Quick Assignment 1: Solution
Total: 100

CS 2500: Algorithms

Due Date: August 29, 2024 at 11.59 PM

Solutions

1. To show that the Algorithm `MaxElement` satisfies all the properties of an algorithm as discussed in Lecture 1, we can break down the properties and verify them against the algorithm:

   (a) **Input:**
   - An algorithm should have zero or more inputs.
   - **Verification:** In this case, the algorithm takes an array `a` of integers as input, and `n`, which is the number of elements in the array. Therefore, the input condition is satisfied.

   (b) **Output:**
   - An algorithm should produce at least one output.
   - **Verification:** The algorithm returns the value of `maxval`, which represents the maximum element in the array `a`. Thus, it satisfies the output condition.

   (c) **Definiteness:**
   - Each step of the algorithm must be precisely defined; the actions to be performed must be clear and unambiguous.
   - **Verification:**
     - The initialization step (`maxval` ← `a[0]`) is clear and definite.
     - The loop (`For i` ← `1 to n-1`) is well-defined, iterating through the array from the second element to the last.
     - The condition (`If a[i] > maxval`) is straightforward, and the update (`maxval` ← `a[i]`) is precise.
     - The return statement (`Return maxval`) is unambiguous.

     Therefore, all steps are definite, and the algorithm satisfies this property.

   (d) **Finiteness:**
   - The algorithm must always terminate after a finite number of steps.
   - **Verification:**
     - The loop runs exactly `n-1` times, where `n` is the length of the array. Each operation inside the loop is a basic operation that takes constant time.
     - After the loop finishes, the algorithm returns the maximum value and terminates.

     Since the number of iterations and steps are finite, the algorithm satisfies the finiteness condition.

(e) **Effectiveness:**
- The steps of the algorithm must be basic enough to be carried out, in principle, by a person using paper and pencil.
- **Verification:**
  – The operations performed by the algorithm (comparisons, assignments) are basic and can be executed manually if needed.

  Therefore, the algorithm is effective.

2. To show that $7n^2 \in O(n^3)$, we need to prove that there exist positive constants c and k such that:

$$7n^2 \leq c \cdot n^3 \quad \text{for all } n \geq k$$

- Start with the inequality:
$$7n^2 \leq c \cdot n^3$$
- Divide both sides by n^3 (assuming $n > 0$):
$$\frac{7n^2}{n^3} \leq c$$
$$\frac{7}{n} \leq c$$
- As n grows larger, $\frac{7}{n}$ becomes smaller. Thus, for $n \geq k = 1$, we can choose $c = 7$. This gives:
$$\frac{7}{n} \leq 7 \quad \text{for all } n \geq 1$$

Therefore, $7n^2 \in O(n^3)$.

To determine whether $n^3 \in O(7n^2)$, we need to check if there exist positive constants c' and k' such that:

$$n^3 \leq c' \cdot 7n^2 \quad \text{for all } n \geq k'$$

- Start with the inequality:
$$n^3 \leq c' \cdot 7n^2$$
- Divide both sides by n^2 (assuming $n > 0$):
$$n \leq 7c'$$
- The inequality $n \leq 7c'$ implies that n is bounded by a constant multiple of c'. However, as n becomes arbitrarily large, this inequality cannot hold because n grows without bound, while $7c'$ remains constant.

Therefore, it is **not** true that $n^3 \in O(7n^2)$.

3. Consider the sum:
$$S(n) = 1^k + 2^k + \cdots + n^k$$

This sum consists of n terms: $1^k, 2^k, \ldots, n^k$.

The idea is to compare each term in the sum with the largest term in the sum. The largest term in this sum is n^k because n^k is the value of i^k when $i = n$, and n is the largest value that i can take in the sum.

So, every term i^k in the sum $S(n)$ satisfies:

$$i^k \leq n^k$$

This inequality tells us that each of the terms $1^k, 2^k, \ldots, n^k$ is less than or equal to n^k.

Since each term i^k is at most n^k, we can say that:

$$S(n) = 1^k + 2^k + \cdots + n^k \leq n^k + n^k + \cdots + n^k \quad \text{(with } n \text{ terms)}$$

Notice that on the right-hand side, we have n terms, each equal to n^k.

So the sum of these n terms is:

$$S(n) \leq n \cdot n^k = n^{k+1}$$

We have shown that the entire sum $S(n)$ is bounded above by n^{k+1}, meaning that:

$$S(n) = 1^k + 2^k + \cdots + n^k \leq n^{k+1}$$

In Big-O notation, this means $S(n)$ is $O(n^{k+1})$.

4. The first algorithm performs $n^2 \cdot 2^n$ operations. Here, n^2 is a polynomial term and 2^n is an exponential term. Exponential functions grow much faster than polynomial functions as n increases. Therefore, the growth rate of $n^2 \cdot 2^n$ is dominated by the 2^n term.

   The second algorithm performs $n!$ operations. The factorial function $n!$ grows faster than both polynomial and exponential functions for large n.

   As n grows, the factorial function $n!$ eventually grows faster than $n^2 \cdot 2^n$. This means that, for sufficiently large n, the first algorithm (with $n^2 \cdot 2^n$ operations) will use fewer operations compared to the second algorithm (with $n!$ operations).

5. To show that $\log_2(x^2+1)$ and $\log_2 x$ are of the same order, we need to prove that:

$$\log_2(x^2+1) = O(\log_2 x) \quad \text{and} \quad \log_2 x = O(\log_2(x^2+1))$$

   To show $\log_2(x^2+1) = O(\log_2 x)$, we need to show that there exists a constant $c_1 > 0$ and a value x_0 such that for all $x \geq x_0$:

$$\log_2(x^2+1) \leq c_1 \cdot \log_2 x$$

   Since $x^2 + 1 \leq 2x^2$ for all $x \geq 1$, we have:

$$\log_2(x^2+1) \leq \log_2(2x^2) = \log_2 2 + \log_2(x^2) = 1 + 2\log_2 x$$

   This shows that:

$$\log_2(x^2+1) \leq 2\log_2 x + 1$$

   Thus, for $c_1 = 2$, we have:

$$\log_2(x^2+1) \leq c_1 \cdot \log_2 x + 1$$

   As x grows larger, the constant 1 becomes negligible, so:

$$\log_2(x^2+1) = O(\log_2 x)$$

   Next, to show $\log_2 x = O(\log_2(x^2+1))$, we need to show that there exists a constant $c_2 > 0$ and a value x_1 such that for all $x \geq x_1$:

$$\log_2 x \leq c_2 \cdot \log_2(x^2+1)$$

Since $x^2 + 1 \geq x^2$, we have:

$$\log_2(x^2 + 1) \geq \log_2(x^2) = 2\log_2 x$$

Thus:

$$\log_2 x \leq \frac{1}{2} \cdot \log_2(x^2 + 1) \leq 1 \cdot \log_2(x^2 + 1)$$

This shows that:

$$\log_2 x = O(\log_2(x^2 + 1))$$

Since $\log_2(x^2 + 1) = O(\log_2 x)$ and $\log_2 x = O(\log_2(x^2 + 1))$, we conclude that $\log_2(x^2 + 1)$ and $\log_2 x$ are of the same order. In Big-O notation, this can be expressed as:

$$\log_2(x^2 + 1) \sim \log_2 x$$

Note on the Base of the Logarithm

If instead of $\log_2(x^2 + 1)$, we had $\log_{10}(x^2 + 1)$, the overall proof would remain essentially the same. The base of the logarithm affects the constant factor but not the asymptotic behavior:

- By the change of base formula, $\log_{10}(x^2 + 1)$ can be expressed in terms of $\log_2(x^2 + 1)$ as:

$$\log_{10}(x^2 + 1) = \frac{\log_2(x^2 + 1)}{\log_2(10)}$$

- This means that $\log_{10}(x^2 + 1)$ and $\log_2(x^2 + 1)$ differ only by a constant factor $\frac{1}{\log_2(10)}$.
- Since Big-O notation abstracts away constant factors, the proof's conclusion remains the same: $\log_{10}(x^2 + 1)$ and $\log_{10} x$ (or $\log_2(x)$) would also be of the same order.

Thus, the asymptotic relationship between $\log(x^2 + 1)$ and $\log x$ does not depend on whether the logarithms are in base 2, base 10, or any other base.

6. Given a list of non-decreasing integers, we want to find a mode, which is the value that appears most frequently in the list.

   Algorithm:

   (a) **Initialize Variables:**
   - current_value to track the value currently being processed.
   - current_count to count occurrences of current_value.
   - mode_value to store the mode found so far.
   - mode_count to store the highest count of any value found so far.

   (b) **Iterate Through the List:**
   - Start from the first element of the list.
   - For each element in the list:
     - If the element is the same as current_value, increment current_count.
     - If the element is different from current_value:
       * Compare current_count with mode_count. If current_count is greater than mode_count, update mode_value to current_value and mode_count to current_count.
       * Update current_value to the new element and reset current_count to 1.

- After the loop, do a final check to compare current_count with mode_count to account for the last sequence of numbers.

(c) **Return** mode_value as the mode of the list.

Pseudocode: The algorithm is shown in Algorithm 1 below.

Time Complexity Analysis:

- **Initialization:** The initialization step takes constant time, $O(1)$.
- **Iteration Through the List:** The algorithm iterates through the list exactly once, performing constant-time operations at each step (comparison, assignment). Therefore, the time complexity for this step is $O(n)$, where n is the length of the list.
- **Final Comparison:** The final comparison after the loop is also a constant-time operation, $O(1)$.

Worst-Case Time Complexity: Since the dominant operation is the single pass through the list, the overall worst-case time complexity of the algorithm is $O(n)$.

This algorithm efficiently finds the mode in a list of non-decreasing integers with a linear time complexity, which is optimal for this problem.

Algorithm 1: Find Mode in a Non-Decreasing List

```
find_mode(list) if list is empty then
    return None;
current_value ← list[0];
current_count ← 1;
mode_value ← list[0];
mode_count ← 1;
for i ← 1 to length(list) - 1 do
    if list[i] == current_value then
        current_count ← current_count + 1;
    else
        if current_count ¿ mode_count then
            mode_value ← current_value;
            mode_count ← current_count;
        current_value ← list[i];
        current_count ← 1;
if current_count ¿ mode_count then
    mode_value ← current_value;
    mode_count ← current_count;
return mode_value;
```