**COMP 212 : Functional Programming, Spring 2023**

**Homework 08**

Name: Samir Cerrato

Wes Email: scerrato@wesleyan.edu

|  |  |  |
| --- | --- | --- |
| Question | Points | Score |
| 1 | 12 |  |
| 2 | 20 |  |
| Total: | 32 |  |

If possible, please type/write your answers on this sheet and upload a copy of the PDF to your google drive handin folder. Otherwise, please write the answers in some sort of word processor and upload a PDF. Please name the file hw08-written.pdf.

1. **Analysis**

(a) The following append function was a task in lab (see the lab handout and the lecture notes for Lect 15-16 for an explanation of how tabulate works):

fun myAppend (s1 : ’a Seq.seq, s2 : ’a Seq.seq) : ’a Seq.seq =

Seq.tabulate (fn i => case i < Seq.length s1 of true => Seq.nth (i, s1)

| false => Seq.nth (i - (Seq.length s1), s2), Seq.length s1 + Seq.length s2)

(2) i. Give a tight *O*-bound for the work of myAppend. Make sure you explicitly state what quantities you are analyzing the work in terms of. Briefly explain why your answer is correct.

**Solution:**

Length = s1 + s2 = n

Work of myAppend: Wmyppend(s1,s2) = Wtabulate(Wp\_function, n)

The work of this function is determined using the tabulate function’s parameters.

Wp\_function = k0 + Wseq.nth(I, max(s1,s2))

Wp\_function = k0 + k1 because Wseq.nth(I, max(s1,s2)) evaultes to k1

Wp\_function = k0 + k1 = k2 -> the runtime is shown to be O(1)

The work of the myAppend function can be found after finding the work of the function inside

Tabulate.

WmyAppend(s1, s2) = Wtabulate(k2, n)

WmyAppend(s1, s2) = k2 \* n

The work is the length of (s1+s2) times the work of the function inside the tabulate function, making the runtime be O(n).

(2) ii. Give a tight *O*-bound for the span of myAppend. Make sure you explicitly state what

quantities you are analyzing the span in terms of. Briefly explain why your answer is correct.

**Solution:**

S is either s1 or s2 and Length = s1 + s2 = n

Span for myAppend: Smyappend(s1, s2) = Stabulate(Sp\_function, n)

We find the span of the function within the tabulate

Sp\_function = k0 + (max of i of) Sseq.nth(i, s)

Sp\_function = k0 + k1 because the max evaultes to k1

Sp\_function = k0 + k1 = k2 -> the run time is O(1)

The span of the myAppend function can be found after finding the span of the function inside

Tabulate.

SmyAppend(s1, s2) = Stabulate(k2, i)

SmyAppend(s1, s2) = k2

The tabulate function could be parallel and thus, the span is the max of span. The elements are spanned acrossed s and the span of myAppend is the same as the tabulate function, meaning the runtime is O(1).

(b) Consider the following reverse function:

fun reverse’ (s : ’a Seq.seq) : ’a Seq.seq =

Seq.reduce (fn (x,y) => myAppend (y, x),

Seq.empty(),

Seq.map (Seq.singleton, s))

Seq.singleton and Seq.empty take constant time.

(2) i. Give a tight *O*-bound for the work of reverse’, in terms of the length of s.

Briefly explain your answer.

**Solution:**

The work for reverse can be found using Wreverse = Wmap + Wreduce. Wmap will have a runtime of O(s) because we are given that Seq.singleton takes constant time. That function runs an s amount of times inside of the map function that is called. The map is a tree and the leaves represent elements. As we descend the tree, the number of nodes increases because the each node gets split into two separate nodes and so on (ex: 1 -> 0.5, 0.5 -> 0.25, 0.25, 0.25, 0.25 …). The myAppend function finds the work being done and we’ve concluded that to be O(s). It will do this work for each level of the tree and the number of levels in a tree is determined by log(s). The product is then s \* log(s) and this is the work of the reduce function. Plugging the values into the equation in the beginning, we get Wreverse = Wmap + Wreduce becomes Wreverse = O(s) + O(s \* log(s)). Because the s \* log(s) half is the leading factor, we get a big O notation of n(s \* log(s)).

(2) ii. Give a tight *O*-bound for the span of reverse’, in terms of the length of s.

Briefly explain your answer.

**Solution:**

The span for reverse can be found using Sreverse = Smap + Sreduce. The span for Seq.singleton and the map function is O(1) because Seq.singleton can be parallelized. The span for reduce is O(log s) because to determine the span of a tree, you must account for the levels of the tree, which is determined by log(s). We multiply this span by the max span of Seq.singleton being O(1) and get a final span of O(log s) for the reduce function. Plugging the values into the equation in the beginning, we get Sreverse = Smap + Sreduce becomes Sreverse = O(1) + O(log s). Because log(s) is the leading factor, we are left with a big O notation of O(log s).

(c) Consider the following alternative implementation of the reverse function:

fun reverse (s : ’a Seq.seq) : ’a Seq.seq =

Seq.tabulate (fn i => Seq.nth ((Seq.length s) - (i + 1), s), Seq.length s)

(2) i. Give a tight *O*-bound for the work of reverse, in terms of the length of s. Briefly explain why there is a discrepancy between this and the work of reverse’.

**Solution:**

The runtime of Tabulate is O(1). The function is taking in each element of the sequence and thus, runs in constant time. Seq.length and Seq.nth are running in constant time and this means the parameter is constant. The length is s. We multiple the length by the constant run time from Seq.length and Seq.nth. Finally, we get a runtime of s\*1= O(s). Therefore, the work for reverse is O(s).

The discrepancy is that the reverse’ functions has an additional step of appending the elements from each level of the tree, which requires a runtime of O(s \* log(s)). In reverse, the elements are reversed using Seq.nth from the final value of the sequence and uses Seq.length to traverse all the elements of the sequence, running in constant time. Thus, reverse has a runtime of O(s) and not O(s\* log(s)).

(2) ii. Give a tight *O*-bound for the span of reverse, in terms of the length of s. Briefly explain why there is a discrepancy between this and the span of reverse’.

If we consider Seq.nth and Seq.length to be constant in terms of span, then we know the parameter has a constant runtime as well of O(1). The span of tabulate is dependent on the max span of the parameter, which is constant. Thus, the span of tabulate is O(1).

The discrepancy is that the reverse’ function must acknowledge the multiple levels of the tree which is log(s). The reverse function has no levels to take into account because it must only consider Seq.length that has a runtime of O(1). This is because it needs to traverse each element to find the length of the sequence.

2. **NON-COLLABORATIVE CHALLENGE PROBLEM: Tree Shrinking**

**Remember that non-collaborative challenge problems are to be done independently. You are not allowed to communicate with anyone about the problems, except to ask the instructor or TAs clarification questions (not hints). Additionally, you are not allowed to search for help on the specific problem from any sources besides the course materials.**

Sometimes, a computation will produce a tree with patterns like Node(Empty,t) or Node(t,Empty) in it, which can be optimized to just t without changing the contents of the tree. The following function does this:

fun shrink (t : ’a tree) : ’a tree = case t of

Empty => Empty

| Leaf x => Leaf x

| Node (l,r) => (case (shrink l, shrink r) of

(Empty, r’) => r’

| (l’,Empty) => l’

| (l’,r’) => Node(l’,r’))

That is, if either subtree of a tree shrinks to the empty tree, we delete that node, and otherwise we make a node of the shrunken subtrees.

In homework 7, you implemented reduce for trees.

fun reduce (n : ’a \* ’a -> ’a, e : ’a, t : ’a tree) : ’a = ...

In this problem, you will prove that the behavior of reduce is unchanged by shrinking:

**Theorem 1.** *Suppose we have total n:’a \* ’a -> ’a and e:’a such that*

1. *For all x, n(e,x)*∼= *x*
2. *For all x, n(x,e)*∼= *x*

*Then for all t, reduce(n, e,, shrink t)*∼= *reduce(n, e, t)*

(20) (a) Prove this theorem by induction on  *t.*

**WTS:**  *reduce(n, e, shrink t)*∼= *reduce(n, e, t)*

Case for Node (t1, t2):

**Inductive Hypothesis:**

Reduce (n, e, shrink l) ~= reduce (n, e, l)

Reduce (n, e, shrink r) ~= reduce (n, e, r)

**Want to show:**

Reduce (n, e, shrink (node(t1,t2)) ~= reduce (n, e, node(t1,t2))

**Proof**:

Shrink Node(t1,t2)

* Case node(l,r) of
* Case (shrink l, shrink r) of (Empty, r’) => r’
  + Reduce (n, e, shrink(Node(l,r)) = reduce (n, e, r’)
* N(e, reduce(n,e,shrinks(Node(l,r))) = n(e, reduce(n,e,r’)
* Reduce (n,e, shrink(Node(l,r))) ~= reduce(n,e,Node(l,r))
* Case(shrink l, shrink r) of. (l’, Empty) => l’ (follows the same concept as the previous case)
* Case(shrink l, shrink r) of (l’,r’) => Node(l’,r’)
  + Reduce (n,e,shrink(Node(l,r))) = n(reduce(n,e,l’), reduce(n,e,r’))
  + Reduce(n,e,Node(l,r)) = n(reduce(n,e,l), reduce(n,e,r))
  + N(reduce(n,e,l’), reduce(n,e,r’) ~= n(reduce(n,e,l), reduce(n,e,r))
  + Reduce (n,e,l’) ~= reduce(n,e,l)
  + Reduce (n,e,r’) ~= reduce(n,e,r)

Therefore,

N(reduce(n,e,l), reduce(n,e,r) ~= n(reduce(n,e,l’), reduce(n,e,r’))

Reduce(n,e,shrink(Node(l,r)) ~= reduce(n,e,Node(l,r))

**By I.H**