HW6

Keerthana Golla

October 2023

Problem - 1

Analysis of Amortized Time Complexity

To establish the amortized time complexity of both operations $\alpha(i)$ and $\beta(i)$ as $O(1)$, a thorough analysis of their time complexities and their frequencies in a sequence of operations is required.

Let's begin by examining the $\alpha(i)$ operation. It is known that each $\alpha(i)$ operation takes $O(i)$ time. However, crucially, every $\alpha(i)$ operation is followed by at least i $\beta(i)$ operations. Consequently, the cumulative time spent on $\beta(i)$ operations succeeding an $\alpha(i)$ operation is $O(i)$.

Now, delving deeper into the sequence of operations, suppose we have a sequence S of m operations, where each $\alpha(i)$ operation is succeeded by a minimum of i $\beta(i)$ operations. In such a scenario, the number of $\alpha(i)$ operations is at most m/i since each $\alpha(i)$ operation is followed by at least i $\beta(i)$ operations. Thus, the total time spent on $\alpha(i)$ operations in sequence S is at most $\sum(m/i)$, where the summation encompasses all possible values of i.

To simplify this expression, recognizing that every $\alpha(i)$ operation is followed by at least i $\beta(i)$ operations, the total number of $\beta(i)$ operations in sequence S is at least m. Consequently, $\sum(m/i) \leq \sum(m/1) = m\sum(1/i)$, where the summation encompasses all possible values of i.

Now, evaluating the summation $\sum(1/i)$, known as the harmonic series, it's established that $\sum(1/i)$ is bounded by $O(\log n)$, where n is the number of terms in the series. In our context, the number of terms is m, leading to $\sum(1/i) = O(\log m)$.

Dominant Factor: The total amortized time is dominated by the $O(\log m)$ term arising from the $\alpha(i)$ operations.

Constant Factor: However, $O(\log m)$ is a logarithmic growth rate, and when considering it as a constant factor in the context of amortized analysis, it behaves more like a constant than a function that grows with the input size. This is because logarithmic growth is much slower than linear or polynomial growth.

Effective Amortized Time: As $O(\log m)$ behaves like a constant in this context, we can effectively treat it as a constant factor. Therefore, the amortized time for both operations $\alpha(i)$ is effectively $O(1)$.

Average Cost per Operation: Despite the individual time complexity of $O(\log m)$ for $\alpha(i)$ operations, when amortized over the entire sequence of operations, the average cost per operation remains constant ($O(1)$). This means that, on average, each operation, whether $\alpha(i)$ or $\beta(i)$, takes constant time, regardless of the input size.

Similarly, demonstrating the $O(1)$ amortized time for $\beta(i)$ operations involves recognizing that each $\alpha(i)$ operation is followed by at least i $\beta(i)$ operations, and each $\beta(i)$ operation takes $O(1)$ time.

Operation $\alpha(i)$:

- Each $\alpha(i)$ operation takes $O(i)$ time.
- However, it is followed by at least i operations $\beta(i)$.
- The total time spent on these $\beta(i)$ operations is $O(i)$.

Sequence Analysis:

- Consider a sequence S of m operations, where each $\alpha(i)$ operation is succeeded by at least i operations $\beta(i)$.
- The number of $\alpha(i)$ operations in this sequence is at most m/i.
- Consequently, the total time spent on $\alpha(i)$ operations in sequence S is at most $\sum(m/i)$.

Simplification:

- Every $\alpha(i)$ operation is followed by at least i $\beta(i)$ operations, ensuring that the total number of $\beta(i)$ operations in sequence S is at least m.
- This simplifies the expression to $\sum(m/i) \leq \sum(m/1) = m\sum(1/i)$.

Harmonic Series:

- The evaluation of the harmonic series $\sum(1/i)$ is known to be $O(\log n)$, where n is the number of terms in the series. In this context, with m terms, $\sum(1/i) = O(\log m)$.

Amortized Time for $\alpha(i)$:

- The total time spent on $\alpha(i)$ operations in sequence S is at most $m\sum(1/i)$, resulting in $O(m \log m)$.
- Since m is the total number of operations, the amortized time for $\alpha(i)$ operations is $O(\log m)$ per operation.Like explained above , even though the individual time complexity for (i) operations is $O(\log m)$, when amortized over a sequence of operations, the average cost per operation remains constant, making the overall amortized time for operations $\alpha(i)$ is $O(1)$.

Operation $\beta(i)$:

- Every $\alpha(i)$ operation is followed by at least i operations $\beta(i)$.
- Each $\beta(i)$ operation takes $O(1)$ time.

Amortized Time for $\beta(i)$:

- The time complexity of $\beta(i)$ operations is $O(1)$ per operation.

Conclusion:

After considering the frequency and time complexity of each operation in the sequence, we've demonstrated that the amortized time for both operations $\alpha(i)$ and $\beta(i)$ is $O(1)$. This implies that, on average, the time taken by these operations remains constant regardless of the size of the input or the number of operations performed.

Problem - 2

(Remember this is problem solved in class, lets use same way to prove this and taking CLRS as reference we can prove the amortised time for INCREMENT and RESET) To implement the increment and reset operations efficiently, we can use a technique called the "bitwise counter with a carry bit." The key idea is to maintain a counter as an array of bits, and whenever an increment operation is performed, we set the lowest-order 0 bit to 1. If all bits are already 1, we perform a reset operation by setting all bits to 0 and then set the highest-order bit to 1.

Here is the algorithm:

```
class BitwiseCounter:
    def __init__(self, m):
        self.counter = [0] * m  # Initialize counter as an array of m bits
        self.highest_order_1 = -1  # Pointer to the highest-order 1 bit

    def increment(self):
        j = 0
        while j < len(self.counter) and self.counter[j] == 1:
            self.counter[j] = 0
            j += 1

        if j < len(self.counter):
            self.counter[j] = 1
            self.highest_order_1 = max(self.highest_order_1, j)

    def reset(self):
        for j in range(len(self.counter)):
            self.counter[j] = 0
        self.counter[self.highest_order_1] = 1

    def get_counter(self):
        return self.counter
```

Consider a bit array A, where each bit $A[i]$ undergoes a flip every time the INCREMENT operation is called. The flipping behavior of each bit is as follows:

Bit $A[0]$ flips every time INCREMENT is called.

Bit $A[1]$ flips every other time, with a total of $\left\lfloor \frac{n}{2} \right\rfloor$ flips in a sequence of n INCREMENT operations.

Bit $A[2]$ flips every fourth time, with a total of $\left\lfloor \frac{n}{4} \right\rfloor$ flips in a sequence of n INCREMENT operations.

In general, for i in the range 0 to $k-1$, bit $A[i]$ flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ times in a sequence of n INCREMENT operations on an initiall

For $i \geq k$, bit $A[i]$ does not exist, and therefore, it does not flip.

The total number of flips in the sequence is given by:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{k-1} \frac{1}{2^i} = n \left(1 - \frac{1}{2^k}\right) < 2n$$

The worst-case time complexity for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$. The average cost of each operation, and hence the amortized cost per operation, is $O(n/n) = O(1)$.

similarly we can prove for to RESET .let's prove the correctness and analyze the running time:

Correctness:

1. **Increment Operation:**
    - If there is a 0 bit, the increment operation will set the lowest-order 0 bit to 1.
    - If all bits are 1, the operation resets the counter and sets the highest-order bit to 1.
    - This ensures that the counter is incremented correctly.
2. **Reset Operation:**
    - The reset operation sets all bits to 0 and then sets the highest-order bit to 1.
    - This ensures that the counter is reset to 0.

Running Time Analysis:

1. **Amortized Time per Increment Operation:**
    - Each increment operation takes $O(1)$ time on average.
    - The operation may trigger a reset, but it only happens when all bits are 1.
    - The reset operation takes $O(m)$ time, but the frequency of resets is limited (at most once for every 2^m increments).
    - Therefore, the amortized time per increment is $O(1)$.
2. **Reset Operation:**
    - The reset operation takes $O(m)$ time, as it needs to set each bit to 0 and then set the highest-order bit to 1.
    - However, the reset operation is infrequent, and its cost is distributed over a sequence of increment operations.
    - Therefore, the amortized time per reset is $O(1)$.

In conclusion, the sequence of n increment and reset operations takes $O(1)$ amortized time per operation, satisfying the requirements. The key to achieving this efficiency is the strategy of resetting the counter only when necessary (i.e., when all bits are 1) and using the carry mechanism to efficiently handle increments.

Problem - 3

To solve this problem, we can use the concept of a "mother vertex" in a directed graph. A mother vertex is a vertex from which we can reach all other vertices in the graph through directed paths. The existence of a mother vertex implies that there is a vertex that can reach every other vertex.

Here's an algorithm to find a mother vertex in a directed graph represented as an adjacency list:

Algorithm 1 Finding a Mother Vertex

```
1:  procedure DFS(v, visited)
2:      visited[v] ← True
3:      for i in graph[v] do
4:          if not visited[i] then
5:              DFS(i, visited)
6:          end if
7:      end for
8:  end procedure
9:  procedure FindMotherVertex
10:     visited ← False × V
11:     last_visited ← 0
12:     for i in 0 to V − 1 do
13:         if not visited[i] then
14:             DFS(i, visited)
15:             last_visited ← i
16:         end if
17:     end for
18:     visited ← False × V
19:     DFS(last_visited, visited)
20:     if all(visited) then
21:         return last_visited
22:     else
23:         return −1                          ▷ No mother vertex found
24:     end if
25: end procedure
```

Explanation of the Algorithm:

1. Perform a Depth First Search (DFS) on the graph and mark the last visited vertex during the DFS traversal.
2. Reset the visited array and perform another DFS starting from the last visited vertex.
3. If all vertices are visited in the second DFS, then the last visited vertex is a mother vertex.
4. Return the last visited vertex if it is a mother vertex; otherwise, return -1.

Proof of Correctness for the Algorithm

Let the last finished vertex be v. Basically, we need to prove that there cannot be an edge from another vertex u to v if u is not another mother vertex (or there cannot exist a non-mother vertex u such that $u \to v$ is an edge). There can be two possibilities.

1. A recursive DFS call is made for u before v. If an edge $u \to v$ exists, then v must have finished before u because v is reachable through u, and a vertex finishes after all its descendants.

2. A recursive DFS call is made for v before u. In this case also, if an edge $u \to v$ exists, then either v must finish before u (which contradicts our assumption that v is finished at the end) OR u should be reachable from v (which means u is another mother vertex).

Proof of Claim 1: The algorithm always returns a vertex.

- The algorithm iterates through all vertices in the graph.
- During the first DFS traversal, it marks the last visited vertex.
- In the second DFS traversal, it checks if all vertices are visited.
- If all vertices are visited, it returns the last visited vertex.
- If no such vertex is found, it returns -1.

Therefore, the algorithm always returns a vertex.

Proof of Claim 2: The returned vertex is a "mother vertex."

- The first DFS traversal from any vertex marks the last vertex that can be reached from that vertex.
- The second DFS traversal starts from the last visited vertex in the first traversal.
- If there is a mother vertex, it means there is a vertex from which all other vertices can be reached.
- The last visited vertex in the first traversal is such a vertex.
- The second DFS traversal will visit all vertices reachable from the last visited vertex.
- If all vertices are visited in the second DFS traversal, the last visited vertex is a "mother vertex."

Therefore, the algorithm correctly identifies a "mother vertex" if one exists.
In conclusion, the algorithm is correct as it always returns a vertex, and if a vertex is returned, it is indeed a "mother vertex."

Analysis of the Running Time:

The time complexity of the algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges. The DFS traversal takes $O(|V| + |E|)$ time, and performing it twice does not change the overall time complexity.