1755. Closest Subsequence Sum

Hard | Bit Manipulation | Array | Two Pointers | Dynamic Programming | Bitmask | Leetcode Link

Problem Description

In this problem, you are given an array of integers `nums` and an integer `goal`. The objective is to select a subsequence from the array such that the absolute difference between the sum of the selected subsequence and the `goal` is minimized. In other words, you want to find a subsequence whose sum is as close as possible to the `goal`. A subsequence of an array can be derived by omitting any number of elements from the array, which could potentially be none or all of them.

Intuition

The direct approach to solve this problem would involve generating all possible subsequences of the given array `nums` and calculating the sum for each subsequence to see how close it is to the `goal`. However, this approach is not feasible because the number of possible subsequences of an array is `2^n`, which would result in exponential time complexity making it impractical for large arrays.

The intuition behind the provided solution is to use the "meet in the middle" strategy. This involves dividing the array `nums` into two nearly equal halves and then separately generating all possible sums of subsequences for each half. Once you have all subset sums from both halves, you can sort one of the halves (for example, the right half) to then use binary search to quickly find the best match for each subset sum from the other half (the left half). This way, you have reduced the original problem, which would have a complexity of `O(2^n)`, into two subproblems each having a complexity of `O(2^(n/2))`, which is significantly more manageable.

The `minAbsDifference` function first generates all possible subset sums for both halves of the array using the helper function `getSubSeqSum`. It stores these sums in two sets, `left` and `right`. After that, it sorts the sums from the right half to allow efficient searching. For each sum in the left set, the function computes the complement value needed to reach the `goal`. Using binary search, it then checks whether a sum in the right set exists that is close to this complement value. The result is the smallest absolute difference found during this pairing process between left and right subset sums.

Solution Approach

The solution approach consists of the following key parts:

1. **Divide the array into two halves**: First, the original array `nums` is split into two halves. This is a crucial step in the "meet in the middle" strategy.

2. **Generate all possible sums of both halves**: The method `getSubSeqSum` is recursively called to calculate all feasible subset sums for the two halves of the array, which are stored in the `left` and `right` sets. This is done through classic backtracking – for each element, we can either choose to include it in the current subset or not, leading to two recursive calls.

   The pseudo-code for subset sum generation would be similar to:

```
function getSubSeqSum(index, currentSum, array, resultSet):
    if index == length of array:
        add currentSum to resultSet
        return
    end if

    // Don't include the current element
    call getSubSeqSum(index + 1, currentSum, array, resultSet)

    // Include the current element
    call getSubSeqSum(index + 1, currentSum + array[index], array, resultSet)
end function
```

3. **Sort the sums of one half and use binary search**: After the generation of all subset sums for both halves, the sums from the right half are sorted to leverage binary search. The binary search allows us to find the closest element in `right` to the `goal - left_sum`, giving us the potential minimum difference.

   The logic for binary search comparison is as follows:

```
for each sum in left set:
    compute remaining = goal - sum
    find the index of the minimum element that is greater than or equal to remaining in the right
    if such an element exists, update result to minimum of result or absolute difference between that element and remaining
    if there is an element less than remaining (index > 0), update result to minimum of result or absolute difference between th
end for
```

4. **Calculate and return the result**: The overall minimum difference is tracked in the `result` variable, which is initialized with infinity (`inf`). It gets updated whenever a smaller absolute difference is found between a pair of left and right subset sums with respect to the `goal`.

By exploiting the "meet in the middle" strategy and binary search, we manage to reduce the time complexity of the problem from exponential to `O(n * 2^(n/2))`, which is much more efficient for inputs within the problem's constraints.

Example Walkthrough

Let's assume we have the following input:

`nums = [1, 2, 3, 4]` and `goal = 6`.

Following the solution approach described:

1. **Divide the array into two halves**: We split `nums` into `left_half = [1, 2]` and `right_half = [3, 4]`.

2. **Generate all possible sums of both halves**: We use the `getSubSeqSum` method.

   For the `left_half`:

```
Using index 0, with currentSum 0 -> left {0}
Include number at index 0 -> left {0, 1}
Using index 1, with currentSum 1 -> left {0, 1}
Include number at index 1 -> left {0, 1, 3}
Repeat without including number 1 -> left {0, 1, 3, 2}
Include number at index 1 -> left {0, 1, 3, 2, 3}
```

   So the possible sums for `left_half` are `{0, 1, 2, 3}`.

   For the `right_half`:

```
Using index 0, with currentSum 0 -> right {0}
Include number at index 0 -> right {0, 3}
Using index 1, with currentSum 3 -> right {0, 3}
Include number at index 1 -> right {0, 3, 7}
Repeat without including number 3 -> right {0, 3, 7, 4}
Include number at index 1 -> right {0, 3, 7, 4, 7}
```

   So the possible sums for `right_half` are `{0, 3, 4, 7}`.

3. **Sort the sums of one half and use binary search**: We sort the `right_half` sums to `{0, 3, 4, 7}` (although it's already sorted in this example).

   We perform binary searches for complement values (`goal - left_sum`) for each sum in the `left_half` sums.

```
left_sum = 0, remaining = goal - 0 = 6, closest in right is 7 (absolute difference 1)
left_sum = 1, remaining = goal - 1 = 5, closest in right is 4 or 7 (absolute differences 1 and 2)
left_sum = 2, remaining = goal - 2 = 4, closest in right is 4 (absolute difference 0)
left_sum = 3, remaining = goal - 3 = 3, closest in right is 3 (absolute difference 0)
```

4. **Calculate and return the result**: From the binary search steps, we find that the smallest absolute difference is 0, which can be achieved with `left_sum` of `2` and `right_sum` of `4` or `left_sum` of `3` and `right_sum` of `3`. Hence the minimum absolute difference between any subsequence sum and the `goal` is 0.

By using the described strategy, the problem that had a potentially exponential complexity is tackled in a much more manageable way, making it possible to solve efficiently even with larger inputs.

Python Solution

```python
from typing import List, Set
from bisect import bisect_left

class Solution:
    def minAbsDifference(self, nums: List[int], goal: int) -> int:
        # Split the array into two halves for separate processing
        n = len(nums)
        left_half_sums = set()
        right_half_sums = set()

        # Generate all possible sums of subsets for both halves
        self._get_subset_sums(0, 0, nums[:n // 2], left_half_sums)
        self._get_subset_sums(0, 0, nums[n // 2:], right_half_sums)

        # Initialize the result to infinity (unbounded)
        result = float('inf')
        # Sort the sums generated from the right half for binary search
        right_half_sums = sorted(right_half_sums)
        right_half_len = len(right_half_sums)

        # Iterate through every sum of the left half
        for left_sum in left_half_sums:
            remaining = goal - left_sum
            # Find the closest sum in the right half to the remaining goal
            idx = bisect_left(right_half_sums, remaining)

            # If the index is within the bounds, check if the sum reduces the absolute difference
            if idx < right_half_len:
                result = min(result, abs(remaining - right_half_sums[idx]))

            # Also check the number immediately before the found index to ensure minimum difference
            if idx > 0:
                result = min(result, abs(remaining - right_half_sums[idx - 1]))

        # Return the minimum absolute difference found
        return result

    def _get_subset_sums(self, index: int, current_sum: int, array: List[int], result_set: Set[int]):
        """Helper method to calculate all possible subset sums of a given array."""
        # If we've reached the end of the array, add the current sum to the result set
        if index == len(array):
            result_set.add(current_sum)
            return

        # Recursive call to include the current index element and to exclude it, respectively
        self._get_subset_sums(index + 1, current_sum, array, result_set)
        self._get_subset_sums(index + 1, current_sum + array[index], array, result_set)

```

Java Solution

```java
class Solution {
    public int minAbsDifference(int[] nums, int goal) {
        // Divide the array into two halves
        int n = nums.length;
        List<Integer> leftSums = new ArrayList<>();
        List<Integer> rightSums = new ArrayList<>();

        // Generate all possible sums in the first half of the array
        generateSums(nums, leftSums, 0, n / 2, 0);
        // Generate all possible sums in the second half of the array
        generateSums(nums, rightSums, n / 2, n, 0);

        // Sort the list of sums from the right half for binary search
        rightSums.sort(Integer::compareTo);

        // Initialize result with the highest possible value of integer
        int result = Integer.MAX_VALUE;

        // Iterate through each sum in the left half and use binary search to find
        // the closest pair in the right half that makes the sum close to the goal
        for (Integer leftSum : leftSums) {
            int target = goal - leftSum;
            int left = 0, right = rightSums.size();
            while (left < right) {
                int mid = (left + right) >> 1;
                if (rightSums.get(mid) < target) {
                    left = mid + 1;
                } else {
                    right = mid;
                }
            }
            // Check against the element on the right
            if (left < rightSums.size()) {
                result = Math.min(result, Math.abs(target - rightSums.get(left)));
            }
            // Check against the element on the left
            if (left > 0) {
                result = Math.min(result, Math.abs(target - rightSums.get(left - 1)));
            }
        }

        return result;
    }

    // Helper method to recursively generate all possible subset sums
    private void generateSums(int[] nums, List<Integer> sums, int start, int end, int currentSum) {
        if (start == end) {
            sums.add(currentSum);
            return;
        }

        // Don't include the current element
        generateSums(nums, sums, start + 1, end, currentSum);
        // Include the current element
        generateSums(nums, sums, start + 1, end, currentSum + nums[start]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minAbsDifference(vector<int>& nums, int goal) {
        int numsSize = nums.size(); // The size of the input array

        // Two vectors to store the subsets' sum for left and right subarrays
        vector<int> leftSums;
        vector<int> rightSums;

        // Generate all possible sums for the left and right halves
        generateSubsetsSum(nums, leftSums, 0, numsSize / 2, 0);
        generateSubsetsSum(nums, rightSums, numsSize / 2, numsSize, 0);

        // Sort the sums of the right subarray to utilize binary search later
        sort(rightSums.begin(), rightSums.end());

        // This variable will hold the minimum absolute difference
        int minDiff = INT_MAX;

        // For each sum in the left subarray, look for the closest sum in the right subarray
        // such that the sum of both is closest to the given goal
        for (int sumLeft : leftSums) {
            int target = goal - sumLeft; // The required sum from the right subarray

            // Perform binary search on rightSums to find an approximation of target
            int leftIndex = 0, rightIndex = rightSums.size();
            while (leftIndex < rightIndex) {
                int midIndex = (leftIndex + rightIndex) / 2;
                if (rightSums[midIndex] < target) {
                    leftIndex = midIndex + 1;
                } else {
                    rightIndex = midIndex;
                }
            }

            // Update minDiff with the closer of two candidates
            if (leftIndex < rightSums.size()) {
                minDiff = min(minDiff, abs(target - rightSums[leftIndex]));
            }
            if (leftIndex > 0) {
                minDiff = min(minDiff, abs(target - rightSums[leftIndex - 1]));
            }
        }

        // Return the minimum absolute difference found
        return minDiff;
    }

private:
    // Utility function to generate all possible subset sums using DFS
    void generateSubsetsSum(vector<int>& nums, vector<int>& sums, int startIndex, int endIndex, int currentSum) {
        // Base case: If starting index reached the end index, add the currentSum to sums
        if (startIndex == endIndex) {
            sums.push_back(currentSum);
            return;
        }

        // Exclude the current element and proceed to the next
        generateSubsetsSum(nums, sums, startIndex + 1, endIndex, currentSum);

        // Include the current element and proceed to the next
        generateSubsetsSum(nums, sums, startIndex + 1, endIndex, currentSum + nums[startIndex]);
    }
};

```

Typescript Solution

```typescript
// Utility function to generate all possible subset sums using DFS
function generateSubsetsSum(nums: number[], sums: number[], startIndex: number, endIndex: number, currentSum: number): void {
    // Base case: If starting index reached the end index, add the currentSum to sums
    if (startIndex === endIndex) {
        sums.push(currentSum);
        return;
    }

    // Exclude the current element and proceed to the next
    generateSubsetsSum(nums, sums, startIndex + 1, endIndex, currentSum);

    // Include the current element and proceed to the next
    generateSubsetsSum(nums, sums, startIndex + 1, endIndex, currentSum + nums[startIndex]);
}

// The main function to find the minimum absolute difference to the goal
function minAbsDifference(nums: number[], goal: number): number {
    let numsSize = nums.length; // The size of the input array

    // Two arrays to store the subsets' sum for left and right partitions
    let leftSums: number[] = [];
    let rightSums: number[] = [];

    // Generate all possible sums for the left and right halves
    generateSubsetsSum(nums, leftSums, 0, numsSize / 2, 0);
    generateSubsetsSum(nums, rightSums, numsSize / 2, numsSize, 0);

    // Sort the sums of the right partition to utilize binary search later
    rightSums.sort((a, b) => a - b);

    // This variable will hold the minimum absolute difference
    let minDiff = Number.MAX_SAFE_INTEGER;

    // For each sum in the left partition, look for the closest sum in the right partition
    for (let sumLeft of leftSums) {
        let target = goal - sumLeft; // The required sum from the right partition

        // Perform binary search on rightSums to find an approximation of target
        let leftIndex = 0;
        let rightIndex = rightSums.length;
        while (leftIndex < rightIndex) {
            let midIndex = Math.floor((leftIndex + rightIndex) / 2);
            if (rightSums[midIndex] < target) {
                leftIndex = midIndex + 1;
            } else {
                rightIndex = midIndex;
            }
        }

        // Update minDiff with the closer of two candidates
        if (leftIndex < rightSums.length) {
            minDiff = Math.min(minDiff, Math.abs(target - rightSums[leftIndex]));
        }
        if (leftIndex > 0) {
            minDiff = Math.min(minDiff, Math.abs(target - rightSums[leftIndex - 1]));
        }
    }

    // Return the minimum absolute difference found
    return minDiff;
}

```

Time and Space Complexity

The time complexity and space complexity analysis for the `minAbsDifference` function is as follows:

Time Complexity:

- The function `getSubSeqSum` generates all possible subsets' sums for the input subarrays. This function is called recursively for each element in the input subarray. There are `2^n` subsets possible for an array with `n` elements, resulting in a time complexity of `O(2^n)` for creating all subsets for an array.
- Since `getSubSeqSum` is first called with half the size of the input array `O(2^(n/2))`, and then with the remaining half `O(2^(n/2))`, the total time for the subset sum generation steps for both halves is `O(2*(2^(n/2)))` which simplifies to `O(2^(n/2 + 1))`.
- The set `right` is sorted afterwards which has at most `2^(n/2)` elements leading to a sort time complexity of `O(2^(n/2) * log(2^(n/2)))` which simplifies to `O((n/2) * 2^(n/2))`.
- Next, the for loop iterates over all elements of `left`, and for each element, a binary search is performed on `right` using `bisect_left`. This gives a time complexity of `O(2^(n/2) * log(2^(n/2)))` for the loop which simplifies to `O((n/2) * 2^(n/2))`.

The overall time complexity can be expressed as `O(2^(n/2 + 1) + (n/2) * 2^(n/2) + (n/2) * 2^(n/2))` which simplifies to `O(n * 2^(n/2))`.

Space Complexity:

- The space required is to store all subset sums for both halves and considering the deepest recursion call stack. This leads to `O(2^(n/2))` for `left` and `O(2^(n/2))` for `right` sets separately.
- The recursion call stack of the function `getSubSeqSum` will go to a maximum depth of `n/2` in both halves, so the space used by the call stack is `O(n/2 + n/2)` which simplifies to `O(n)`.

Hence, the space complexity can be viewed as the maximum space consumed at any point, giving `O(2^(n/2) + 2^(n/2) + n)` which simplifies to `O(2^(n/2) + n)`.