Homework 4

1. **Using Big Oh notation, indicate the time requirement of each of the following tasks in the worst case. Describe any assumptions that you make.**

   (a) **After arriving at a party, you shake hands with each person there.**

   Suppose that there are n people to shake hands with, and suppose that shaking one person's hand is independent of how many people are in the party. Since shaking one person's hand is independent on how many people are in the party, it would take some constant time, c, to shake one person's hand. We have to shake hands with n people, and it takes c time to shake one person's hand, so it takes $c \cdot n$ time. $c \cdot n \leq c \cdot n$, for $n \geq 1$. Since $c \cdot n \leq c \cdot n$, $c \cdot n = O(n)$. The time complexity is $O(n)$.

   (b) **Each person in a room shakes hands with everyone else in the room.**

   Suppose there are n people in the room. That means each person has to shake hands with $n-1$ people. Like above, suppose that shaking hands with one person takes some constant time, c, to shake one person's hand. Each person has to shake hands with $n-1$ people, and it takes c time to shake one person's hand, so it takes $c \cdot (n-1)$ time for one person to shake everybody else's hand. Since everybody has to shake everyone else's hand, we have to do this n times. That means the time for this is $c \cdot (n-1) \cdot n = (n-1) \cdot n = n^2 - n$. $n^2 - n \leq n^2$, for $n \geq 1$. Since $n^2 - n \leq n^2$, $n^2 - n = O(n^2)$. The time complexity is $O(n^2)$.

   (c) **You climb a flight of stairs.**

   Suppose there are n steps on the flight of stairs, and suppose that climbing one step is independent of how many steps we have to climb. Since climbing one step is independent of how many steps I have to climb, it would take some constant time, c, to climb one step. I have to climb n steps, so it would take $c \cdot n$ time to climb a flight of stairs. $c \cdot n \leq c \cdot n$, for $n \geq 1$. Since $c \cdot n \leq c \cdot n$, $c \cdot n = O(n)$. The time complexity is $O(n)$.

   (d) **You slide down the banister.**

   Suppose that I slide down the banister at constant speed. Sliding down a banister of height n takes $c \cdot n$, for some constant c. $c \cdot n \leq c \cdot n$, for $n \geq 1$. Since $c \cdot n \leq c \cdot n$, $c \cdot n = O(n)$.

   (e) **After entering an elevator, you press a button to choose a floor.**

   Suppose there are n floors available to choose from on the elevator, and suppose that the time it takes to press one button to choose a floor is independent of how many floors there are available to choose from. Since the time it takes to press one button to choose a floor is independent of how many floors there are available to choose from, it takes some constant time, c. $c \leq c$. Since $c \leq c$, $c = O(1)$. The time complexity is $O(1)$.

   (f) **You ride the elevator from the ground floor up to the n^{th} floor.**

   Suppose that the time it takes the elevator to go from one floor to the next is independent of how many floors there are in that building. Since taking the elevator from one floor to the next is independent on how many floors there are in the building, it takes some constant time, c, for the elevator to go from one floor to the next. I have to ride the elevator from the ground floor up to the n^{th} floor, so that is n floors. Combining the information above, the total time to ride the elevator from the ground floor up the n^{th} floor is $c \cdot n$. $c \cdot n \leq c \cdot n$, for $n \geq 1$. Since $c \cdot n \leq c \cdot n$, $c \cdot n = O(n)$.

(g) **You read a book twice.**

Suppose there are n pages in the book, and suppose reading one page of the book is independent of how many pages are in that book. Since reading one page of the book is independent of how many pages are in that book, it takes some constant time, c, to read one page of the book. From the information above, the total time is $c \cdot n$ to read one book. $c \cdot n \leq c \cdot n$, for $n \geq 1$. Since $c \cdot n \leq c \cdot n$, $c \cdot n = O(n)$. The time complexity to read a book once is $O(n)$. I have to read the book twice, so the time complexity is $O(n) + O(n) = O(n)$.

3. **Using Big Oh notation, indicate the time requirement of each of the following tasks in the worst case.**

(a) **Display all the integers in an array of integers.**

Refer to Homework4Driver.java

Suppose there are n integers in the array, and the integers are contiguous(no gap with null). Displaying one integer from the array is independent of the number of integers in the array, so it takes some constant time, c, to display one integer from the array. Since there are n integers, the time complexity to display all integers is $c \cdot n = O(n)$.

The running time of my algorithm is $4n + 3$ because there are 3 operations, then there is a loop from the first integer to the last integer of the array with 4 operations per loop. $4n + 3 \leq 7n$, for $n \geq 1$. Since $4n + 3 \leq 7n$, $4n + 3 = O(n)$ The efficiency is $O(n)$

(b) **Display all the integers in a chain of linked nodes.**

Refer to Homework4Driver.java

Suppose there are n integers in the linked list, and the integers are to be displayed sequentially. Displaying one integer from the linked list would be independent on the number of nodes in the linked list because we are displaying the integers sequentially. Therefore, there will always be a currentNode reference to the node previous to the node we want to display. It will only take a constant number of operations to reach the node we want displayed. Thus, the time complexity is $c \cdot n = O(n)$.

The running time of my algorithm is $3n + 2$ because there are 2 operation, then there is a loop from the first node to the last node, with 3 operations each iteration. $3n + 2 \leq 5n$, for $n \geq 1$. Since $3n + 2 \leq 5n$, $3n + 2 = O(n)$. The efficiency is $O(n)$.

(c) **Display the n^{th} integer in an array of integers.**

Refer to Homework4Driver.java

From above, displaying one integer from the array is independent of the number of integers in the array, so it takes some constant time, c, to display one integer from the array. Random access to an element of the array will only require some constant time, so the time complexity to display the n^{th} integer is $c = O(1)$

The running time of my algorithm is 5 because there are 5 operations. $5 \leq 5 \cdot 1$. Since $5 \leq 5 \cdot 1$, $5 = O(1)$. The efficiency is $O(1)$

(d) **Compute the sum of the first n even integers in an array of integers.**

Refer to Homework4Driver.java

The worst case would have all integers in the array be even, and n be the same as the number of integers in the array. My algorithm would first make 2 operations, then go into a loop from the first integer to the last integer. Within that loop, there would be some constant number of operations that are independent of how many integers are in the array. There are n integers, so the running time is $7n + 5$ because there are 5 operations, then there is a loop from the first integer to the n^{th} integer with 7 operations each iteration. $7n + 5 \leq 12n$, for $n \geq 1$. Since $7n + 5 \leq 12n$, $7n + 5 = O(n)$. The efficiency is $O(n)$.

4. **Chapter 5 describes an implementation of the ADT list that uses an array that can expand dynamically. Using Big Oh notation, derive the time complexity of the method `doubleArray`, as given in Segment 5.18.**

```
/** Task: Doubles the size of the array of list entries. */
private void doubleArray()
{
    T[] oldList = list; // save reference to array of list entries
    int oldSize = oldList.length; // save old max size of array

    list = (T[]) new Object[2 * oldSize]; // double size of array

    // copy entries from old array to new, bigger array
    for (int index = 0; index < oldSize; index++)
        list[index] = oldList[index];
} // end doubleArray
```

Suppose oldList is a size n array. The algorithm begins with 8 operations, then goes into a loop from the first element in oldList to the last element with 3 operations each iteration. The running time is $3n + 8$. $3n + 8 \leq 11n$, for $n \geq 1$. Since $3n + 8 \leq 11n$, $3n + 8 = O(n)$. The efficiency is $O(n)$.

5. **Suppose that you alter the linked implementation of the ADT list to include a tail reference, as described in Segments 7.21 through 7.24 of Chapter 7. The time efficiencies of what methods, if any, are affected by this change? Use Big Oh notation to describe the efficiencies of the affected methods.**

Refer to LList.java

Method	**Original Efficiency**	**New Efficiency**
boolean add(T newEntry)	$O(n)$	$O(1)$
boolean add(int newPosition, T newEntry)	$O(1)$ to $O(n)$	no change
T remove(int givenPosition)	$O(1)$ to $O(n)$	no change
void clear()	$O(1)$	no change
boolean replace(int givenPosition, T newEntry)	$O(1)$ to $O(n)$	no change
T getEntry(int givenPosition)	$O(1)$ to $O(n)$	no change
boolean contains(T anEntry)	$O(1)$ to $O(n)$	no change
int getLength()	$O(1)$	no change
boolean isEmpty()	$O(1)$	no change
boolean isFull()	$O(1)$	no change
void display()	$O(n)$	no change

Only the `boolean add(T newEntry)` method changed efficiency. Without a tail reference to the last node of the linked list, add would have to iterate from the head to the tail which took $O(n)$ time, since add would have to iterate through n nodes. With a tail reference, add could directly use the reference to the last node and append a new node instead of iterating from the head to the tail.

The other methods did not change efficiency because they either had the option to alter or view the $(n-1)^{st}$ node, or was independent of the number of nodes in the linked list.

7. **In the worst case, Algorithm X requires $n^2 + 9n + 5$ operations and Algorithm Y requires $5n^2$ operations. What is the Big Oh of each algorithm?**

Algorithm X: $n^2 + 9n + 5 \leq 15n^2$, for $n \geq 1$. Since $n^2 + 9n + 5 \leq 15n^2$, $n^2 + 9n + 5 = O(n^2)$.

Algorithm Y: $5n^2 \leq 5n^2$, for $n \geq 1$. Since $5n^2 \leq 5n^2$, $5n^2 = O(n^2)$.

10. **Suppose that your implementation of a particular algorithm appears in Java as follows:**

```
for(int pass = 1; pass <= n; pass++)
{
    for(int index = 0; index < n; index++)
    {
        for(int count = 1; count < 10; count++)
        {
            ...
        } // end for
    } // end for
} // end for
```

The algorithm involves an array of n items. The previous code shows the only repetition in the algorithm, but it does not show the computations that occur within the loops. These computations, however, are independent of n. What is the order of the algorithm?

There are 6 operations independent of loops. The most outer loop iterates from 1 to n with 3 operations each iteration. The most outer loop has n iterations, so the running time for that loop is $3n$. The second most outer loop iterates from 0 to n-1 with 3 operations each iteration. The second most outer loop has n iterations, so the running time for that loop is $3n$. The most inner loop iterates from 1 to 9 with 2 + c operations, for some constant c. The most inner loop has 9 iterations, so the running time for that loop is $9(2 + c) = 18 + 9c$. The total running time is $3n \cdot 3n \cdot (18 + 9c) + 6 = 9(18 + 9c)n^2 + 6 = 9Cn^2 + 6$, for another constant C. $9Cn^2 + 6 \leq (9C + 6)n^2$, for $n \geq 1$. Since $9C + 6$ is a constant, and $9Cn^2 + 6 \leq (9C + 6)n^2$, $9Cn^2 + 6 = O(n^2)$. The efficiency is $O(n^2)$.

11. **Repeat Exercise 10, but replace 10 with n in the inner loop.**

From above, there are 6 operations independent of loops, the running time for the most outer loop is $3n$, and the running time for the second most outer loop is also $3n$. If we replace 10 with n in the inner loop, the loop will iterate from 1 to n with 2 + c operations, for some constant c. The most inner loop has n iterations, so the running time for that loop is $(2 + c)n = c_1 n$, for some constant c_1. The total running time is $3n \cdot 3n \cdot c_1 n + 6 = 9c_1 n^3 + 6$. $9c_1 n^3 + 6 \leq (9c_1 + 6)n^3$, for $n \geq 1$. Since $9c_1 + 6$ is a constant, and $9c_1 n^3 + 6 \leq (9c_1 + 6)n^3$, $9c_1 n^3 + 6 = O(n^3)$. The efficiency is $O(n^3)$.

16. **Segment 9.15 and the chapter summary showed the relationships among typical growth-rate functions. Indicate where the following growth-rate functions belong in the ordering:**

(a) $n^2 \log n$

$\log n \geq 1$ for $n \geq 2$ That means $n^2 \log n \geq n^2$ for $n \geq 2$. $\log n \leq n$ for $n \geq 1$. That means $n^2 \log n \leq n^3$ for $n \geq 1$. Thus, $n^2 \log n$ resides between n^2 and n^3.

(b) $\sqrt{n}$

$\sqrt{n} = n^{\frac{1}{2}}$. That means $n^{\frac{1}{2}} \leq n$ for $n \geq 1$. $n^{\frac{1}{2}} \geq \log n$ for $n \geq 2$. $n^{\frac{1}{2}} \leq \log^2 n$ for $n \geq 3$. That means $n^{\frac{1}{2}}$ is between $\log n$ and $\log^2 n \equiv \sqrt{n}$ is between $\log n$ and $\log^2 n$.

18. **Suppose that you have dictionary whose words are not sorted in alphabetical order. As a function of the number, n, of words, what is the time complexity of searching for a particular word in this dictionary?**

Since the dictionary is not sorted, we will have to search through the dictionary one word a time to make sure we don't skip the we are looking for. I am assuming that we are looking through the dictionary in the order of the first word to the last word. The best case complexity, the searched word is the first word in the dictionary. The average case complexity, the searched word is in the middle of the dictionary. The worst case complexity, the searched word is the last word in the dictionary. It seems like the average case complexity and the worst case complexity will result in the same Big Oh upper bound, and the best case complexity is very trivial. Thus, I will only focus on the worst case complexity. We must look through the entire dictionary of words, so we must look at n words. That means the time complexity for searching a particular word in this dictionary is $O(n)$.