University of Nebraska-Lincoln

Assignment 1 (Written Portion)

Student name: *Justin Ho and Abby Seibel*

Course: *CSCE310 Data Structures and Algorithms* – Professor: *Dr. Christopher Bourke*
Due date: *20th of July 2020*

Question 1

Answer.

(a) Given that $a_k > 0$, then using limits

$$\begin{aligned}\lim_{n\to+\infty}\frac{k,p(n)}{n^k} &= \lim_{n\to+\infty}\frac{a_kn^k + a_{k-1}n^{k-1} + \ldots + a_0}{n^k}\\ &= \lim_{n\to+\infty}\frac{a_kn^k}{n^k} + \lim_{n\to+\infty}\frac{a_{k-1}n^{k-1}}{n^k} + \ldots + \lim_{n\to+\infty}\frac{a_0}{n^k}\\ &= a_k + 0\\ &= a_k\end{aligned}$$

$\therefore k,p(n) \in \Theta(n^k)$

(b) Given $\alpha > \beta > 0$
$\alpha = \beta \times \gamma,\ whereby\ \gamma > 1$
$Using\ limits:$

$\lim_{n\to+\infty}\frac{\beta^n}{\beta^n} = 1$
$\therefore \beta^n \in \Theta(\beta^n)$

$$\begin{aligned}\lim_{n\to+\infty}\frac{\alpha^n}{\beta^n} &= \lim_{n\to+\infty}\frac{(\beta\gamma)^n}{\beta^n}\\ &= \lim_{n\to+\infty}\gamma^n\\ &= \infty\end{aligned}$$

$\therefore \alpha^n \notin \Theta(\beta^n)$

Proving that for $\alpha > \beta > 0$, α^n would have different orders of growth

Furthermore, by using the limit test, little-o classification tells us that if the limit converges to 0 then by definition the bases are different orders of growth.
$\lim_{n\to\infty}\frac{\beta^n}{\alpha^n} = \lim_{n\to\infty}(\frac{\beta}{\alpha})^n$
$since\ \alpha > \beta,\ \frac{\beta}{\alpha} < 1$
$\lim_{n\to\infty}(\frac{\beta}{\alpha})^n = 0$
Showing that the functions are strictly a different order.

Question 2

Answer.

(a) Performing LIFO Operation using a queue

These algorithms are assumed to be in a class, where it has access to an instance of a queue called Q

Algorithm 1: PUSH Performing LIFO Operation using a queue

Input: A queue $Q = \{q_1, q_2, \ldots, q_n\}$ and an input x
Output: A queue $Q = \{q_1, q_2, \ldots, q_n\ x\}$

```
if Q is empty then
    Q.enqueue(x)
else
    while Q is not empty do
        temp ← Q.dequeue
    Q.enqueue(x)
    for each item in temp do
        Q.enque(item)
```

Algorithm 2: POP Performing LIFO Operation using a queue

Input: A queue $Q = \{q_1, q_2, \ldots, q_n\}$
Output: The Last item that has been added to the queue x

```
x ← Q.dequeue
return x
```

(b) Performing FIFO Operation using two stacks

Algorithm 3: ENQUEUE Performing Enqueue with two stacks

Input: Two stacks s1 which is used for pushing and s2 which is used for popping and an item x that is to be added to the queue
Output: x is added to the queue

```
if s1 and s2 are empty then
    s1.push(x)
else
    while s2 is not empty do
        item ← s2.pop()
        s1.push(item)
    s1.push(x)
```

Algorithm 4: DEQUEUE Performing Dequeue with two stacks

Input: Two stacks $s1$ and $s2$ that represents a queue
Output: the first item that was placed into the "queue" that is represented by $s1$ and $s2$

if $s1$ *and* $s2$ *are empty* **then**
  return ϕ
while $s1$ *is not empty* **do**
  $tempItem \leftarrow s1.pop()$
  $s2.push(tempItem)$
$item \leftarrow s2.pop()$
return *item*

Question 3

Answer. Sorting a set with Binary Search Tree

Algorithm 5: INORDERTRAVERSAL exploits the structure of a binary tree and processes left- root-right to produce a sorted result

Input: Tree node *position*, List *result*
Output: void

if $node = \phi$ **then**
  return
INORDERTRAVERSAL$(position.leftChild, result)$
$result.append(position)$
INORDERTRAVERSAL$(position.rightChild, result)$

Algorithm 6: BST SORT Sort a set of values using a Binary Search Trees

Input: BST T, with a root r
Output: A the collection of elements in the tree sorted by the standards of the BST

List result $\leftarrow \phi$
$position \leftarrow r$
$INORDERTRAVERSAL(postion, result)$
return *result*

Algorithm Complexity Analysis

1. **Identify the Input**
   The input of this algorithm is a BST with root r that holds a collection of items
2. **Identify the input size**
   The input size is the number of elements in the tree T, which is n

3. **Identify the elementary Operation**
   The elementary operation is the processing of the node recursively in the InOrderTraversal call

4. **Analysis how How many Times Elementary Operation is Executed**
   The elementary operation is performed for every node in the tree, thus if the size of the tree is n the operation will be performed n times

5. **Asymptotic Analysis**
   Because the operation is performed n times, this algorithm is classified in the class of $\Theta(n)$.

Question 4

Answer. Traversing the graph given a start and end point u, v respectively and find all distinct paths using backtracking.

Algorithm 7: DISTINCT PATHS Traverse a tree and returns the number of distinct paths from starting point u and end point v

---

Input: An undirected graph $G = (V, E)$, a start point u and end point v
Output: Number of distinct paths, n

$distinctPaths \leftarrow []$
$DFS(G, u, v, path \leftarrow [], distinctPaths)$
$n \leftarrow length(distinctPaths)$
return n

Algorithm 8: DFS A recursive algorithm using the Depth-First Search on a tree and keeps track of the number of distinct paths from starting point u and end point v

---

Input: An undirected graph $G = (V, E)$, a start point u and end point v, Current Path of the traversal*path*, Set of Distinct Paths generated *distinctPaths*
Output: Set of Distinct Paths generated *distinctPaths*

```
if u not in path then
    path.append(u)
    if u = v then
        distinctPaths.append(path)
        return
    for each node in u.neighbours do
        if node not in path then
            DFS(G, node, v, path, distinctPaths)
return
```

---

1. **Identify the Input**
   The input of this algorithm is the number of possible paths between the starting point u and ending point v.

2. **Identify the input size**
   The input size is the number of vertices n in the graph excluding vertices u and v, hence, it is $n-2$

3. **Identify the Elementary Operation**
   The elementary operation is the processing of the node recursively in the InOrderTraversal call

4. **Analysis how How many Times Elementary Operation is Executed**
   Assuming the worst case scenario, the graph is a complete graph. Therefore, the number of execution of the elementary operation of this algorithm is a permutation of n-2:
   $$\begin{aligned} P(n, n-2) &= \frac{n!}{(n-(n-2))!} \\ &= \frac{n!}{2!} \\ &= \frac{n!}{2} \end{aligned}$$

5. **Asymptotic Analysis**
   Using Limits:
   $$\begin{aligned} \lim_{x \to +\infty} \frac{n!/2}{n!} &= \lim_{x \to +\infty} \frac{1}{2} \\ &= \frac{1}{2} \end{aligned}$$
   $$\therefore \frac{n!}{2} \in \Theta(n!)$$