Weekly Assignment 1: Solution
Total: 100

CS 2500: Algorithms

Due Date: September 3, 2024 at 11.59 PM

Problems

1. (a) **Brute-force algorithm explanation:**
The brute-force algorithm for string matching works by comparing the target string (of length m) to every possible substring of the text (of length n) starting at each position from 0 to $n-m$. For each starting position i, it checks whether the substring of the text starting at index i matches the target string. If a match is found, the algorithm returns the index of the first match. If no match is found after examining all possible starting positions, the algorithm returns that the target string is not present in the text.
Specifically, for each position i in the text, the algorithm compares each character in the substring of length m starting at i with the corresponding character in the target string. If all m characters match, then a match is found. Otherwise, it moves to the next starting position in the text.

   (b) **Pseudocode:**

Algorithm 1: Brute-Force String Matching

```
Function BruteForceStringMatch(text, target):
    n ← length of text
    m ← length of target
    for $i \leftarrow 0$ to $(n-m)$ do
        match_found ← True
        for $j \leftarrow 0$ to $(m-1)$ do
            if $text[i+j] \neq target[j]$ then
                match_found ← False
        if $match\_found == True$ then
            return $i$ First match found at index $i$
    return $-1$ No match found
```

The outer loop iterates through all possible starting positions in the text. The inner loop checks each character of the target string against the corresponding substring in the text. If all characters match, the index i is returned. If no match is found, the algorithm returns -1.

   (c) **Big-O time complexity:**
In the worst case, the brute-force algorithm must check each possible starting position in the text (there are $n-m+1$ such positions) and, at each position, compare all m characters of the target string with the corresponding substring in the text. Each comparison takes $O(1)$ time, and there are m comparisons for each starting position. Therefore, the total number of comparisons

is proportional to $(n - m + 1) \times m$, which simplifies to $O(n \times m)$. Thus, the worst-case time complexity of the brute-force string matching algorithm is $O(n \times m)$.

2. (a) **Prove that the algorithm uses $O(n^3)$ comparisons:**

   We will analyze the number of comparisons in the algorithm step-by-step. The algorithm initializes $m_{ij} = a_i$ if $j \geq i$, otherwise it initializes $m_{ij} = 0$. This initialization step takes $O(n^2)$ time since there are $n \times n$ entries in the matrix M.

   The algorithm then enters a nested loop structure:

   - The outer loop runs for $i = 1$ to n, so it runs n times.
   - The second loop runs for $j = i + 1$ to n, so for each i, this loop runs $(n - i)$ times.
   - The innermost loop runs for $k = i + 1$ to j, so for each (i, j), this loop runs $(j - i)$ times.

   In total, the number of comparisons made by the innermost statement $m_{ij} \leftarrow \min(m_{ij}, a_k)$ depends on how often the innermost loop runs for different values of i and j.

   Let's count the total number of comparisons:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=i+1}^{j} 1$$

   The innermost summation gives $j - i$, so the total number of comparisons is:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i) = \sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2}$$

   This sum evaluates to $O(n^3)$. Therefore, the algorithm uses at most $O(n^3)$ comparisons.

   (b) **Prove that the algorithm uses $\Omega(n^3)$ comparisons:**

   To show the algorithm uses at least $\Omega(n^3)$ comparisons, consider the following. Focus only on the cases where $i \leq \frac{n}{4}$ and $j \geq \frac{3n}{4}$ in the outer loops. In these cases, the number of iterations of the inner loop remains large. Specifically, when $i \leq \frac{n}{4}$ and $j \geq \frac{3n}{4}$, the number of comparisons in the innermost loop $m_{ij} \leftarrow \min(m_{ij}, a_k)$ is approximately proportional to $j - i$, which is at least $\frac{n}{2}$. There are $\frac{n}{4}$ such values of i and j, so the total number of comparisons made in these cases is:

$$\sum_{i=1}^{n/4} \sum_{j=3n/4}^{n} \frac{n}{2} = \Omega(n^3)$$

   Therefore, the algorithm uses at least $\Omega(n^3)$ comparisons.

   From parts (a) and (b), we can conclude that the algorithm uses $\Theta(n^3)$ comparisons.

3. In the context of analyzing the growth of complexity functions, we assume that $f(n)$ and $g(n)$ are always positive for the following reasons:

   (a) **Focus on asymptotic behavior**: Big-O notation is concerned with the behavior of functions as n grows large. For time and space complexity, $f(n)$ and $g(n)$ represent the number of operations or memory usage, which are always non-negative. These quantities grow as input size increases, so we only care about how fast they grow, not whether they are negative.

   (b) **Simplifies comparisons**: If $f(n)$ and $g(n)$ could take negative values, it would complicate the comparison of their growth rates. For example, one function might alternate between positive and negative values, making it harder to evaluate whether $|f(n)| \leq C \cdot |g(n)|$ holds for all n. By assuming both functions are positive, we can more easily compare their growth rates using inequalities.

   (c) **Physical interpretation of complexity**: In real-world computations, both time and space complexities are inherently positive quantities. Operations take time, and memory usage consumes space, both of which cannot be negative. Therefore, assuming that $f(n)$ and $g(n)$ are positive aligns with the practical nature of analyzing algorithms.

For these reasons, it is typical to assume that complexity functions are positive when using Big-O notation.

4. To show that $n^2 \log n \notin \Theta(n^2)$, we need to examine the growth rates of the two functions and determine whether they are asymptotically equivalent.

   A function $f(n)$ is in $\Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and k such that for all $n \geq k$:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

   This means that $f(n)$ grows at the same rate as $g(n)$ up to constant factors.

   Consider the ratio of the two functions:

$$\frac{n^2 \log n}{n^2} = \log n$$

   As $n \to \infty$, we have $\log n \to \infty$. This implies that the ratio $\frac{n^2 \log n}{n^2}$ is unbounded and grows without limit.

   Therefore, there is no constant c_2 such that:

$$n^2 \log n \leq c_2 \cdot n^2$$

   for sufficiently large n. The absence of such a constant means the upper bound required by the Θ notation cannot be satisfied.

   Because $n^2 \log n$ grows faster than any constant multiple of n^2, the two functions are not asymptotically equivalent. Thus,

$$n^2 \log n \notin \Theta(n^2).$$

Finding Unique Triplets Summing to Zero

Part 1: Coding Exercise

Function Implementation

```
def tripletSum(nums):
    nums.sort()
    result = []
    n = len(nums)
    for i in range(n - 2):
        # Skip duplicates for the first element
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        a = nums[i]
        left = i + 1
        right = n - 1
        while left < right:
            b = nums[left]
            c = nums[right]
            total = a + b + c
            if total == 0:
                result.append([a, b, c])
                # Skip duplicates for the second element
                while left < right and nums[left] == nums[left + 1]:
                    left += 1
                # Skip duplicates for the third element
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1
                left += 1
                right -= 1
            elif total < 0:
                left += 1
            else:
                right -= 1
    return result
```

Approach and Algorithm

The function `tripletSum(nums)` finds all unique triplets in the array `nums` that sum up to zero. The algorithm follows these steps:

1. **Sorting the Array:** The input array `nums` is sorted in ascending order. This facilitates the two-pointer technique and helps in easily skipping duplicates.
2. **Iterating Through the Array:** We iterate through the array with an index `i` from `0` to `n - 3` (inclusive).
3. **Skipping Duplicates for the First Element:** If the current element is the same as the previous element, we skip it to avoid duplicate triplets.
4. **Two-Pointer Technique:** For each `nums[i]`, we set two pointers:
    - `left` initialized to `i + 1`
    - `right` initialized to `n - 1`
5. **Finding Triplets:** While `left` is less than `right`, we calculate the sum `total = nums[i] + nums[left] + nums[right]`.
    - If `total == 0`, we have found a triplet. We add it to the `result` list.
    - We then skip duplicates for `nums[left]` and `nums[right]` to avoid duplicate triplets.
    - If `total < 0`, we increment `left` to increase the sum.
    - If `total > 0`, we decrement `right` to decrease the sum.
6. **Returning the Result:** After all iterations, we return the `result` list containing all unique triplets.

Challenges Encountered

The main challenge was to ensure that duplicate triplets are not included in the result. This was addressed by:

- Skipping duplicate elements for the first element (`nums[i]`).
- Skipping duplicates for the second (`nums[left]`) and third (`nums[right]`) elements after a triplet is found.

Part 2: Analysis

1. **Time Complexity Analysis**

   The time complexity of the implemented algorithm is $O(n^2)$, where n is the number of elements in the array `nums`.

   Explanation:

   - Sorting the array takes $O(n \log n)$ time.
   - The main loop runs $O(n)$ times.
   - Inside the main loop, the two-pointer technique takes $O(n)$ time for each iteration.
   - Therefore, the overall time complexity is $O(n \log n) + O(n^2) \approx O(n^2)$.

   Possibility of Improvement:

   It is challenging to improve the time complexity below $O(n^2)$ due to the need to examine all possible triplets. However, for specific cases or with additional constraints, optimization techniques or data structures might be applied, but generally, $O(n^2)$ is the optimal time complexity for this problem.

2. **Space Complexity Analysis**

   The space complexity of the solution is $O(n)$.

   Explanation:

   - Sorting the array in-place does not require additional space beyond $O(n)$.
   - The result list can contain up to $O(n^2)$ triplets in the worst case.
   - However, since only unique triplets are stored, and each triplet uses constant space, the additional space used is proportional to the number of unique triplets found.
   - Therefore, the space complexity is $O(n)$ for storing the result.

   Time-Space Trade-offs:

   There is a trade-off between time and space when considering storing duplicates or using hash sets to detect duplicates. In this implementation, we use sorting and in-place duplicate skipping to minimize additional space usage while maintaining efficient time complexity.

3. **Ensuring Unique Triplets**

   To ensure that the returned triplets are unique, the algorithm:

   - Sorts the input array, which groups duplicate elements together.
   - Skips duplicate elements for the first element (`nums[i]`) by checking if `nums[i]` is the same as `nums[i - 1]`.
   - After finding a valid triplet, skips duplicates for the second and third elements (`nums[left]` and `nums[right]`) by moving the pointers past any duplicate values.

   This approach ensures that each unique triplet is added to the result list only once.

4. **Handling Edge Cases**

   (a) **Array with Fewer Than Three Elements**

   If the array contains fewer than three elements, the function will return an empty list.

   Explanation:

   - The main loop runs from `i = 0` to `n - 3`.
   - If `n < 3`, the loop will not execute, and the empty `result` list will be returned.

(b) **All Elements Are the Same (e.g., [0, 0, 0])**
The algorithm will correctly identify the triplet [0, 0, 0] if the sum equals zero.
Explanation:

- After sorting, all elements are identical.
- The algorithm checks for duplicates and will still process the first occurrence.
- It will find that $0 + 0 + 0 = 0$ and add [0, 0, 0] to the result.
- Subsequent duplicates are skipped to avoid duplicate triplets.

(c) **Array Contains Only Positive or Only Negative Numbers**
The algorithm will return an empty list if no triplets sum to zero.
Explanation:

- For all positive numbers, any sum of three positive numbers will be positive.
- For all negative numbers, any sum of three negative numbers will be negative.
- Therefore, no triplets will sum to zero, and the function returns an empty list.

5. **Performance with Large Input Size (nums.length** $= 3000$**)**
As the size of `nums` increases, the time complexity becomes significant due to the $O(n^2)$ nature.
Potential Bottlenecks:

   - The nested loops resulting from the main loop and the while loop can lead to performance issues for large n.

   Possible Solutions:

   - Optimize code to reduce constant factors.
   - Utilize parallel processing if applicable.
   - Implement heuristics to skip unnecessary computations.

6. **Modifying the Algorithm for Pairs Summing to Zero**
To find pairs of numbers that sum to zero, we can:

   - Sort the array.
   - Use two pointers, one at the start and one at the end of the array.
   - Move pointers towards each other based on the sum.

   Complexity Comparison:

   - Finding pairs has a time complexity of $O(n)$ after sorting, which is $O(n \log n)$.
   - Finding triplets has a time complexity of $O(n^2)$.

   The complexity increases significantly when moving from pairs to triplets due to the additional loop required to consider all combinations.