O(n!)$. We repeat the same process for n pivots.
- Space complexity : $O(n)$. Recursion of depth is used.

Approach #2 Modified Brute Force [Time Limit Exceeded]

Algorithm

In Brute Force, for numbers in the range (i, j), we picked up every number from i to j as the pivot and found the maximum cost out of its left and right segments. But an important point to observe is that if we choose any number from the range $(i, \frac{i+j}{2})$ as the pivot, the right segment(consisting of numbers larger than the picked up pivot) will be longer than the left segment(consisting of numbers smaller than i). Thus, we will always get the maximum cost from its right segment and it will be larger than the minimum cost achievable by choosing some other pivot. Therefore, our objective here is to reduce the larger cost which is coming from the right segment. Thus, it is wise to choose the pivot from the range $(\frac{i+j}{2}, j)$. In this way the costs of the two segments will be nearer to each other and this will minimize the overall cost.

Thus, while choosing the pivot instead of iterating from i to j, we iterate from $\frac{i+j}{2}$ to j and find the minimum achievable cost similar to brute force.

Java

```
public class Solution {
    public int calcCost(int low, int high) {
        if (low >= high)
            return 0;
        int minres = Integer.MAX_VALUE;
        for (int i = (low + high) / 2; i <= high; i++) {
            int res = i + Math.max(calcCost(i + 1, high), calcCost(low, i - 1));
            minres = Math.min(res, minres);
        }

        return minres;
    }

    public int getMoneyAmount(int n) {
        return calcCost(1, n);
    }
}
```

Complexity Analysis

- Time complexity : $O(n!)$. We choose a number as pivot and repeat the pivoting process further n times $O(n!)$. We repeat the same process for n pivots.
- Space complexity : $O(n)$. Recursion of depth is used.

Approach #3 Using DP [Accepted]

Algorithm

The problem of finding the minimum cost of reaching the destination number choosing i as a pivot can be divided into the subproblem of finding the maximum out of the minimum costs of its left and right segments as explained above. For each segment, we can continue the process leading to smaller and smaller subproblems. This leads us to the conclusion that we can use DP for this problem.

We need to use a dp matrix, where $dp[i, j]$ refers to the minimum cost of finding the worst number given only the numbers in the range (i, j). Now, we need to know how to fill in the entries of this dp. If we are given only a single number k, no matter what the number is the cost of finding that number is always 0 since we always hit the number directly without any wrong guess. Thus, firstly, we fill in all the entries of the dp which correspond to segments of length 1 i.e. all entries $dp[k, k]$ are initialized to 0. Then, in order to find the entries for segments of length 2, we need all the entries for segments of length 1. Thus, in general, to fill in the entries corresponding to segments of length len, we need all the entries of length $len - 1$ and below to be already filled. Thus, we need to fill the entries in the order of their segment lengths. Thus, we fill the entries of dp diagonally.

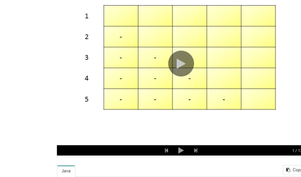
Now, what criteria do we need to fill up the dp matrix? For any entry $dp[i, j]$, given the current segment length of interest is len i.e. $len = j - i + 1$, we assume as if we are available only with the numbers in the range (i, j). To fill in its current entry, we follow the same process as Approach 1, choosing every number as the pivot and finding the minimum cost as:

$$\text{cost}(i, j) = \text{pivot} + \max\left(\text{cost}(i, \text{pivot} - 1), \text{cost}(\text{pivot} + 1, n)\right)$$

But, we have an advantage in terms of calculating the cost here, since we already know the costs for the segments of length smaller than len from dp. Thus, the dp equation becomes:

$$dp[i, j] = \min_{\text{pivot} \in (i, j)} \left[\text{pivot} + \max\left(dp[i, \text{pivot} - 1], dp[\text{pivot} + 1, n]\right)\right]$$

where $\min_{\text{pivot} \in (i, j)}$ indicates the minimum obtained by considering every number in the range (i, j) as the pivot.



Java

```
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] dp = new int[n + 1][n + 1];
        for (int len = 2; len <= n; len++) {
            for (int start = 1; start <= n - len + 1; start++) {
                int minres = Integer.MAX_VALUE;
                for (int pivot = start; pivot < start + len; pivot++) {
                    int res = pivot + Math.max(dp[start][pivot - 1], dp[pivot + 1][start + len - 1]);
                    minres = Math.min(res, minres);
                }
                dp[start][start + len - 1] = minres;
            }
        }
        return dp[1][n];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$. We traverse the complete dp matrix once $(O(n^2))$. For every entry we take atmost n numbers as pivot.
- Space complexity : $O(n^2)$. dp matrix of size n^2 is used.

Approach #4 Better Approach using DP [Accepted]

Algorithm

In the last approach, we chose every possible pivot from the range (i, j). But, as per the argument given in Approach 2, we can choose pivots only from the range $(i + (len - 1)/2, j)$, where len is the current segment length of interest. Thus the governing equation is:

$$dp[i, j] = \min_{\text{pivot} \in \left(i + \frac{len-1}{2}, j\right)} \left[\text{pivot} + \max\left(dp[i, \text{pivot} - 1], dp[\text{pivot} + 1, n]\right)\right]$$

Thus, we can optimize the Approach 3 to some extent.

Java

```
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] dp = new int[n + 1][n + 1];

        for (int len = 2; len <= n; len++) {
            for (int start = 1; start <= n - len + 1; start++) {
                int minres = Integer.MAX_VALUE;
                for (int pivot = start + (len - 1) / 2; pivot < start + len - 1; pivot++) {
                    int res = pivot + Math.max(dp[start][pivot - 1], dp[pivot + 1][start + len - 1]);
                    minres = Math.min(res, minres);
                }
                dp[start][start + len - 1] = minres;
            }

        }
        return dp[1][n];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$. We traverse the complete dp matrix once $(O(n^2))$. For every entry we take atmost n numbers as pivot.
- Space complexity : $O(n^2)$. dp matrix of size n^2 is used.

Rate this article: ★ ★ ★ ★ ★

Previous Next

Comments: 36 Sort By

Type comment here... (Markdown is supported)

Preview Post

cdb ★ 117 ⏱ August 01, 2019 3:29 PM
I'd refuse to play this game.
53 ⬆ ⬇ ↱ Share ↳ Reply
SHOW 3 REPLIES

Carlos_Galow ★ 410 ⏱ November 3, 2019 11:22 AM
This article is really hard to understand, don't know why leetcode accepts it as solution.
46 ⬆ ⬇ ↱ Share ↳ Reply
SHOW 1 REPLY

ps220 ★ 10 ⏱ June 7, 2019 9:46 PM
Why the worst case in binary search is Not the solution? Say we search n in range(1,n+1) and add up all the middle number until it reaches n.
11 ⬆ ⬇ ↱ Share ↳ Reply
SHOW 2 REPLIES

UR2211 ★ 15 ⏱ June 1, 2020 3:03 AM
This problem is no where close to medium
6 ⬆ ⬇ ↱ Share ↳ Reply

goodstudy123 ★ 30 ⏱ November 21, 2019 4:56 AM
In Approach #4, why we can choose pivots only from the range (i+(len-1)/2,j) ?
8 ⬆ ⬇ ↱ Share ↳ Reply

hello_world_en ★ 284 ⏱ June 30, 2019 1:26 PM
`pivot < start + len - 1` should be
`pivot < start + len - 2`
`int[][] dp = new int[n + 1][n + 1];` should be
`int[][] dp = new int[n + 2][n + 2];`
4 ⬆ ⬇ ↱ Share ↳ Reply

Innovat ★ 121 ⏱ January 4, 2017 3:46 AM
`cost(i,j)=pivot+max(cost(i,pivot-1),cost(pivot+1,n))` should be
`cost(i,j)=pivot+max(cost(i,pivot-1),cost(pivot+1,j))` ?
3 ⬆ ⬇ ↱ Share ↳ Reply

tcbrown ★ 31 ⏱ June 24, 2020 9:45 AM
I win the game by not guessing anything, so that I won't lose any money.
3 ⬆ ⬇ ↱ Share ↳ Reply

nzace ★ 11 ⏱ May 13, 2019 5:56 AM
For solution 3 I think:

```
dp[i,j]=min pivot∈(i,j) [pivot+max(dp[i,pivot-1],dp[pivot+1,n])]
```

Read More
2 ⬆ ⬇ ↱ Share ↳ Reply
SHOW 1 REPLY

Navarone ★ 1140 ⏱ April 20, 2019 4:01 AM
Hi, I just have an idea but don't know why it is wrong, hope someone can help me!
I think we only need to store the cost of subarray of the same length once.
For example, `[1,2,3,4]` (length=4), we will first pick `3` and then `1`, so the cost is `4`. And for any
Read More
0 ⬆ ⬇ ↱ Share ↳ Reply
SHOW 2 REPLIES