Name: Kunj Patel____ UnityID: kmpatel7______ Date: 02/10/17_____

Name: Hardik Aneja____ UnityID: haneja_______ Date: 02/10/17_____

CSC316: Homework 2

Submission: Submit your completed assignment to Gradescope.

Gradescope, which we will use for grading, requires all submissions to conform to the same format. Therefore, to facilitate grading, you must either:

OPTION 1: Type your answers into this .docx document in the spaces provided

- use an equation editor feature for clarity when typing complex math expressions
- use the shape tools to draw any data structures, or draw your data structures in a graphics program and insert the resulting image into this document

OPTION 2: Print this document, hand-write your solutions in the spaces provided, then scan the printed pages. If you provide hand-written solutions, they must be clearly legible, or you will not receive full credit.

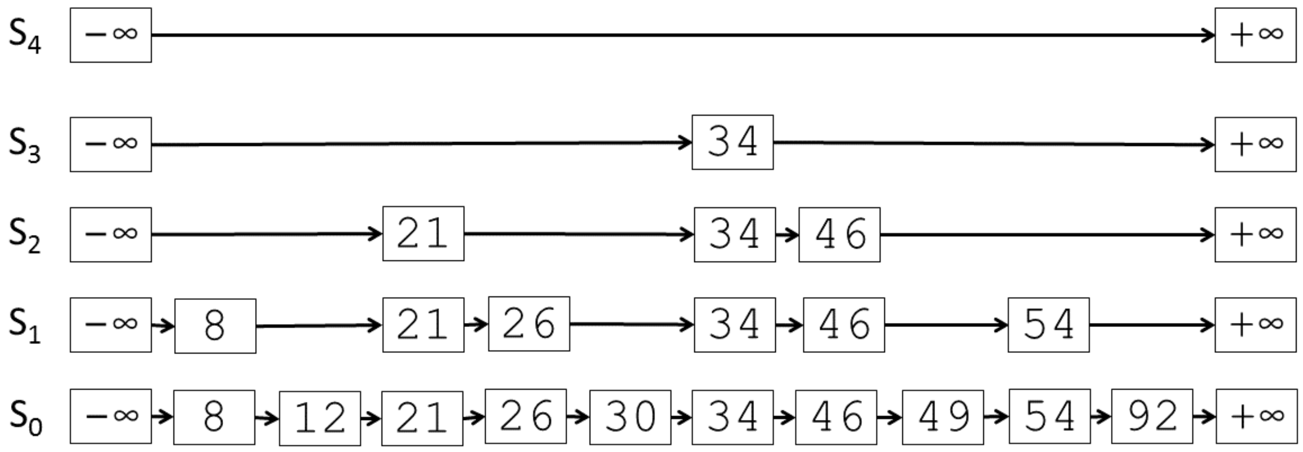
Collaboration Rules:

- You are allowed to form pairs of 2 students to discuss and collaborate on homework problems.
- At the beginning of each problem, you must clearly document
  - the names and email addresses (if available) of all people you discussed the problem with, whether they are current CSC316 students or not.
  - any websites from which you visited to better understand the concepts. You must also briefly describe how the website helped you devise your solutions. If a website provides a solution, say so; however, also explain how the website helped you better understand the concepts.
- When in doubt, ask the instructor!

Problem	Points Earned	Points Possible
Skip List		30
Recursive Multiplication		10
Analyzing Recursive Multiplication		10
Peculiar Algorithm Analysis		25
Climbing a Staircase		10
Data Structure Selection		15
TOTAL		100

Skip Lists

Answer each question based on the following skip list:



(1) Suppose you lookUp the value “30” in the skip list.

a. List the keys of the nodes that “30” is compared to at each level (in order).

S_4: +∞

S_3: 34

S_2: 21, 34

S_1: 26, 34

S_0: 30

b. State the average running time of the lookUp operation in terms of Big-Oh: **O(log(n))**

(2) Suppose the value “52” is inserted into the skip list. A random number r = 0.976 is generated.

a. Assuming the skip list contains “quad nodes” as described in lecture, list the nodes whose “next” fields are updated during the insert operation.

S_4: -∞

S_3: 34

S_2: 46

S_1: 46

S_0: 49

b. After the insert operation, how many total S_i lists are present in the skip list?

S_5 should be added, increasing the number of lists from 5 to 6.

c. State the average running time of the insert operation in terms of Big-Oh: **O(log(n))**

(3) Suppose the value “34” is removed from the original skip list above (assume “52” was not inserted). A random number r = 0.319 is generated.

a. Assuming the skip list contains “quad nodes” as described in lecture, list the nodes whose “next” fields are updated during the remove operation.

S_4:

S_3: -∞

S_2: 21

S_1: 26

S_0: 30

b. After the remove operation, how many total S_i lists are present?

S_4 should be removed, decreasing the number of lists from 5 to 4.

c. State the average running time of the remove operation in terms of Big-Oh: **O(log(n))**

(4) Who first described (or “invented”) skip lists? In what year? Which static analysis tool did the creator of the skip list also co-create?

a. Name: **William Pugh**

b. Year: **1989(courses.cs.vt.edu, Wikipedia)**

c. Static Analysis tool: **FindBugs**

(5) How does the average-case running time of lookUp using a skip list compare to the worst-case running time of binary search using a sorted array-based list?

The average case running time of lookUp for a skip list is O(log(n)) while the worst case running time of binary search is O(log(n)). This makes sense because skip list tries to approximate the performance of binary search with the advantages of a linked list. Skip lists trade memory space for speed of lookUp.

Recursive Multiplication.

Complete the following algorithm `multiply` (and its associated helper algorithm) that returns the sum of the elements in a stack multiplied by the sum of the elements in a queue. However, <u>you cannot use the multiplication operator (*) in your algorithm</u>!!!

```
Algorithm multiply(S, Q)
Input     a stack S with n elements
          a queue Q with n elements
Output    sum of elements in S multiplied by sum of the elements in Q
1.   sSum ← 0
2.   qSum ← 0

3.   for i←1 to n do
4.        sSum += S pop element
5.   for i←1 to n do
6.        qSum += Q dequeue element

7.   if qSum = 0 or sSum = 0 then
8.        return 0
9.   else if qSum < 0 and sSum < 0 then
10.       return multiplyHelper(-sSum, -qSum)
11.  else if qSum < 0 then
12.       return -multiplyHelper(sSum, -qSum)
13.  else if sSum < 0 then
14.       return -multiplyHelper(-sSum, qSum)
15.  else
16.       return multiplyHelper(sSum, qSum);
```

```
Algorithm multiplyHelper(value1, value2)

Input value1 and value2 to multiply together.

Output value1 and value2 multiplied together.

    1. if value2 = 0
    2.   return 0
    3. Else
    4.   return value1 + multiplyHelper(value1, value2-1)
```

Analyze Recursive Multiplication.

Analyze your recursive multiplication algorithm. Show all work. Remember to follow the steps described in lecture:

State your function T(n) of the number of operations performed:

$$T_{total}(n) = T_{multiply}(n) + T_{multiplyHelper}(n)$$

$T_{multiply}(n) =$

Line	# Simple Operations	Description of Operation
1.	1	1. Variable assignment (sSum = 0)
2.	1	2. Variable assignment (qSum = 0)
3.	1	3. Variable assignment (i = 1)
	n+1	Comparison of i < n
	n	Adding i+1 at end of iteration
	n	Variable assignment (i = i + 1)
4.	n	4. Stack pop
	n	Adding (sSum + element)
	n	Variable assignment (sSum = sSum + …)
5.	1	5. Variable assignment (i = 1)
	n+1	Comparison of i < n
	n	Adding i+1 at end of iteration
	n	Variable assignment (i = i + 1)
6.	n	6. Queue dequeue
	n	Adding (qSum + element)
	n	Variable assignment (qSum = qSum + …)
7.	1	7. Comparison (qSum = 0)
	1	Comparison (sSum = 0)
8.	1	8. return (0)
9.	1	9. Comparison (qSum < 0)
	1	Comparison (sSum < 0)
10.	1	10.return multiplyHelper(-sSum, -qSum)
11.	1	11.Comparison (qSum < 0)
12.	1	12.return -multiplyHelper(sSum, -qSum)
13.	1	13.Comparison (sSum < 0)
14.	1	14.return -multiplyHelper(-sSum, qSum)
15.	1	15.step into else
16.	1	16.return multiplyHelper(sSum, qSum)
T(n)	12n + 18	

$T_{\text{multiplyHelper}}(m)$ = **m equals value2, and is used to differentiate from n according to the piazza post** @118

$$T(n) = \begin{cases} g(n) & if\ n = d \\ aT(b) + f(n) & if\ n > d \end{cases}$$

$$T(m) = \begin{Bmatrix} c & if\ m = 1 \\ T(m-1) + c & if\ m > 1 \end{Bmatrix}$$

- Show your work for "unfolding" the function T(n)

m	T(m)		Pattern?
1	T(1)	= c	
2	T(2)	= T(1) + c = c + c	
3	T(3)	= T(2) + c = c + c + c	T(m) =mc

- State the running time T(n) in terms of Big-Oh.

$T_{\text{multiplyHelper}}(m) = mc$
$T_{\text{multiply}}(n) = 12n + 18$

Definition: $f(n) \leq c \cdot g(n)$, **f(n) = 12n + 18, g(n) = n**

1. Choose $n_0 = 1$
2. Assume n > 1
3. Find c:

   f(n)/g(n) = (12n + 18) / n = (12n + 18n) / n = 30

4. Choose c = 30
   Therefore, 12n + 18 < 30n when n > 1
   $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$ is true when c = 30 for $n \geq n_0 = 1$.
   f(n) is O(n)

Definition: $f(m) \leq c \cdot g(m)$, **f(m) = mc, g(n) = m**

5. Choose $m_0 = 1$
6. Assume m > 1
7. Find c:

   f(m)/g(m) = (mc) / m = c

8. C is constant therefore,
   $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$ is true.
   f(m) is O(m)

$T_{total}(n) = T_{multiply}(n) + T_{multiplyHelper}(n)$ **= T(n) + T(m)**

Peculiar Algorithm Analysis.

Analyze the peculiar algorithm below to determine the Big-Oh running time efficiency. The algorithm does nothing useful! Do not try to understand what the algorithm does – just analyze it. Assume the algorithm is written correctly. Show all work. Remember to:

- State your function T(n) of the number of operations performed
- Show your work for "unfolding" the function T(n)
- State the running time T(n) in terms of Big-Oh.

```
Algorithm peculiar(T, min, max)
Input an array T of n elements
Output ????????

if max <= min then
    temp ← T[min] * 2
    if temp > (T[max] / 2) then
        T[min] ← T[max] * 2
else
    q ← ( max - min ) / 4
    peculiar( T, min, min + q )
    peculiar( T, min + q + 1, min + q*2 )
    peculiar( T, min + q*2 + 1, min + q*3 )
    peculiar( T, min + q*3 + 1, max )
    for int i ← min to max do
        temp ← T[i]
        T[i] ← T[0]
        T[0] ← temp
```

State your function T(n) of the number of operations performed:

$$T(n) = \begin{cases} C, & n = 1 \\ 4T(n/4) + n, & n > 1 \end{cases}$$

Climbing a Staircase.

Suppose Sally is climbing the staircase to get to Jack's house. However, Sally's legs are only long enough to reach one step *or* two steps on the staircase. Find the number of different ways $W(n)$ to climb *n* steps to get to Jack's front door.

For example:

$W(1) = 1$ since a section of staircase with n=1 step could be climbed only one way:

- 1 step

$W(2) = 2$ since a section of staircase with n=2 steps could be climbed two different ways:

- 1 step, 1 step
- 2 steps

$W(3) = 3$ since a section of staircase with n=3 steps could be climbed three different ways:

- 1 step, 1 step, 1 step
- 1 step, 2 steps
- 2 steps, 1 step

…

Write a function $W(n)$ that uses recursion to express the number of different ways to climb *n* steps to get to Jack's front door. *You do not need to design an algorithm, just state a recursive expression for computing the number of different ways to climb the steps.* Show all work!

$$W(n) = \begin{cases} \mathcal{C}, & n = 1 \\ 2T(n-1), & n > 1 \end{cases}$$

```
Algorithm T

Input: Number of steps:n + 1

Output: Number of ways to climb the stairs

if  n <= 1  then
    return n;
else
    return T(n-1) + T(n-2)
```

Data Structure Selection.

Describe the data structure you would use to solve the following problem. Be sure to:

- **describe *all* necessary aspects of the data structure** to give a complete overview of running time efficiency *(for example: don't just say you are using a "linked list" if you are really using an "unsorted circular linked list with head and tail pointers")*
- **list *all* data structure operations** required to solve the problem *(for example: insert(), lookup())*
- **describe the estimated running time** for each of the operations using your selected data structure
- briefly **justify *why* you selected the chosen data structure and *why* it provides an efficient solution**

Problem Description

A local public library has requested that you build a program to help patrons electronically browse the library stacks. Patrons should be able to enter a single word describing the subject matter of the book, and the program should then return the set of books associated with the entered subject term. Patrons should then be able to browse through the set of books associated with the subject term so that they can find one to checkout.

Assumptions:

Given: Book objects.

We are storing book objects in our data structure and are assuming the library has only one book of each and continuously adds books to the list.

Question	**Response**
Which data structure will you use?	We will use a linked List.
Describe *all* necessary aspects of the data structure.	We will use a linked List with head and tail pointers for efficient insertion of the list items. We will also be using a linked List iterator for efficient searching of the subject Matter.
List the data structure operations required, and	Insert()- O(1): Because we are using the tail pointer, insertion at the end will take one time step

provide estimated running time (in terms of Big-Oh) for each	Search()- O(n)- It has to go through every item in the list in order to find all the books with the matching subject matter. It also uses a move to front heuristic to make search efficient in the long run.
Justify your data structure selection and why it provides an efficient solution to the problem	Linked List provides a really efficient way to insert items and searching is made easy by the list iterator and the move to front heuristic. If we were to use a arrayList, the size would be constrained and if we reach size growArray method would have to copy all the items to new array, slowing the run time down by O(n). Move to front heuristic will be efficient implemented as a linked list because the head node is maintained leading to O(1) insertion while an arrayList would be O(n).