1388. Pizza With 3n Slices

Hard Greedy Array Dynamic Programming Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, we're given a circular arrangement of pizza slices represented by an integer array `slices`, where each element in the array corresponds to the size of a pizza slice. The slices are laid out in clockwise order, and there is a total of `3n` slices, meaning the total number of slices is a multiple of three. The objective is to maximize the sum of sizes of slices that you can pick by following specified rules. The procedure for picking slices is as follows:

1. You pick any slice of the pizza.
2. Your friend Alice picks the next slice in an anti-clockwise direction from where you picked.
3. Your friend Bob picks the next slice in a clockwise direction from where you picked.
4. This process is repeated until no more slices are left.

The goal is to choose slices in such a way that the sum of the sizes of the slices you end up with is maximized.

Intuition

To approach this problem, we start by recognizing that it is a variation of the standard dynamic programming problem known as the "House Robber" problem. However, the twist here is that the arrangement of pizza slices is circular.

In a standard "House Robber" problem, you are trying to maximize the amount you rob from a line of houses without robbing two adjacent houses. Here, each pizza slice is like a "house," but since the pizza is circular, we have to prevent picking adjacent slices, and our choice wraps around the circle.

A key insight is that once we pick the first slice, the circular array problem reduces to two separate 'House Robber' problems:

1. One in which the first slice is included and the last slice is excluded (to maintain non-adjacency), varying the position of the first slice.
2. Another in which the first slice is excluded and the last slice is included.

We now can define a helper dynamic programming function `g(nums)`, which takes a list of slices represented as a linear array `nums` and computes the maximum sum of slices while adhering to the non-adjacent rule (similar to the 'House Robber' problem).

The state `f[i][j]` in the DP table represents the maximum sum we can obtain by considering up to the `i`-th pizza slice (0-indexed) and having picked `j` slices. For the `i`-th slice, we have two options: to take it or not. If we take it, we add its size to our total and look back two elements in the sequence to avoid adjacency, i.e., `f[i - 2][j - 1] + nums[i - 1]`. If we don't take it, we simply move to the previous element without changing the number of picked elements, i.e., `f[i - 1][j]`. We choose the option that yields the higher sum.

Finally, we run this helper function on the two separate scenarios (including the first slice and excluding the last, and vice-versa) and return the maximum sum obtained from both runs.

Solution Approach

The solution approach builds upon the dynamic programming pattern of solving optimization problems, particularly those related to making choices in sequences with constraints, like the "House Robber" problem. In this problem, the circular nature of the pizza slices array imposes a unique constraint, which we address by dividing the problem into two linear subproblems.

The helper function `g(nums)` takes as input a linear array of pizza slices sizes and returns the maximum sum we can achieve by picking non-adjacent slices. It uses a two-dimensional dynamic programming array `f`, where `f[i][j]` represents the maximum sum by considering up to the `i`-th slice (1-indexed in the explanation but 0-indexed in code) and having selected `j` slices.

The following algorithmic steps are taken within the `g(nums)` function:

1. Initialize a DP table `f` with dimensions `(n + 1) x (n + 1)`, where `n` is the length of the input list `nums`, and `n` is one-third of the length of the original `slices` array, representing the maximum number of slices you can choose.
2. Iterate over the slices array with an outer loop going from 1 to `n` and an inner loop going from 1 to `n`.
3. For each `i` and `j`, determine the maximum sum that can be achieved by either including or excluding the current slice:
   - If the slice is excluded, the value remains `f[i - 1][j]`.
   - If the slice is included, it's the sum of the current slice's value `nums[i - 1]` and the value from two steps back (to avoid adjacency) `f[i - 2][j - 1]`, or 0 if `i` is less than 2.
4. Ensure that each entry `f[i][j]` contains the maximum sum obtainable up to that point by choosing the maximum value between including and excluding the current slice.

After defining the helper function, we apply it to two scenarios: `slices[:-1]` and `slices[1:]`. This corresponds to two cases — when we include the first slice and exclude the last one, and when we exclude the first slice and include the last one. This effectively deals with the circular arrangement by treating it as two separate linear arrays.

The final result returned by the `maxSizeSlices` function is the maximum value obtained from both scenarios, achieved by calling `max(a, b)` where `a` and `b` are the results from the helper function `g` on the two subarrays.

In conclusion, the dynamic programming approach offers an efficient way to tackle this problem by reducing it to simpler subproblems and combining the results of these subproblems to obtain the final solution, which guarantees maximum sum of pizza slices you can pick.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have a circular array of pizza slices `slices = [1, 5, 3, 4, 2, 3]`, meaning there are `3n = 6` slices in total.

We will use the helper function `g(nums)` described in the solution approach to solve two separate 'House Robber' subproblems:

- The first subproblem includes the first slice and excludes the last, so we consider `nums = [1, 5, 3, 4]`.
- The second subproblem excludes the first slice and includes the last, so we consider `nums = [5, 3, 4, 2]`.

For each of these arrays, our dynamic programming table `f[i][j]` calculates the maximum sum obtainable by selecting `j` slices when considering up to the `i`-th slice. Let us walk through solving one of these subproblems: `nums = [1, 5, 3, 4]`.

We initialize a DP table `f` with dimensions `(4 + 1) x (2 + 1)`, since `n = 4` (the number of elements in `nums`) and `n = 2` (we can choose at most `6/3 = 2` slices).

The DP table is initially filled with zeros:

```
f = [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Now we start filling the table:

1. For `i = 1`, we cannot select the first slice as there are no previous slices to compare with. So `f[1][1]` would just be the value of the first slice (1 here).
2. For `i = 2`, we can either take the current slice (5) or the previous slice (1), but not both. We take 5 because it is the higher value: `f[2][1] = 5`.
3. As we move to `i = 3`, we look at including slice 3, which means we add 3 to `f[1][1]` (since selecting slice 3 means we cannot select slice 5), or we exclude it, which means we carry over the 5 from `f[2][1]`.

Here is the table with values filled:

```
1 1 *
1 5 5
1 5 8
1 5 8
1 5 9
1 5 9
```

The maximum sum we can obtain for `nums = [1, 5, 3, 4]` is `f[4][2] = 9`.

We would run the same process for `nums = [5, 3, 4, 2]` and let's say we get a maximum sum of `f[4][2] = 10`.

The result of our `maxSizeSlices` function will be the maximum value between the two subproblems, which is max(9, 10) = 10. Thus, by applying the solution approach on the given circular array of pizza slices, we deduce that the maximum sum of the sizes of the slices we can pick is 10.

By breaking it down into simpler, non-circular arrays and using the helper dynamic programming function `g(nums)`, we have managed to solve the problem efficiently.

Python Solution

```python
from typing import List

class Solution:
    def max_size_slices(self, slices: List[int]) -> int:
        # Helper function to calculate the maximum sum of non-adjacent slices
        def g(nums: List[int]) -> int:
            slice_count = len(nums)
            # The number of slices to be taken is one-third of the total number of slices provided
            take_slices = len(slices) // 3
            # Initialize a DP array to store subproblem solutions
            dp = [[0] * (take_slices + 1) for _ in range(slice_count + 1)]

            # Populate the dp array with maximum sum values for each subproblem
            for i in range(1, slice_count + 1):
                for j in range(1, take_slices + 1):
                    # Choose between not taking the current slice or taking it (and skipping the adjacent one)
                    dp[i][j] = max(
                        dp[i - 1][j],
                        dp[i - 2][j - 1] if i >= 2 else 0 + nums[i - 1]
                    )

            return dp[slice_count][take_slices]

        # Since we cannot take adjacent slices and we can't take the first and last together,
        # we perform the calculation twice—once without the first slice and once without the last slice
        max_sum_exclude_first = calculate_max_sum(slices[1:])
        max_sum_exclude_last = calculate_max_sum(slices[:-1])

        # Return the maximum of the two calculated sums
        return max(max_sum_exclude_first, max_sum_exclude_last)

# Example to use the class and method
# solution = Solution()
# print(solution.max_size_slices([1,2,3,4,5,6])) # Example usage with an array of pizza slices
```

Java Solution

```java
class Solution {
    private int n; // Number of slices to pick

    // Function to return the maximum sum of slices
    public int maxSizeSlices(int[] slices) {
        n = slices.length / 3; // Calculate the number of slices we can pick
        // Initialization an array to store slices without the first element
        int[] slicesWithoutFirst = new int[slices.length - 1];
        // Copy the array 'slices' to 'slicesWithoutFirst' starting from the second element
        System.arraycopy(slices, 1, slicesWithoutFirst, 0, slicesWithoutFirst.length);
        int maxSumExcludingFirst = calculateMaxSum(slicesWithoutFirst);

        // Initialization another array to store slices without the last element
        int[] slicesWithoutLast = new int[slices.length - 1];
        // Copy the array 'slices' to 'slicesWithoutLast' up to the second-to-last element
        System.arraycopy(slices, 0, slicesWithoutLast, 0, slicesWithoutLast.length);
        int maxSumExcludingLast = calculateMaxSum(slicesWithoutLast);

        // Return the maximum sum of two cases (excluding the first slice or the last slice)
        return Math.max(maxSumExcludingFirst, maxSumExcludingLast);
    }

    // Helper function to calculate the maximum sum using dynamic programming
    private int calculateMaxSum(int[] nums) {
        int numsLength = nums.length;
        // `dp` array where `dp[i][j]` represents the max sum using first `i` slices and `j` picks
        int[][] dp = new int[numsLength + 1][n + 1];
        for (int i = 1; i <= numsLength; ++i) {
            for (int j = 1; j <= n; ++j) {
                // Choose the bigger between not taking or taking the current slice
                dp[i][j] = Math.max(dp[i - 1][j],
                                    (i >= 2 ? dp[i - 2][j - 1] : 0) + nums[i - 1]);
            }
        }
        // Return the maximum sum for `numsLength` slices and `n` picks
        return dp[numsLength][n];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <cstring>

class Solution {
public:
    int maxSizeSlices(vector<int>& slices) {
        // Determine the size of a set (n) we want to calculate, which is one-third of the total slices
        int n = slices.size() / 3;

        // Lambda function to calculate the maximum size of pizza slices using dynamic programming
        auto getMaxSize = [&](vector<int>& nums) -> int {
            int m = nums.size(); // Number of elements in the current pizza slice array
            int dp[m + 1][n + 1]; // dp[i][j] will store the max size when considering i slices for j sets
            memset(dp, 0, sizeof(dp));

            // Initialize the dynamic programming array to 0
            memset(dp, 0, sizeof(dp));

            // Build the dp array using previous computed values
            for (int i = 1; i <= m; ++i) {
                for (int j = 1; j <= n; ++j) {
                    // Either take the current slice and add the best of i-2 for j-1 sets or skip the current slice
                    dp[i][j] = std::max(dp[i - 1][j], (i >= 2 ? dp[i - 2][j - 1] : 0) + nums[i - 1]);
                }
            }

            // Return the maximum size possible for the current pizza slices array
            return dp[m][n];
        };

        // Calculate the max size for two cases to avoid circle adjacency:
        // Case 1: Include all slices except the last
        vector<int> numsWithoutLastSlice(slices.end() - 1, slices.end());
        int maxWithoutLast = getMaxSize(numsWithoutLastSlice);

        // Case 2: Include all slices except the first
        vector<int> numsWithoutFirstSlice(slices.begin() + 1, slices.end());
        int maxWithoutFirst = getMaxSize(numsWithoutFirstSlice);

        // Return the maximum size by comparing the two cases
        return std::max(maxWithoutLast, maxWithoutFirst);
    }
};
```

Typescript Solution

```typescript
function maxSizeSlices(slices: number[]): number {
    // Calculate the number of slices to select, which is one third of the total slices.
    const selectCount = Math.floor(slices.length / 3);

    // Helper function to calculate the maximum size of slices from a given subarray.
    const calculateMaxSize = (nums: number[]): number => {
        const length = nums.length;
        const dp: number[][] = Array.from({ length: length + 1 }, () => Array(selectCount + 1).fill(0));

        // Populate the dynamic programming table with the maximum sizes.
        for (let i = 1; i <= length; ++i) {
            for (let j = 1; j <= selectCount; ++j) {
                dp[i][j] = Math.max(
                    dp[i - 1][j], // Previous state without current slice
                    (i >= 2 ? dp[i - 2][j - 1] : 0) + nums[i - 1] // Current slice added, only if we have more than 2 pizza slice (to skip adjacent)
                );
            }
        }

        // Return the maximum size for the given subarray
        return dp[length][selectCount];
    };

    // Calculate the maximum size by taking two cases
    // 1. Excluding the last slice
    // 2. Excluding the first slice
    // Since the pizza is circular, we cannot take the first and last slices together.
    const maxSizeExcludeLast = calculateMaxSize(slices.slice(0, -1));
    const maxSizeExcludeFirst = calculateMaxSize(slices.slice(1));

    // Return the maximum of the two cases
    return Math.max(maxSizeExcludeLast, maxSizeExcludeFirst);
}
```

Time and Space Complexity

The given code defines a function `g(nums: List[int]) -> int` that calculates the maximum sum of `n` non-adjacent elements in a list was using dynamic programming. This function is called twice on two different slices of the input list `slices` in order to account for the circular nature of the problem by excluding the first or the last element.

Time Complexity:

The time complexity of the function `g` is determined by the nested loops. There are two loops:

- The outer loop runs `n` times, where `n` is the length of the input list `nums`.
- The inner loop runs `t` times, where `n = len(slices) // 3`.

Inside the inner loop, only constant-time operations are performed. Therefore, the overall time complexity for function `g` is `O(n * m)`. Since `g` is called twice, once for `slices[:-1]` and once for `slices[1:]`, the time complexity of the entire `maxSizeSlices` function remains the same, as both slices have a length very close to the original length (`len(slices) - 1`). Thus, the time complexity for the `maxSizeSlices` function is also `O(n * m)`, where `n` is nearly equal to `len(slices)`.

Space Complexity:

Space complexity is determined by the amount of extra space used by the algorithm, which in this case is primarily due to the 2D array `f` defined in function `g`.

- The 2D array `f` has dimensions `(n + 1) * (m + 1)`, where `n` is the length of `nums` and `o = len(slices) // 3`.

Therefore, the space complexity of function `g` is `O(n * m)`. Since the space used by `f` is reused when `g` is called the second time, the space complexity of the entire `maxSizeSlices` function is also `O(n * m)`. Here, `n` is nearly equal to `len(slices)`.