Assignment 1
SFWRENG 2CO3: Data Structures and Algorithms—Winter 2023

Problem 1

P1.1 Order the following functions of n on increasing growth rates and group functions with identical growth rates. Explain your answers.

$$\begin{array}{cccccc} ln(n^3) & n & n^2\log_2 n & 4^{\log_2 n} & \log_2\sqrt{n} & 2n\log_2 n \\ n+\log_2 n^4 & 2^{\log_2(16)} & n^{-1} & 16 & n^{\log_2 4} & \log_2 n^n \end{array}$$

Answer (Top = largest growth rate, bottom = smallest growth rate):

- $n^2\log_2 n$
- $4^{\log_2 n} = n^{\log_2 4}$
- $2n\log_2 n = \log_2 n^n$
- $n+\log_2 n^4 = n$
- $ln(n^3) = \log_2\sqrt{n}$
- $2^{\log_2(16)} = 16$
- n^{-1}

Reasoning

Going down the list, I took the limit of the top function divided by the bottom function and if it was ∞ then the top function had a greater growth rate, else if it was a constant > 0, then the functions were equivalent.

$\lim_{n\to\infty} \frac{n^2\log_2 n}{4^{\log_2 n}} = \infty$

$\lim_{n\to\infty} \frac{4^{\log_2 n}}{n^{\log_2 4}} = 1$

$\lim_{n\to\infty} \frac{n^{\log_2 4}}{2n\log_2 n} = \infty$

$\lim_{n\to\infty} \frac{2n\log_2 n}{\log_2 n^n} = 1$

$\lim_{n\to\infty} \frac{\log_2 n^n}{n+\log_2 n^4} = \infty$

$\lim_{n\to\infty} \frac{n+\log_2 n^4}{n} = 1$

$\lim_{n\to\infty} \frac{n}{ln(n^3)} = \infty$

$\lim_{n\to\infty} \frac{ln(n^3)}{\log_2\sqrt{n}} = 6ln(2)$

$\lim_{n\to\infty} \frac{\log_2\sqrt{n}}{2^{\log_2(16)}} = \infty$

$\lim_{n\to\infty} \frac{2^{\log_2(16)}}{16} = 1$

$\lim_{n\to\infty} \frac{16}{n^{-1}} = \infty$

P1.2 Assume n is an exact power of 2 ($n = 2^j$ for some natural number j) and consider the recurrence

$$T(n) = \left\{ \begin{array}{ll} 5 & \text{if } n = 1; \\ 4T(\frac{n}{2}) + 7n & \text{if } n > 1. \end{array} \right\} \tag{1}$$

Answer:
Use induction to prove that $T(n) = f(n)$ with $f(n) = 5n^2 + 7n(n-1)$.

Induction hypothesis: $T(n) = f(n)$ with $f(n) = 5n^2 + 7n(n-1)$ for all $n < i$

Base Case: $T(1) = f(1)$
$T(1) = 5$
$f(1) = 5(1^2) + 7(1)(1-1) = 5$

Step: Prove $T(i) = f(i), i > 1$ Filling in i in T(i), gives $T(i) = 4T(\frac{i}{2}) + 7i$ We have $\frac{i}{2} < i$. Hence induction hypothesis gives $T(\frac{i}{2}) = 5(\frac{i}{2})^2 + 7\frac{i}{2}(\frac{i}{2} - 1)$

$T(i) = 4T(\frac{i}{2}) + 7i = 4(5(\frac{i}{2})^2 + 7\frac{i}{2}(\frac{i}{2} - 1)) + 7i$
$= 4(5(\frac{i^2}{4}) + 7\frac{i}{2}(\frac{i}{2} - 1)) + 7i$
$= 5(i^2) + 28\frac{i}{2}(\frac{i}{2} - 1) + 7i$
$= 5(i^2) + 14i(\frac{i}{2} - 1) + 7i$
$= 5(i^2) + 7i(i-2) + 7i$
$= 5(i^2) + 7i((i-2) + 1)$
$= 5(i^2) + 7i(i-1)$

This proves that T(n) = f(n)

Problem 2

Assume we have a list L of unknown length that provides a constant-cost operation $Exists(L, i)$ that returns true if the list has an i-th element (if $0 \leq i < |L|$, where $|L|$ is the length of the list). For example, $Exists($ ['AValue', 'OtherValue'], 1) returns true and $Exists([`AValue', `OtherValue'], 2)$ returns false. The following algorithm computes the length of list L:

```
def ListLength(L):
    len = 0
    while Exists(L, len) do
        len = len + 1
    end while
    return len
```

P2.1. Prove that ListLength is correct and terminates and provide the complexity of ListLength.

Answer (Correctness and Termination):

```
    def ListLength(L):
        len = 0
        # Base case: len = 0 and L is an empty list implies inv. (|L| = len = 0)
        # inv: |L| >= len >= 0
        while Exists(L, len) do # bf: |L| - len
            # Given: invariant and Exists(L, len) -> L[len] exists
            len = len + 1
            # Known: |L| >= len and |L| > len_old
        end while
        # Known: |L| >= len and ¬Exists(L, len), which means |L| must be = len
        # Induction step: prove the invariant holds at each step
        return len  # |L| = len
```

Termination: bound function is $|L| - len$. When $|L| - len = 0$, then len is correct, anything past it will go into the negatives, which is not part of the natural numbers.

Answer (Complexity):

```
    def ListLength(L):
        len = 0 # 1 operation
        while Exists(L, len) do      # 1 check, 1 function call
            len = len + 1            # 1 operation, 1 assignment
        end while
        return len     # 1 return
```

Every loop would be 4 operations, then +2 operations for the whole function. Therefore, complexity would be 4N + 2, which is $\sim N$, where N = $|L|$

P2.2. Sketch an algorithm that can determine the length of list L in $\sim log_2(|L|)$ time. Include an argument as to why your algorithm is correct and terminates.

Answer:

```
def BetterListLength(L):
    if !Exists(L, 0) return 0;      # 1 check, 1 return
    start = 0    # 1 operation
    end = 1      # 1 operation
    # loop 1
    while Exists(L, end):     # 1 test
        start = end          # 1 assignment
        end = end * 2    # 1 operation, 1 assignment
    # loop 2
    while start+1 < end:       # 1 test, 1 operation
        middle = (start + end)//2   # 2 operations, 1 assignment
        if Exists(L, middle)    # 1 test
            start = middle    # 1 assignment
        else
            end = middle     # 1 assigment
    return start             # 1 return
```

To begin, for loop 1, its complexity is $\sim \log_2 |L|$ since end doubles every time until it's larger than $|L|$

For loop 2, its complexity would also be $\sim \log_2 |L|$, since it is similar to binary search. The difference between Loop 2 and binary search is the start and end values, and the conditional. Based off loop 1 we know $start = 2^{i-1}$ and $end = 2^i$. We also know $2^{i-1} < |L| <= 2^i$, where $i >= 0$ and $i \in \mathbb{N}$, so $start < |L| <= end$. And due to this, we can say $end - start < |L|$ as long as $|L| >= 0$. Now, given a start and end, we check if the middle value exists in the list. If it does, then we get rid of the first half of the values we are checking (by setting start = middle), else we get rid of the second half of the values we are checking (by setting end = middle). In that case, we are always dividing the values we are checking in half, which is a $\sim \log_2 |L|$ algorithm.

Given n = 2^i

Visually loop 2 will go like this:

Iteration 1: end - start = 2^i values

Iteration 2: end - start = $2^i - 1$ values

. . .

Iteration i: end - start = 2^0 values

Terminates

The height of this would be $\log_2 n = \log_2 2^i = i$. And since we know $|L| <= 2^i$, we also know $\log_2 |L| <= i$

Since loop 1 is $\sim \log_2 |L|$ and loop 2 is $\sim \log_2 |L|$ and they are consecutive, so the total complexity of the algorithm would be $\sim 2\log_2 |L|$ and that is equal to $\sim \log_2 |L|$ since we can drop the constants.

Also it's worth noting, for loop 1, there are 4 operations total per loop, and for loop 2, there are 7 operations per loop. There are also 4 extra operations, which would give us complexity of $4\log_2 |L| + 7\log_2 |L| + 4$. However, these numbers are negligible in the complexity calculation, so they were left out.

To double check, I implemented the code in Java. When implemented with Java code, when inputting an array of length 10,000 there are 28 loops that occur. Checking the boundaries of multiple numbers, I found that for: $2^{n-1} < x <= 2^n$, where $n >= 0$, there are 2n loops total in both while loops combined.

In terms of termination, for loop 1, the bound function is $|L| - end$, and we know end must at some point be bigger than $|L|$, thus L[end] won't exist, so it must terminate. For loop 2, we know at every loop, start either gets larger or end gets smaller until middle = (start + end)//2 = start or middle = end, which means $start + 1 = end$

Problem 3

P3.1. Consider an initially-empty stack S and the sequence of operations

$$PUSH(S, 4), PUSH(S, 19), POP(S), PUSH(S, 2), POP(S), POP(S), PUSH(S, 9)$$

Illustrate the result of each operation (clearly indicate the content of the stack after the operation and, in case of a pop, the value returned by the operation)

Answer:
Assume a stack is of the following form [..., ...] (Top).
For the following answer, the function call happens before it's respective stack.

Function call	Resulting Stack	Return value
$Initially$	[ ] (Top)	none
$PUSH(S, 4)$	[4]	none
$PUSH(S, 19)$	[4, 19]	none
$POP(S)$	[4]	19
$PUSH(S, 2)$	[4, 2]	none
$POP(S)$	[4]	2
$POP(S)$	[ ]	4
$PUSH(S, 9)$	[9]	none

P3.2. Consider an initially-empty queue Q and the sequence of operations

$$ENQUEUE(Q, 4), ENQUEUE(Q, 19), DEQUEUE(Q), ENQUEUE(Q, 2), DEQUEUE(Q),$$
$$DEQUEUE(Q), ENQUEUE(Q, 9)$$

Illustrate the result of each operation (clearly indicate the content of the stack after the operation and, in case of a $DEQUEUE$, the value returned by the operation).

Answer:
Assume a queue is of the following form (Bottom) [..., ...] (Top).
For the following answer, the function call happens before it's respective queue.

Function call	Resulting Queue	Return value
$Initially$	(Bottom) [ ] (Top)	none
$ENQUEUE(Q, 4)$	[4]	none
$ENQUEUE(Q, 19)$	[4, 19]	none
$DEQUEUE(Q)$	[19]	4
$ENQUEUE(Q, 2)$	[19, 2]	none
$DEQUEUE(Q)$	[2]	19
$DEQUEUE(Q)$	[ ]	2
$ENQUEUE(S, 9)$	[9]	none

P3.3. Consider bags of values B_1 and B_2. The SetUnion operation takes bags B_1 and B_2 and returns a bag holding all values originally in B_1 and B_2 (possibly destroying B_1 and B_2 in the process). Provide a data structure to represent B_1 and B_2 such that SetUnion can be implemented in constant time.

Answer:
The bag can be implemented using a singly linked list, with both a first and last pointer. In order to define SetUnion, you would just need one assignment operation that assigns $B_1.last.next$ to $B_2.first$, and one assignment to make $B_2.last$ be $B_1.last$ now. This would make it have a complexity of constant time as there are only 2 operations in the whole function. The function would look similar to:

Assuming B1 and B2 won't be destroyed

```
def SetUnion(B1, B2):
    B1.last.next = B2.first
    B1.last = B2.last
    return B1
```

Assuming B1 and B2 may be destroyed

```
def SetUnion(B1, B2):
    B3 = new LinkedList();
    B3.first = B1.first
    B1.last.next = B2.first
    B3.last = B2.last
    return B3
```

Note: I defined a new LinkedList as in the question, it mentions B1 or B2 can be destroyed, so setting it to a new B3 will ensure the return value will not be destroyed.

Problem 4

The main difference between an array and a linked list is that arrays support random access efficiently: one can lookup the i-th value in an array in constant time. Many algorithms such as BinarySearch rely on random access.
4.1. Provide an algorithm $Get(L, i)$ that returns the i-th value in linked list L. What is the complexity of this algorithm?

Answer:

```
def Get(L, i):
    head = L.first              # 1 assignment
    counter = 0                 # 1 assignment
    while (head != null):       # 1 check
        if (counter == i)       # 1 check
            return head.value   # 1 return
        head = head.next        # 1 assignment
        counter++               # 1 operation, 1 assignment
    return "not found"          # 1 return
```

Every loop there is 2 checks, 2 assignments, and 1 operation. There is 1 return that will be hit in the algorithm and 2 extra assignments.
Therefore, there is 5 operations per loop and +3 extra operations.
Complexity is 5N + 3 which is $\sim N$
NOTE: N = i where i is the i-th value passed in.

4.2. Consider the algorithm BinarySearch from the slides. What is the average complexity of this algorithm to search for a value $v \in L$ when list L is represented by a linked lists (using the above Get algorithm to implement $L[mid]$). Feel free to assume that the length of the list is an exact power of two if that simplifies your argument.

Answer:

```
def BinarySearch(L, start, end, v):
    if start + 1 = end then
        return start if L[start] = v
    else
        mid = (start + end)/2
        if (L[mid] <= v) then
            return BinarySearch(L, mid, end, v)
        else
            return BinarySearch(L, start, mid, v)
```

BinarySearch has a time complexity of $\sim \log_2 N$, where N = $|L|$ (length of the list). In this case, we have nested an n (where n is the index you pass into the Get method) time complexity algorithm in it to get L[mid], so it would be $\sim (mid) * \log_2 |L|$, where mid is some constant times $|L|$.

Outer loop average case complexity = $\sim \log_2 n$
Inner loop (The Get(L,i) function) =

$\sum_{i=0}^{|L|-1}$ p(i is @ n-th value of array) * (complexity of one Get(L,i) call)
$= \sum_{i=0}^{|L|-1} \frac{1}{|L|} * (5i + 3)$
$= \frac{|L|(|L|-1)}{2|L|}$
$= \frac{|L|-1}{2}$
$\sim |L|$

Since the inner loop is nested within the outer loop, we multiply the two complexities to find the complexity of our algorithm, which will give us $\frac{|L|-1}{2} * \log_2 |L| \sim |L| \log |L|$.