1140. Stone Game II (DP, Suffix-Prefix Sum)

26 May 2023 06:26 AM

1140. Stone Game II

Hint

Medium 1.7K 299

Companies

Alice and Bob continue their games with piles of stones. There are a 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.

↳ Alice wants to win

Alice and Bob take turns, with Alice starting first. Initially, `M = 1`.

choose

On each player's turn, that player can take **all the stones** in the **first** `X` remaining piles, where `1 <= X <= 2M`. Then, we set `M = max(M, X)`.

The game continues until all the stones have been taken.

Assuming Alice and Bob play optimally, return the maximum number of stones Alice can get.

Example 1:

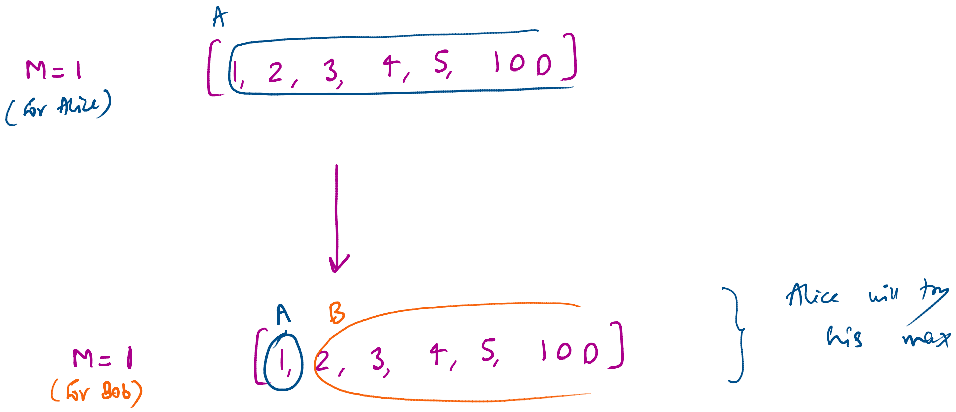
Bob

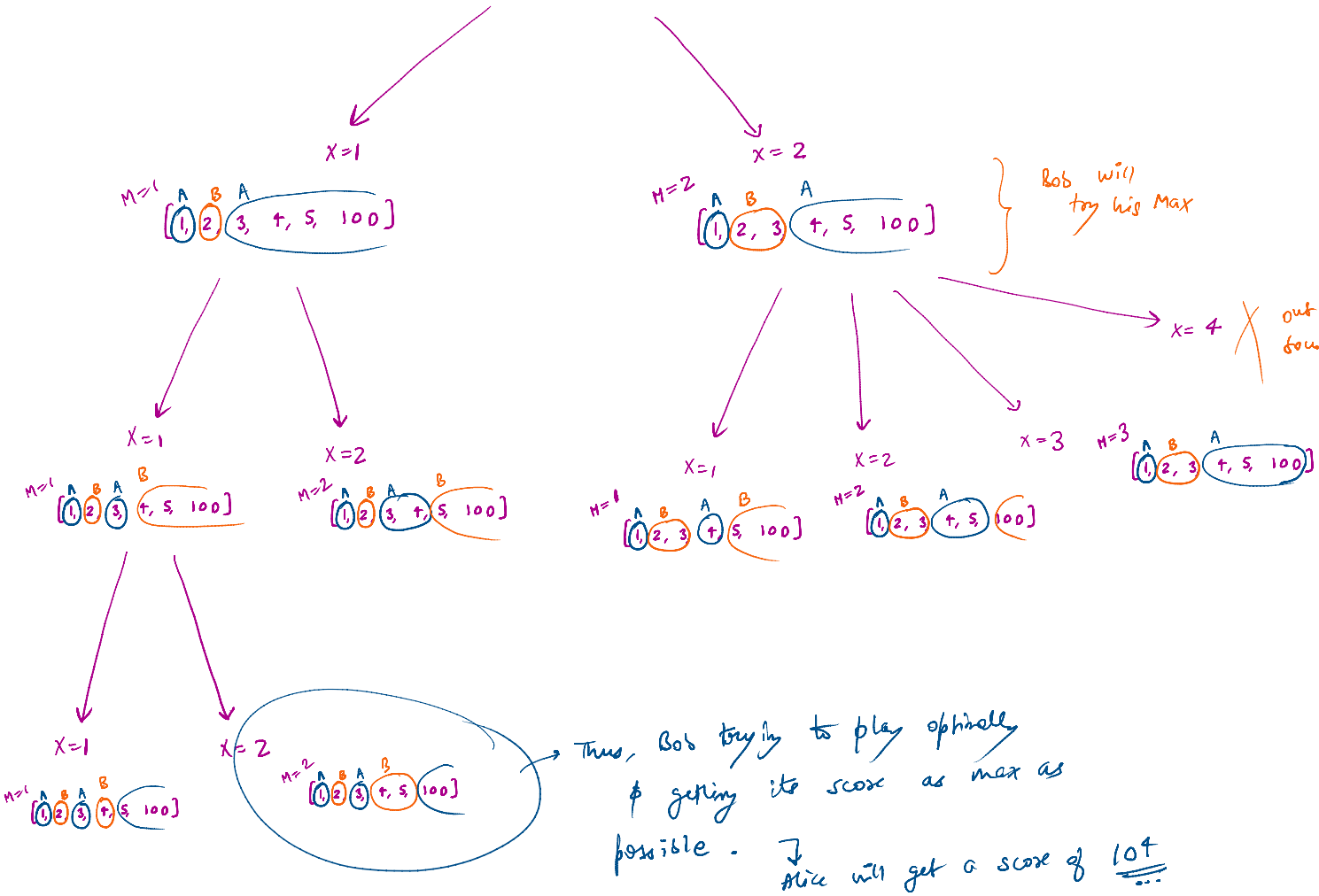
Alice

```
Input: piles = [2,7,9,4,4]
Output: 10
Explanation:  If Alice takes one pile at the beginning, Bob takes two piles, then Alice takes 2 piles again. Alice can get 2 + 4 + 4 = 10 piles in total. If Alice takes two piles at the beginning, then Bob can take all three piles left. In this case, Alice get 2 + 7 = 9 piles in total. So we return 10 since it's larger.
```

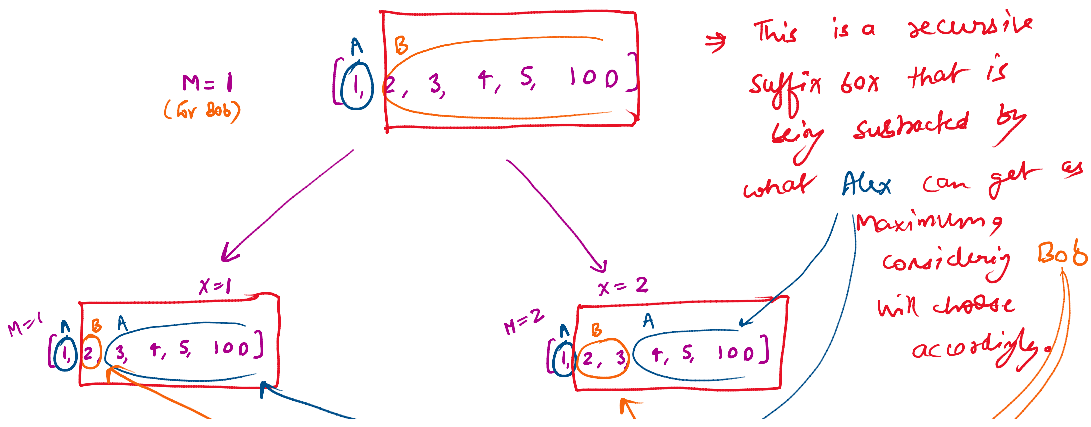
Example 2:

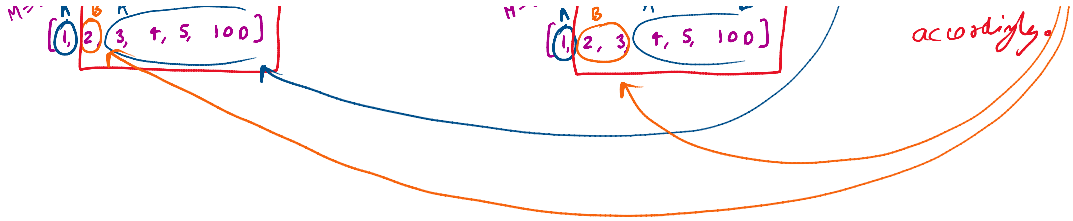
```
Input: piles = [1,2,3,4,5,100]
Output: 104
```





⇒ It is a simple recursion where every player can choose any number of stones from $1 \le X \le 2M$ (Initially $M=1$) & M becomes $M = \max(M, X)$





↳ Recursive call will have 2 parameters
- current (i) → (i) at which index I have to maximize suffix
- current (M) → what would be the next M according to current X i.e. $1 \le X \le 2M$

⇒ As we can also have repeating sub-problem for the pair of combinations (i, M) where i → suffix i

we can use Memoization. ☺

Let's write the code & understand with example :- ⇒

```cpp
class Solution {
public:
    int helper(vector<int>& piles, vector<vector<int>>& dp, const vector<int>& suffixSum, int i, int M) {
        if (i == piles.size()) return 0;
        if (i + 2 * M >= piles.size()) return suffixSum[i];

        if (dp[i][M] != 0) return dp[i][M];

        int result = 0;
        for (int x = 1; x <= 2 * M; ++x) {
            result = max(result, suffixSum[i] - helper(piles, dp, suffixSum, i + x, max(M, x)));
        }
        dp[i][M] = result;
        return result;
    }

    int stoneGameII(vector<int>& piles) {
        if (piles.empty()) return 0;
        vector<vector<int>> dp(piles.size(), std::vector<int>(piles.size(), 0));

        vector<int> suffixSum(piles.size());
        suffixSum[suffixSum.size() - 1] = piles[piles.size() - 1];
        for (int i = piles.size() - 2; i >= 0; --i) suffixSum[i] = piles[i] + suffixSum[i + 1];

        return helper(piles, dp, suffixSum, 0, 1);
    }
};
```

when i reaches end

→ Base case

when I can take all end elements so take ALL.

→ Memoization.

Trying every possibility of $1 \le X \le 2M$

Max condition as per question.

Taking max of what he/she can take

taking from i suffix

thinking I took (x) Piles next opponent will start from (i+x)

Memoization

Normal suffix sum

→ Edge case

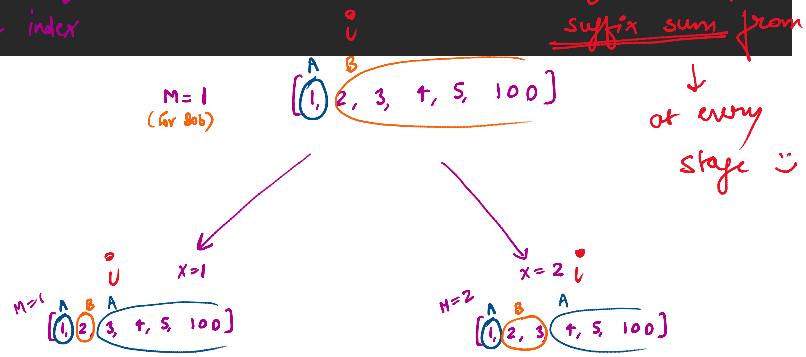
→ DP memo initialization

→ Suffix sum array

start index

start M

As I would be needing to compute suffix sum from i at every stage :)



Constraints:

- $1 \le piles.length \le 100$
- $1 \le piles[i] \le 10^4$

n

time :- $O(n * M * (2M)) = O(n^3)$

Will not go beyond n

will not go beyond n

space: $O(n * M) = O(n^2)$

Tabulation part is also exactly same :- ⇒

```cpp
class Solution {
public:
    int stoneGameII(vector<int>& piles) {
        int n = piles.size();
        vector<int> suffixSum(n + 1);
        suffixSum[n - 1] = piles[n - 1];
        for (int i = n - 2; i >= 0; i--) suffixSum[i] = suffixSum[i + 1] + piles[i];

        vector<vector<int>> dp(n, vector<int>(n + 1));

        for (int i = n - 1; i >= 0; i--) {
            for (int m = 1; m <= n; m++) {
                if (i + 2 * m >= n) {
                    dp[i][m] = suffixSum[i];
                } else {
                    for (int x = 1; x <= 2 * m; x++) {
                        int opponentScore = dp[i + x][max(x, m)];
                        int score = suffixSum[i] - opponentScore;
                        dp[i][m] = max(dp[i][m], score);
                    }
                }
            }
        }

        return dp[0][1];
    }
};
```

Same Initialization & Pre-computation Part

all i

all possible (m) for those i

base case of recursion

try out every $1 \le x \le 2*m$

Computing everything backwards

Agar will take max

time : $O(n^3)$

space : $O(n^2)$

} same as Recursion + Memo

C++

```cpp
class Solution {
public:
    int helper(vector<int>& piles, vector<vector<int>>& dp, const vector<int>&
suffixSum, int i, int M) {
        if (i == piles.size()) return 0;
        if (i + 2 * M >= piles.size()) return suffixSum[i];
        if (dp[i][M] != 0) return dp[i][M];
        int result = 0;
        for (int x = 1; x <= 2 * M; ++x) {
            result = max(result, suffixSum[i] - helper(piles, dp, suffixSum, i + x,
max(M, x)));
        }
        dp[i][M] = result;
        return result;
    }
    int stoneGameII(vector<int>& piles) {
        if (piles.empty()) return 0;
        vector<vector<int>> dp(piles.size(), std::vector<int>(piles.size(), 0));
        vector<int> suffixSum(piles.size());
        suffixSum[suffixSum.size() - 1] = piles[piles.size() - 1];
        for (int i = piles.size() - 2; i >= 0; --i) suffixSum[i] = piles[i] +
suffixSum[i + 1];
        return helper(piles, dp, suffixSum, 0, 1);
    }
};
```

JAVA

```java
class Solution {
    public int helper(int[] piles, int[][] dp, int[] suffixSum, int i, int M) {
        if (i == piles.length) return 0;
        if (i + 2 * M >= piles.length) return suffixSum[i];
        if (dp[i][M] != 0) return dp[i][M];
        int result = 0;
        for (int x = 1; x <= 2 * M; ++x) {
            result = Math.max(result, suffixSum[i] - helper(piles, dp, suffixSum, i
+ x, Math.max(M, x)));
        }
```

```
        dp[i][M] = result;
        return result;
    }
    public int stoneGameII(int[] piles) {
        if (piles.length == 0) return 0;
        int[][] dp = new int[piles.length][piles.length];
        int[] suffixSum = new int[piles.length];
        suffixSum[suffixSum.length - 1] = piles[piles.length - 1];
        for (int i = piles.length - 2; i >= 0; --i) suffixSum[i] = piles[i] +
suffixSum[i + 1];
        return helper(piles, dp, suffixSum, 0, 1);
    }
}
```

PYTHON3

```
class Solution:
    def helper(self, piles, dp, suffixSum, i, M):
        if i == len(piles):
            return 0
        if i + 2 * M >= len(piles):
            return suffixSum[i]
        if dp[i][M] != 0:
            return dp[i][M]
        result = 0
        for x in range(1, 2 * M + 1):
            result = max(result, suffixSum[i] - self.helper(piles, dp, suffixSum,
i + x, max(M, x)))
        dp[i][M] = result
        return result
    def stoneGameII(self, piles):
        if not piles:
            return 0
        dp = [[0] * len(piles) for _ in range(len(piles))]
        suffixSum = [0] * len(piles)
        suffixSum[-1] = piles[-1]
        for i in range(len(piles) - 2, -1, -1):
            suffixSum[i] = piles[i] + suffixSum[i + 1]
        return self.helper(piles, dp, suffixSum, 0, 1)
```