Chris Grimes
CS 46101
Homework # 1
Problem 1.

a) Show that if f(n) is O(g(n)) and d(n) is O(h(n)), then f(n) × d(n) is O(g(n) × h(n))

if f(n)=O(g(n)) and we assign the value of O(n), then both f(n) and O(g(n))= O(n)

if d(n)=O(h(n)) and we assign the value of O(1), then both d(n) and O(h(n))=O(1)

then, f(n) x d(n) is equivalent to O(n) x O(1) and O(g(n)) x O(h(n)) is equivalent to O(n) x O(1)

b) Show that 3(n + 1)^5 + 2n^3 log(n) is O(n^5).

3(n+1)^5=3n^5+ 15n^4+30n^3+30n^2+15n+3

add to it 2n^3log(n), and we get

3n^5+ 15n^4+30n^3+30n^2+15n+3+2n^3 log(n) and the leading term is 3n^5

since, 3 is a constant multiple we can drop it leaving us with n^5

so, 3(n + 1)^5 + 2n^3 log(n) is O(n^5)

c) Algorithm A executes 10n^2 log(n) operations, while algorithm B executes n^3 operations. Determine the minimum integer value n_0 such that A executes fewer operations than B for n ≥ n0.

if n_0=11 then algorithm B will execute more instruction than algorithm A

for values n_0<10 algorithm A executes more instructions

if n_0=9, 10n^2log(n)=772.936432646 and n^3=729

for n_0=10 both algorithms execute 1000 instructions

for values n_0>10 algorithm B executes more instructions

if n_0=11, 10n^2log(n)=1260.08514904 and n^3=1331

Chris Grimes
CS 46101
Homework # 1
Problem 2.

a) What does the following algorithm do? Analyze its worst-case running time, and express it using “Big-Oh” notation.

Algorithm Foo (a, n):

Input: two integers, a and n

Output: ?

```
k ← 0                    //worst case scenario this runs once
b ← 1                    //worst case scenario this runs once
while k < n^2 do         //worst case scenario this loop runs n^2 times
        k ← k + 1        //worst case scenario this runs n^2 times
        b ← b ∗ a        //worst case scenario this runs n^2 times
return b                 //worst case scenario this runs once --> will return b*(a^n)
```

Because worst case scenario the **while** loop has to run n^2 times this algorithm has a “Big-Oh” notation of O(n^2).

Chris Grimes
CS 46101
Homework # 1

b) What does the following algorithm do? Analyze its worst-case running time, and express it using “Big-Oh” notation.

Algorithm Bar (a, n):

Input: two integers, a and n

Output: ?

```
k ← n^2                     //worst case scenario this runs once
b ← 1                       //worst case scenario this runs once
c ← a                       //worst case scenario this runs once
while k > 0 do              //worst case scenario this loop runs n+ a constant times
    if k mod 2 = 0 then     //then if k is even exe, could run multiple times in succession
        k ← k/2
        c ← c ∗ c           //every time c is squared
    else                    //if k is odd exe, will not run twice in a row
        k ← k − 1
        b ← b ∗ c           //b=b(c^x) where x is the amount of times c was squared above
return b                    //worst case scenario this runs once
```

Because worst case scenario k alternates between the **if** check, being cut in half, and the **else** check, being decremented by one, meaning that the **while** loop will run n+ a constant times depending on the staring value on n this algorithm has a “Big-Oh” notation of O(n).

Chris Grimes
CS 46101
Homework # 1
Problem 3.

a) Describe the output of the following series of stack operations on a single, initially empty stack:

	//output: none
push(5),	//output: none
push(3),	//output: none
pop(),	//output: 3
push(2),	//output: none
push(8),	//output: none
pop(),	//output: 8
push(9),	//output: none
push(1),	//output: none
pop(),	//output: 1
push(7),	//output: none
push(6),	//output: none
pop(),	//output: 6
pop(),	//output:7
push(4),	//output: none
pop(),	//output: 4
pop().	//output: 9

output is as follows: 3 8 1 6 7 4 9

Chris Grimes
CS 46101
Homework # 1

b) Describe the output of the following series of queue operations on a single, initially empty queue:

	//output: none
enqueue(5),	//output: none
enqueue(3),	//output: none
dequeue(),	//output:5
enqueue(2),	//output: none
enqueue(8),	//output: none
dequeue(),	//output:3
enqueue(9),	//output: none
enqueue(1),	//output: none
dequeue(),	//output:2
enqueue(7),	//output: none
enqueue(6),	//output: none
dequeue(),	//output:8
dequeue(),	//output:9
enqueue(4),	//output: none
dequeue(),	//output:1
dequeue()	//output:7

output as follows: 5 3 2 8 9 1 7

Chris Grimes
CS 46101
Homework # 1

c) Describe in pseudo-code a linear-time algorithm for creating a copy stack S' of a stack S. As the result, you must end up with two identical stacks S' and S. To access the stack, you are only allowed to use the methods of stack ADT.

Assuming that each instance of a stack has one pointer that always points to it's topOfStack

Algorithm copyStack (stack S):

Input: one stack instance S

Output: a copy of stack S named S'

*topOfStackPtr← the pointer to S's top of stack

* bottomOfStackPtr ← nullPtr

while topOfStackPtr != 0

if S' top of stack=0

S' top of stack points at a new instance of the type that S is a stack, of created with the same data // if the original S is a stack of ints S' top of stack will point at a new int with the same value

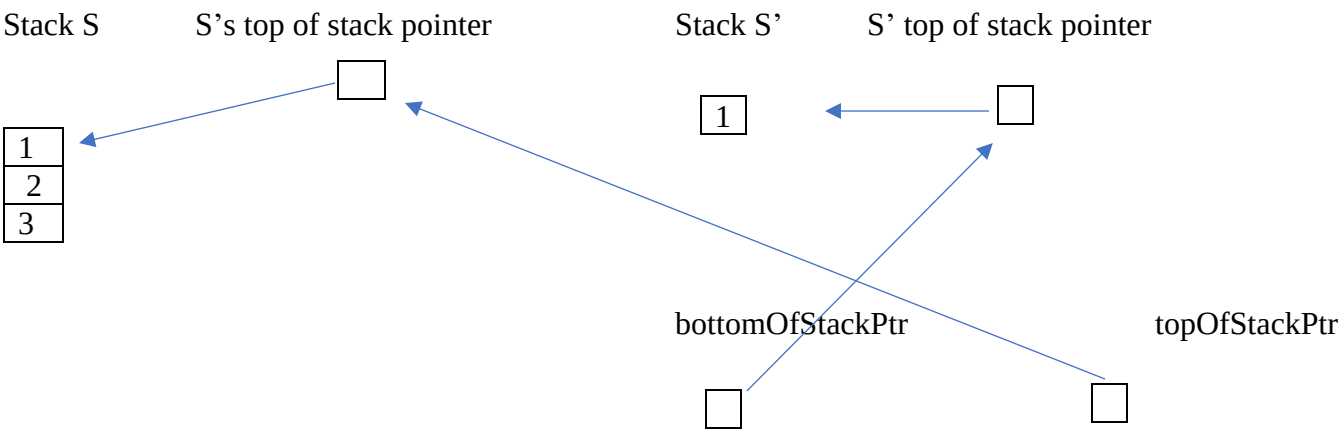
bottomOfStackPtr← S' top of stack

else

bottomOfStack's next← a new instance of the type that topOfStackPtr points at with the same value

bottomOfStack ← bottomOfStack's next

topOfStackPtr← topOfStackPtr's next



Chris Grimes
CS 46101
Homework # 1
d) Describe how to implement two stacks using one array. The total number of elements in both stacks is limited by the array length; all stack operations should run in O(1) time

Declare an array, **Stacks**, of size n where n can be any integer, two pointers one for each stack 1 &2 to maintain topOfStack for each substack. Have stack 1 start from **Stack**[0] and have stack 2 start from **Stack**[n-1]. Each time a stack needs to grow move the pointer and put the new item in the next position, that is if stack 1 needs to push increment the pointer and store the relevant data in that position, if stack 2 needs to push decrement its pointer and store the relevant data in that position. To pop to relevant pointer can return the data stored in said position and change where it's pointed to, that is if stack 1 needs to pop its pointer returns to data stored at its current position and the pointer would get decremented, for stack 2 the pointer returns the data stored it the current position and then increments.

Chris Grimes
CS 46101
Homework # 1

Problem 4.

a) The following are parts of their original implementation of a queue using two stacks (in stack and out stack). Analyze the worst-case running times of its enqueue and dequeue methods and express them using “Big-Oh” notation.

```
Algorithm enqueue(o)
    in stack.push(o)                    //worst case scenario this line executes 1 time
Algorithm dequeue()
    while (! in stack.isEmpty()) do     //worst case scenario this lines runs n times
        out stack.push(in stack.pop())
    if (out stack.isEmpty()) then       //worst case scenario this line runs 1 time
        throw a QueueEmptyException
    return obj ← out stack.pop()
    while (! out stack.isEmpty()) do    //worst case scenario this lines runs n times
        in stack.push(out stack.pop())
    return return obj;
```

The “Big-Oh” notation for enqueue(o) is O(1) and the “Big-Oh” notation for dequeue() is O(2n).

Chris Grimes
CS 46101
Homework # 1

b) Sometime in the early twenty-first century a war erupted between the humans and the machines, which humans lost. 120 years after its creation, the hovercraft Nebuchadnezzar ended up in the hands of the human resistance leader and hacker extraordinaire, Morpheus. Always on the run, the rebels needed much faster software to escape the machines, so Morpheus and his crew set out to optimize Neb's code. Thus a new implementation of a queue (still using two stacks) was born:

```
Algorithm enqueue(o)                    //per operation
    in stack.push(o)                    //worst case scenario with 2n operations this line runs 2n times
Algorithm dequeue()                     //per operation
    if (out stack.isEmpty()) then
        while (! in stack.isEmpty()) do  //worst case scenario with n operations this line runs n^2 times
        out stack.push(in stack.pop())
    if (out stack.isEmpty()) then                //worst case scenario this line runs once
        throw a QueueEmptyException
    return out stack.pop()
```

What is the worst-case complexity of performing a series of 2n enqueue and n dequeue operations in an unspecified order? Express this using "Big-Oh" notation.

Using "Big-Oh" notation for enqueue(o) with 2n operations you get O(2n), and using "Big-Oh" notation for dequeue() with n operations you get O(n^2)

Chris Grimes
CS 46101
Homework # 1

Problem 5. A program Thunk written by a graduate student uses an implementation of the sequence ADT as its main component. It performs atRank, insertAtRank and remove operations in some unspecified order. It is known that Thunk performs (n^2)/4 atRank operations, n insertAtRank operations, and n^2 remove operations. Which implementation of the sequence ADT should the student use in the interest of efficiency: the array-based one or the one that uses a doubly-linked list? Explain

With (n^2)/4 atRank operations, n insertAtRank operations, and n^2 remove operations in the interest of efficiency we should implement the ADT based on the doubly-linked list as this has the least costly remove operations and the remove operation is the most used.

	Array(each operation)	List(each operation)
(n^2)/4 atRank operations:	O(1)	O(n)
n insertAtRank operations:	O(n)	O(n)
n^2 remove operations.:	O(n)	O(1)

	Array(all operations)	List(all operations)
(n^2)/4 atRank operations:	O((n^2)/4)	O((n^3)/4)
n insertAtRank operations:	O(n^2)	O(n^2)
n^2 remove operations.:	O(n^3)	O(n^2)