I choose the last question, Leetcode 877. Stone Game (Medimum)

877. Stone Game

Medium 2090 2094 Add to List Share

Alice and Bob play a game with piles of stones. There are an **even** number of piles arranged in a row, and each pile has a **positive** integer number of stones `piles[i]`.

The objective of the game is to end with the most stones. The **total** number of stones across all the piles is **odd**, so there are no ties.

Alice and Bob take turns, with **Alice starting first**. Each turn, a player takes the entire pile of stones either from the **beginning** or from the **end** of the row. This continues until there are no more piles left, at which point the person with the **most stones wins**.

Assuming Alice and Bob play optimally, return `true` *if Alice wins the game, or* `false` *if Bob wins.*

Example 1:

```
Input: piles = [5,3,4,5]
Output: true
Explanation:
Alice starts first, and can only take the first 5 or the last 5.
Say she takes the first 5, so that the row becomes [3, 4, 5].
If Bob takes 3, then the board is [4, 5], and Alice takes 5 to win with 10 points.
If Bob takes the last 5, then the board is [3, 4], and Alice takes 4 to win with 9
points.
This demonstrated that taking the first 5 was a winning move for Alice, so we return
true.
```

a)Explain the various ways you tried to solve the problem, telling us what worked and what did not work (3 points)

Except the dynamic programming ways, I also tried backtrack way and recursion with memorization way.

The backtrack way:

```
class Solution {
public:
    bool stoneGame(vector<int>& piles) {
        int score = helper(piles, 0, piles.size() - 1, true);
        int total_score = 0;
```

```
        for (int i = 0; i < piles.size(); i++ ) {
            total_score += piles[i];
        }
        return score > total_score / 2;
    }

    int helper(vector<int> &piles, int start, int end, bool alice) {
        if (start == end) {
            if (alice) return piles[start];
            else return 0;
        }
        if (alice) {
            return max(helper(piles, start + 1, end, !alice) + piles[start],
helper(piles, start, end - 1, !alice) + piles[end]);
        }
        return max(helper(piles, start + 1, end, !alice), helper(piles, start,
end - 1, !alice));
    }
};
```

It is the most intuitive brutal force way, basically it is just enumrate all possible cases of the game by build a huge selection tree. The time complexity for this algorithm is expoential, because for every recursion step, we will enumerate both two choices, pick head pile or pick tail pile, which build two subtree in each step. The time complexity is O(2^n). So it does not work on the given test set. I will TLE: https://leetcode.com/submissions/detail/742908024/

Then I modified this way, because sometimes we do not actually need to generate every subtree, some subtree are replicated and can be recorded by memorization

This is the memorization version:

```
class Solution {
private:
    int memo[510][510][2];

public:
    bool stoneGame(vector<int>& piles) {
        int n = piles.size();
        memset(memo, -1, sizeof(memo));
        int score = helper(piles, 0, piles.size() - 1, true);
```

```
        int total_score = 0;
        for (int i = 0; i < n; i++ ) {
            total_score += piles[i];
        }
        return score > total_score / 2;
    }

    int helper(vector<int> &piles, int start, int end, bool alice) {
        if (memo[start][end][(int) alice] >= 0) return memo[start][end][(int)
alice];
        if (start == end) {
            if (alice) return piles[start];
            else return 0;
        }
        int ans = -1;
        if (alice) {
            ans = max(helper(piles, start + 1, end, !alice) + piles[start],
helper(piles, start, end - 1, !alice) + piles[end]);
        } else {
            ans = max(helper(piles, start + 1, end, !alice), helper(piles, start,
end - 1, !alice));
        }
        memo[start][end][(int) alice] = ans;
        return memo[start][end][(int) alice];
    }
};
```

The basic idea is simple, record the result for every recursion result, so when next time it need to recurse on a same segment of piles and in same turn, we could get the result from the `memo`, and do not need to actually generate the sub selection tree.

The time complexity is O(n^2) for this solution, and it can pass all test case: https://leetcode.com/submissions/detail/742914890/

b) Describe what insights you had as you eventually found a correct solution (3 points)

After I came up with the recursion with memorization solution. I knew this question can be solved by dynamic programming to further improve on space complexity.

The dynamic programming way and recursion with memorization are kind of symmetric solution. recursion is moving from what we do not know to what we already know (Top to

bottom) and the dynamic programming way is from what we already know to what we do not know (Bottom to top). So I could establish the equation to the subproblem by using the parameters for recursion way. They are start and end index of the segment and the turn infomation.

What have to do is just rewrite the `ans = max(helper(piles, start + 1, end, !alice) + piles[start], helper(piles, start, end - 1, !alice) + piles[end])` to `dp[start][end][alice] = max(dp[start + 1][end][!alice] + piles[start], dp[start][end][!alice][end]`.

Therefore the insight I learned from this question is: if I cannot figure out the dynamic programming solution, I may start with brutal force, then try to optimize it with memorizatrion, if it works, it is easy to transform it to dynamic programming solution

c) Reflect on what you learned from struggling on this problem, and describe how the struggle itself was valuable for you (3 points)

The most struggle part is find the relation between the problem and its subproblem. I do not recoginize it immediately after read this problem. I start brutal force and optimize it step by step and found the relation.

The most valuable struggle is it makes me learned a thinking pattern: first start with brutal force way, and try to examine what is repetitive or useless work in your brutal force way, then try to optimize it step by step. Especially for dynamic programming questions. This model is more efficient than directly come up with the subproblem relation equation from scretch when facing some hard dynamic programming problem.