CSC 348 – Homework #7

Astrid Augusta Yu

June 5, 2020

Contents

1 Extra Algorithms and Theorems

1.1 Foreach Linearity Theorem

Definition 1. A recursive algorithm R is a **foreach algorithm**[1] it can be associated with a $n_0 \in \mathbb{N}$ (the base case size) and the following conditions are met:

- Some finite sequence of values A of size n such that $n \geq n_0$ is a parameter of that R. It may take in more parameters than just A, as well.
- At the beginning of R, it performs the check $n = n_0$. If this check passes, it terminates in $O(1)$ time.
- If the R fails that check, then in all of its following branches, it will:
  1. possibly perform an $O(1)$ operation
  2. call R, but with a new A such that $|A| = n - 1$
  3. terminate

Example 1.1. The MaxElement algorithm is a foreach algorithm because:

- It uses $n_0 = 1$
- It takes in a finite sequence $A = (a_1 \ldots a_n)$ of integers of size n
- It always returns a_1 when $n_0 = 1$
- If $n \neq n_0$, it will call MaxElement on a $(a_2 \ldots a_n)$, a list of size $n - 1$.

Theorem 1 (Foreach Linearity Theorem)**.** *If R is a foreach algorithm with base case n_0, A is a sequence such that $n \geq n_0$, then $R(A)$ has a time complexity of $O(n)$.*

Proof. By definition of a foreach algorithm, the base case when $n = n_0$ will immediately terminate in $O(1)$ time. Therefore,

$$T(n_0) = O(1) \tag{1}$$

Additionally, if it is not the base case, the foreach algorithm will undergo a $O(1)$ operation, then call itself on a $n - 1$ element sequence. Thus, for $n > n_0$,

$$T(n) = T(n-1) + O(1) \tag{2}$$

Now, we will prove by induction that $T(n) = (n - n_0 + 1) \cdot O(1)$.
Base case. Consider $n = n_0$. By (1),

$$T(n_0) = O(1)$$

.

Note that $n_0 - n_0 + 1 = 1$. Therefore,

$$T(n_0) = (n_0 - n_0 + 1) \cdot O(1)$$

[1] a term I literally just came up with

Induction hypothesis. Suppose that for some $k \geq n_0$, $T(k) = (k - n_0 + 1) \cdot O(1)$.
Inductive step. By (2),

$$T(k+1) = T(k) + O(1)$$

Applying the induction hypothesis,

$$T(k+1) = (k - n_0 + 1) \cdot O(1) + O(1)$$

which can be rewritten as

$$T(k+1) = ((k+1) - n_0 + 1) \cdot O(1)$$

Therefore, $T(n) = (n - n_0 + 1) \cdot O(1)$.
Note that the coefficient can be moved into the $O(1)$ like so:

$$T(n) = O(n - n_0 + 1)$$

Note that $-n_0 + 1$ is a constant. Therefore, $T(n) = O(n)$.

`d(^_^)>`

1.2 Map

Algorithm 1: Map(f, A)

Input: A function $f : \mathbb{Z} \to \mathbb{Z}$ and a sequence of integers $A = (a_1 \dots a_n)$
Output: The sequence of integers $(f(a_1), f(a_2), \dots f(a_{n-1}), f(a_n))$
1 **if** $n = 0$ **then**
2   **return** *()*
3 **else**
4   **return** $Map(f, (a_1 \dots a_{n-1})) \circ (f(a_n))$

1.2.1 Correctness

Lemma 1. *If A is a sequence of integers defined as $(a_1 \dots a_n)$, and there exists some $f : \mathbb{Z} \to \mathbb{Z}$, then $Map(f, A) = (f(a_i))_{i=1}^{n}$.*

Proof. We will proceed by induction.
Base case. Suppose $n = 0$. The algorithm will start at line 1. Since $n = 0 = 0$, the check passes and we proceed to line 2.
At line 2, the algorithm returns (), the empty sequence. This is correct because it equals the expected value $(f(a_i))_{i=1}^{0} = ()$.
Inductive hypothesis. Suppose $Map(f, (a_1 \dots a_k)) = (f(a_i))_{i=1}^{k}$.
Inductive step. Consider $n = k + 1$. At line 1, we check if $k + 1 = 0$. However, this can never happen, because if it were the case, then $k = -1$, which violates the fact that $k \in \mathbb{N}$.
Thus, the algorithm proceeds to line 4 via else and returns

$$Map(f, (a_1 \dots a_k)) \circ (f(a_{k+1}))$$

By the inductive hypothesis, this is equivalent to the sequence

$$(f(a_1))_{i=1}^{k} \circ (f(a_{k+1}))$$

which simplifies to

$$(f(a_1))_{i=1}^{k+1}$$

Therefore, by the principle of mathematical induction, $Map(f, A) = (f(a_i))_{i=1}^{n}$. d(^_^)>

Theorem 2. *Map is correct.*

Proof. By Lemma 1, Map is correct. d(^_^)>

1.2.2 Time complexity

Lemma 2. *Map is a foreach algorithm.*

Proof. Let $n_0 = 0$. The following are true about Map:

- It takes in a n-element sequence as its second parameter.
- It performs a check for $n = n_0 = 0$ at line 1. If it passes, it immediately returns.
- If the check fails, it calls Map on $(a_1 \dots a_{n-1})$, a $n-1$ element sequence, performs concatenation (a $O(1)$ operation), and immediately returns.

Therefore, Map is a foreach algorithm. d(^_^)>

Theorem 3. *Map is a $O(n)$ operation.*

Proof. By Lemma 2, Map is a foreach algorithm, so by the Foreach Linearity Theorem (Theorem 1), it has a time complexity of $O(n)$. d(^_^)>

1.3 Sum

Algorithm 2: Sum(A)

Input: A sequence of integers $A = (a_1 \dots a_n)$
Output: The value $\sum_{i=1}^{n} a_i$

```
if n = 0 then
    return 0
else
    return Sum((a_1 ... a_{n-1})) + a_n
```

1.3.1 Correctness

Lemma 3. *If the integer sequence* $A = (a_1 \dots a_n) = (a_i)_{i=1}^n$ *then* $Sum(A) = \sum_{i=1}^n a_i$.

Proof. We will proceed by induction over the input size.

Base case. Suppose $n = 0$. The algorithm will start at line 1, and because $n = 0 = 0$, it will pass the check.

The algorithm proceeds to line 2 and returns 0. Note that $\sum_{i=1}^0 a_i = 0$ because 0 is the additive identity.

Inductive hypothesis. Let $A = (a_1 \dots a_k) = (a_i)_{i=1}^k$. Suppose $Sum(A) = \sum_{i=1}^k a_i$.

Induction step. Let $B = (a_1 \dots a_{k+1}) = (a_i)_{i=1}^{k+1}$. Consider $Sum(B)$.

The algorithm begins at line 1. If $n = k + 1 = 0$, then $k = -1$, which is impossible, since $k \in \mathbb{N}$. Thus, the check at line 1 fails and the algorithm proceeds to line 4 due to else.

At line 4, the algorithm returns $Sum((a_1 \dots a_k)) + a_{k+1}$. By the inductive hypothesis,

$$Sum((a_1 \dots a_k)) + a_{k+1} = \sum_{i=1}^{k} a_i + a_{k+1} = \sum_{i=1}^{k+1} a_i$$

This can be further reduced to

$$\sum_{i=1}^{k} a_i + a_{k+1} = \sum_{i=1}^{k+1} a_i$$

Therefore, by the principle of mathematical induction, for all $n \in \mathbb{N}$, $Sum((a_i)_{i=1}^n) = \sum_{i=1}^n a_i$.
d(^_^)>

Theorem 4. *Sum is correct.*

Proof. By Lemma 3, Sum is correct. d(^_^)>

1.3.2 Time complexity

Lemma 4. *Sum is a foreach algorithm.*

Proof. Let $n_0 = 0$. The following are true about Sum:

- It takes in a n-element sequence as its second parameter.
- It performs a check for $n = n_0 = 0$ at line 1. If it passes, it immediately returns.
- If the check fails, it calls Sum on $(a_1 \dots a_{n-1})$, a $n - 1$ element sequence, performs addition (a $O(1)$ operation), and immediately returns.

Therefore, Sum is a foreach algorithm. d(^_^)>

Theorem 5. *Sum is a* $O(n)$ *operation.*

Proof. By Lemma 4, Sum is a foreach algorithm, so by the Foreach Linearity Theorem (Theorem 1), it has a time complexity of $O(n)$. d(^_^)>

Algorithm 3: ZipConsecutive(A)

Input: A sequence of integers $A = (a_1 \dots a_n) = (a_i)_{i=1}^{n}$ where $n \geq 1$

Output: The sequence of ordered integer pairs $((a_i, a_{i+1}))_{i=1}^{n-1}$, where each pair contains two consecutive numbers from the original sequence

```
1 if n = 1 then
2 |   return ()
3 else
4 |   return ((a_1, a_2)) ∘ ZipConsecutive((a_i)_{i=2}^n)
```

1.4 ZipConsecutive

1.4.1 Correctness

Lemma 5. *If $A = (a_i)_{i+1}^{n}$ is an integer sequence and $n \in \mathbb{N}^+$, then*

$$ZipConsecutive(A) = ((a_i, a_{i+1}))_{i=1}^{n-1}$$

Proof. We will proceed by induction over n, the size of A.

Base case. Suppose $n = 1$. The algorithm starts at line 1, and because $n = 1 = 1$, it passes the check and proceeds to line 2.

At line 2, the empty list is returned. This is equivalent to $((a_i, a_{i+1}))_{i=1}^{-1} = ()$.

Inductive hypothesis. Suppose for some $k \in \mathbb{N}^+$,

$$ZipConsecutive((a_i)_{i+1}^{k}) = ((a_i, a_{i+1}))_{i=1}^{k-1}$$

.

for all integer sequences sequences $(a_i)_{i+1}^{k}$.

Inductive step. Consider $ZipConsecutive((a_i)_{i+1}^{k+1})$.

The algorithm begins on line 1 and checks $k + 1 = 1$. However, if this were true, then $k = 0$, which can never happen because $k \in \mathbb{N}^+$. Thus, it fails and proceeds to line 4 via else.

At line 4, the algorithm returns

$$((a_1, a_2)) \circ ZipConsecutive((a_i)_{i=2}^{k+1})$$

which, by the inductive hypothesis, is equivalent to

$$((a_1, a_2)) \circ ((a_i, a_{i+1}))_{i=2}^{k}$$

We can include the first term in the sequence like so:

$$((a_i, a_{i+1}))_{i=1}^{k}$$

Thus,

$$ZipConsecutive(A) = ((a_i, a_{i+1}))_{i=1}^{n-1}$$

for all $n \in \mathbb{N}^+$. d(^_^)>

Theorem 6. *ZipConsecutive is correct.*

Proof. By Lemma 5, ZipConsecutive is correct. d(^_^)>

1.4.2 Time Complexity

Lemma 6. *ZipConsecutive is a foreach algorithm.*

Proof. Let $n_0 = 0$. The following are true about ZipConsecutive:

- It takes in a n-element sequence as its second parameter.
- It performs a check for $n = n_0 = 0$ at line 1. If it passes, it immediately returns.
- If the check fails, it calls ZipConsecutive on $(a_1 \ldots a_{n-1})$, a $n-1$ element sequence, performs concatenation (a $O(1)$ operation), and immediately returns.

Therefore, ZipConsecutive is a foreach algorithm. d(^_^)>

Theorem 7. *ZipConsecutive is a $O(n)$ operation.*

Proof. By Lemma 6, ZipConsecutive is a foreach algorithm, so by the Foreach Linearity Theorem (Theorem 1), it has a time complexity of $O(n)$. d(^_^)>

2 Questions

2.1 Q1.

Algorithm 4: SumFirstN(n)

Input: Some $n \in \mathbb{Z}^+$
Output: The value $\sum_{i=1}^{n} i$
1 **return** $Sum((i)_{i=1}^{n})$

2.2 Q2.

Lemma 7. *If $n \in \mathbb{N}$, then $SumFirstN(n) = \sum_{i=1}^{n} i$.*

2.3 Q3.

Proof. At line 1, SumFirstN returns $Sum((i)_{i=1}^{n})$. This evaluates to $\sum_{i=1}^{n} i$ by definition of the Sum algorithm. Therefore, $SumFirstN(n) = \sum_{i=1}^{n} i$. d(^_^)>

2.4 Q4.

Define $f : \mathbb{Z} \to \{0, 1\}$ as follows:

$$f(x) = \begin{cases} 1 & if x < 0 \\ 0 & if x \geq 0 \end{cases}$$

Algorithm 5: CountNegatives(A)

Input: An integer sequence $A = (a_i)_{i=1}^{n}$
Output: The number of negatives in A.
return $Sum(Map(f, A))$

2.5 Q5.

Lemma 8. *If some sequence $A = (a_1 \ldots a_n) = (a_n)_{i=1}^n$ has k negatives in it, then*

$$CountNegatives(A) = k$$

Proof. The algorithm starts on line 1 and returns

$$Sum(Map(f, A))$$

which is equivalent to, by definition of A,

$$Sum(Map(f, f(a_i))_{i=1}^n))$$

By definition of Map, this is equivalent to

$$Sum((f(a_i))_{i=1}^n)$$

By definition of sum, this is equivalent to

$$\sum_{i=1}^{n} f(a_i)$$

Suppose B is a sequence that contains all the negative elemnts of A, and C is a sequence that contains every element of A not in B, and therefore, non-negative. This sum can be rewritten as

$$\sum_{b \in B} f(b) + \sum_{c \in C} f(c)$$

Since all $b \in B$ are negative and all $c \in C$ are non-negative, by definition of f, this is equivalent to

$$\sum_{b \in B} 1 + \sum_{c \in C} 0 = \sum_{b \in B} 1$$

Since there are k negatives in A, B has k elements. Therefore,

$$\sum_{b \in B} 1 = k$$

Therefore, $CountNegatives(A) = k$. d(^_^)>

2.6 Q6.

Let $f : \mathbb{Z}^2 \to \mathbb{Z}^+$ be defined as follows:

$$f((a, b)) = |a - b|$$

Algorithm 6: LargestDiff(A)

Input: An integer sequence $A = (a_i)_{i=1}^n$
Output: The largest difference between any two consecutive numbers.
return $MaxElement(Map(f, ZipConsecutive(A)))$

2.7 Q7.

Lemma 9. *Suppose the integer sequence* $A = (a_i)_{i=1}^{n}$*, and* $p, q \in \mathbb{N}$ *such that* $1 \leq p < n$ *and* $1 \leq q < n$*. If for some* p*,* $|a_p - a_{p+1}| \geq |q_j - a_{q+1}|$ *for all possible values of* q*, then* $LargestDiff(A) = |a_p - a_{p+1}|$.

Proof. d(^_^)>

2.8 Q8.

2.8.1 Q8a.

Let $f : \mathbb{Z} \to \mathbb{Z}$ be defined as

$$f(x) = 2^x$$

Algorithm 7: Power2Sum(n)

Input: Some $n \in \mathbb{N}$

Output: The sum of all powers of 2 from 0 to n.

1 **return** $Sum(Map(f, (i)_{i=0}^{n}))$

2.8.2 Q8b.

Lemma 10. *If* $n \in \mathbb{N}$*, then*

$$Power2Sum(n) = \sum_{i=0}^{n} 2^i$$

Proof. Power2Sum starts at line 1 and returns $Sum(Map(f, (i)_{i=0}^{n}))$.

By the definitions of Map and f, this is equivalent to

$$Sum((2^i)_{i=1}^{n})$$

By definition of Sum, this is equivalent to

$$\sum_{i=0}^{n} 2^i$$

Thus, $Power2Sum(n) = \sum_{i=0}^{n} 2^i$. d(^_^)>

2.9 Q9.

Algorithm 8: ConstantPower2Sum(n)

Input: Some $n \in \mathbb{N}$

Output: The sum of all powers of 2 from 0 to n.

return $2^n - 1$

2.10 Q10.

2.10.1 Q10a.

Theorem 8. *SumFirstN runs in $O(n)$ time.*

Proof. At line 1, it generates a sequence of size n, which is $O(n)$, and executes Sum on it, a $O(n)$ algorithm, and finally it returns. Thus, its time complexity is

$$O(n+n) = O(n)$$

`d(^_^)>`

2.10.2 Q10b.

Theorem 9. *CountNegatives runs in $O(n)$ time.*

Proof. At line 1, it executes Map on a n-length sequence, which is a $O(n)$ operation. Then, it executes Sum on the resulting sequence, also $O(n)$. Finally, it returns. Thus, its time complexity is

$$O(n+n) = O(n)$$

`d(^_^)>`

2.10.3 Q10c.

Theorem 10. *LargestDiff runs in $O(n)$ time.*

Proof. At line 1, it executes ZipConsecutive on a n-length sequence, which is a $O(n)$ operation that outputs a $(n-1)$-length sequence. Next, it executes Map on that sequence, which is $O(n-1)$ and outputs a $n-1$ length sequence. Then, it executes MaxElement on the result from that, and MaxElement is a $O(n-1)$ sequence. Finally, it returns. Thus, its time complexity is

$$O(n+(n-1)+(n-1)) = O(n)$$

`d(^_^)>`

2.10.4 Q10d.

Theorem 11. *ConstantPower2 runs in $O(1)$ time.*

Proof. At line 1, it performs an exponentiation and a subtraction, which can each be considered $O(1)$ operations. Then, it returns. Therefore, it runs in

$$O(1+1) = O(1)$$

`d(^_^)>`

2.11 Q11.

2.11.1 Q11a.

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

Tree Method

$$\log_2(n) \text{ occurences} \begin{cases} O(n) \\ O(\frac{n}{2}) \\ O(\frac{n}{4}) \\ \vdots \\ O(4) \\ O(2) \\ O(1) \end{cases}$$

$$T(n) = O\left(\sum_{i=0}^{\log_2(n)} \left(\frac{1}{2}\right)^i \cdot n\right)$$

Note that $\sum_{i=0}^{\log_2(n)} \left(\frac{1}{2}\right)^i$ converges to 1. Therefore,

$$T(n) = O(n)$$

Master Theorem

Let $a = 1$, $b = 2$, and d $= 1$. We can rewrite $T(n)$ as

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Note that $d = 1 > log_b(a) = 0$. Therefore, by the Master Theorem,

$$T(n) = O(n)$$

2.11.2 Q11b.

$$T(n) = T\left(\frac{n}{2}\right) + O\left(n^2\right)$$

Tree Method

$$\log_2(n) \text{ occurences} \begin{cases} O(n^2) \\ O(\frac{n^2}{4}) \\ O(\frac{n^2}{16}) \\ \vdots \\ O(16) \\ O(4) \\ O(1) \end{cases}$$

$$T(n) = O\left(\sum_{i=0}^{\log_2(n)} \left(\frac{1}{4}\right)^i \cdot n^2\right)$$

Note that $\sum_{i=0}^{\log_2(n)} \left(\frac{1}{4}\right)^i$ converges to a finite value. Therefore,

$$T(n) = O(n^2)$$

Master Theorem

Let $a = 1$, $b = 2$, and d $= 2$. We can rewrite $T(n)$ as

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Note that $d = 2 > log_b(a) = 0$. Therefore, by the Master Theorem,

$$T(n) = O(n^2)$$

2.11.3 Q11c.

$$T(n) = T(n-2) + O(1)$$

Tree Method

$$\frac{n}{2} \text{ occurences} \begin{cases} O(1) \\ O(1) \\ \vdots \\ O(1) \\ O(1) \end{cases}$$

Thus,

$$T(n) = O(n)$$

Master Theorem

T cannot be written in the form of

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

because what should be its $\frac{n}{b}$ term is of the form $n-2$. Therefore, the Master Theorem is not applicable here.