题目：

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**.

思路：

这题按类型分的话可以划为博弈类 DP，类似的还有 Lintcode 上的 Coins in a Line 1,2,3

我们先定义 dp[j]，代表着如果我们在区间 [i , j] 内进行查找，所需要的最少 cost 来保证找到结果。(当然，因为给定数字是 [1, n]，这里有一个 index off by one 的问题)。不难发现对于最开始的函数输入 n ，我们的最终结果就是 dp[0][n - 1] ，也即数字区间 [1 , n] 保证得到结

果所需要的最小 cost.

如果以 top-down recursion 的方式分析这个问题，可以发现对于区间 [i, j] ，我们的猜测 i <= k <= j 我们可能出现以下三种结果：

1. k 就是答案，此时子问题的额外 cost = 0 ，当前位置总 cost = k + 0;

2. k 过大，此时我们的有效区间缩小为 [i , k - 1] 当前操作总 cost = k + dp[start][k - 1];

3. k 过小，此时我们的有效区间缩小为 [k + 1 , j] 当前操作总 cost = k + dp[k + 1][j];

由于我们需要 “保证得到结果” ，也就是说对于指定 k 的选择，我们需要准备最坏情况 cost 是以下三种结果生成的 subproblem 中 cost 最大的那个；然而同时对于一个指定区间 [i , j] ，我们可以选择任意 i <= k <= j ，对于这个 k 的主观选择可以由我们自行决定，我们要选的是 k s.t. 其子问题的 cost + 当前操作 cost 最小的一个，至此，每次决策就构成了一次 MiniMax 的博弈。

同时因为我们有很多的 overlapping subproblems ，而且问题本身具有 optimal substructure，提高算法效率最简单直观的方式，就是用 int[][] dp 做缓存，来进行自顶向下的记忆化搜索（top-down memoized search).

1.时间：O（N^2）;空间：O（N^2）

```
class Solution {
public:
    int getMoneyAmount(int n) {
        if (n < 1) return 0;
```

```
        std::vector<std::vector<int>> dp(n + 1, std::vector<int>(n + 1, 0));
        return dfs(dp, 1, n);
    }
private:
    int dfs(std::vector<std::vector<int>>& dp, int start, int end){
        if (start >= end) return 0;
        if (dp[start][end] != 0) return dp[start][end];
        int min_cost = std::numeric_limits<int>::max();
        for (int i = start; i <= end; ++i){
            int tmp = i + std::max(dfs(dp, start, i - 1), dfs(dp, i + 1, end));
            min_cost = std::min(min_cost, tmp);
        }
        dp[start][end] = min_cost;
        return dp[start][end];
    }
};
```