Weekly Assignment 4

Munia Humaira
Student ID: 21116435
Algorithm Design and Analysis

October 4, 2024

Solution 1:

(a) Proof by induction:

1. **Base case ($x = 0$):**
   $f(0, y)$ should give the remainder when 0 is divided by y. This remainder is 0 since $0 = 0 \cdot y + 0$. The recurrence gives $f(0, y) = 0$. So, the base case holds.
2. **Induction step:**
   Assume, recurrence is correct for all values less than x where $x > 0$. To prove it's correct for x let's consider two cases:

   Case - 1: (x is even)
   Let $x = 2k$ for $k \in \mathbb{Z}_0^+$. By division algorithm, $k = qy + r$ where $r = f(k, y)$. Therefore,

   $$x = 2k = 2(qy + r) = 2qy + 2r$$

   And by induction hypothesis, $f(k, y) = f(x/2, y) = r$.

   If $2r < y$,

   $$x = 2qy + 2r = (2q)y + 2r$$

   Since $2r < y$, this is the correct remainder and the recurrence gives $2f(x/2, y) = 2r$.

   If $2r \geq y$,

   $$x = 2qy + 2r = (2q + 1)y + (2r - y)$$

   Since $2r \geq y$, $2r - y$ is the correct remainder and the recurrence gives $2f(x/2, y) - y = 2r - y$.

Case 2: x is odd
Let $x = 2k = 1$ for $k \in \mathbb{Z}_0^+$. By division algorithm, $k = qy + r$ where $r = f(k, y)$. Therefore,

$$x = 2k + 1 = 2(qy + r) + 1 = 2qy + 2r + 1$$

And by induction hypothesis, $f(x/2, y) = f(k, y) = r$.

If $2r + 1 < y$,

$$x = 2qy + 2r + 1 = (2q)y + 2r + 1$$

Since $2r + 1 < y$, this is the correct remainder and the recurrence gives $2f(x/2, y) + 1 = 2r + 1$.

If $2r + 1 \geq y$,

$$x = 2qy + 2r + 1 = (2q + 1)y + (2r + 1 - y)$$

Since $2r + 1 < y$, $(2r + 1 - y)$ is the correct remainder and the recurrence gives $2f(x/2, y) + 1 - y = 2r + 1 - y$.

In all cases, the recurrence correctly computes the remainder and hence, we can conclude by strong induction, the recurrence is correct for all $k \in \mathbb{Z}_0^+$.

(b) With binary encoding, n represents the total number of bits needed for input. For inputs x and y, $n = |x| + |y|$, where $|x|$ is the number of bits in x.
Analyzing every step of the algorithm:

Line 1: Constant time operation, $\mathcal{O}(1)$.

Line 2: A recursive call is made with $x/2$, which is a right shift and takes constant time, $\mathcal{O}(1)$.

Line 3: A multiplication, equivalent to a left shift by one bit and take constant time, $\mathcal{O}(1)$

Line 4: Checking even/odd is constant time operation, $\mathcal{O}(1)$

Line 5: Comparison($r \geq y$) and subtraction ($r \leftarrow r - y$) with y takes $\mathcal{O}(|y|)$ times.

Line 6: Return is also a constant time operation, $\mathcal{O}(1)$

Here, each level of recursion should have $\mathcal{O}(n)$ for checking, comparing and subtracting. And the depth of the recursion is $\mathcal{O}(n)$ as each recursive call halves the value of x which decreases the number of bits in x by 1.

So, the total time complexity is the product of work done in each step and the number of recursive levels:

$$\mathcal{O}(n) * \mathcal{O}(n) = \mathcal{O}(n^2)$$

Hence, *Div* runs in $\mathcal{O}(n^2)$ time where n is the size of the input in bits.

Solution 2:

The time complexity of the algorithm Alice proposed:

1. The algorithm initializes a variable s to 0 (which is a constant-time operation, $\mathcal{O}(1)$).
2. The **foreach** loop runs from 1 to n i. e., the loop executes total n iterations.
3. In each iteration of the loop, the algorithm performs a single addition operation ($s \leftarrow s + i$), which is also constant time, $\mathcal{O}(1)$.

So, total no. of operations is proportional to the no. of iterations, which is n, and the time complexity of this algorithm is $\mathcal{O}(n)$, since it performs a linear number of operations with respect to its input n.

An algorithm is polynomial-time if there exists a constant d such that the algorithm's time-efficiency can be meaningfully characterized as $\mathcal{O}(n^d)$, where n is the size of the input. Based on the above-mentioned definition, the time complexity $\mathcal{O}(n)$ is polynomial time and hence, $SumTon$ is a polynomial-time algorithm.

Solution 3:
Binary search divides the search range in half during each iteration. The no. of comparisons made to find an element depends on how deep we go into the binary search tree, which represents the sequence of comparisons. In each level of the binary search tree, half of the remaining items are eliminated, leaving fewer items to search through at each level.

Given, $n = 2^k - 1$, this creates a perfect binary tree structure.

Organizing the items by levels and their comparison counts:

Level 1: **1 item** needs **1** comparison

Level 2: **2 items** need **2** comparisons each

Level 3: **4 items** need **3** comparisons each

Level 4: **8 items** need **4** comparisons each
...

This continues till level k, where there are 2^{k-1} items, and we do k comparisons.

The expected number of comparisons C is the average number of comparisons across all possible items. The expected number of comparisons can be written as follow:

$$C = \frac{1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 8 \cdot 4 + ... + 2^{k-1} \cdot k}{n}$$

which can be written as:

$$C = \sum_{j=1}^{k} \frac{2^{j-1} \cdot j}{n}$$

where $n = 2^k - 1$

The sum $\sum_{j=1}^{k} 2^{j-1} \cdot j$ grows roughly up to $2^k * k$, which is proportional to $nlogn$. Dividing it by $n = 2^k - 1$, we get the expected number of comparisons $\mathcal{O}(logn)$

So, the expected number of times the comparison $A[mid] = i$ is performed is $\mathcal{O}(logn)$.