CS526 O2
Homework Assignment 2

Problem 1 (20 points). This is practice for analyzing running time of algorithms. Express the running time of the following methods, which are written in pseudocode, using the *big-oh* notation. Assume that all variables are appropriately declared. You must justify your answers. If you show only answers, you will not get any credit even if they are correct.

(1)

```
method1(int[ ] a)  // returns integer
 x = 0; // O(1) It does not depend on array length
 y = 0;
 for (i=1; i<n; i++) { // n is the number of elements in array a // O(n) because the loop will run n times
        if (a[i] == a[i-1]) { // O(1) It does not depend on array length
              x = x + 1; // O(1) It does not depend on array length
        }
        else {
              y = y + 1; // O(1) It does not depend on array length
        }
}
 return (x - y); // O(1) It does not depend on array length
```

→ For this method we only care the fastest growing term → Overall the running time of this method is **O(n)**

(2)

```
method2(int[ ] a, int[ ] b)  // assume equal-length arrays
      x = 0; // O(1) It does not depend on array length
      i = 0; // O(1) It does not depend on array length
while (i < n) { // n is the number of elements in each array // O(n) because the loop will run n  times
            y = 0; // O(1) It does not depend on array length

            j = 0; // O(1) It does not depend on array length

        while (j < n) { // O(n^2) because each time outer loop runs → this loop will run n  time. So, when outer loop run n time → this loop will run n*n = n^2 times → Big O(n^2)
              k = 0
              while (k <= j) {  // O(n^3) because this loop is run 1 + 2+3+4+5 …n time when j run from 0 to n -1 time respectively ( when j = 0, loop run 1 time, j = 1 loop run 2 times, j =2 loop runs 3 times and keeping going when j = n -1 loop will run n times.
```

With 1 + 2+3+4+5 …n = (n*(n+1))/2 = (n^2)/2 +1 → It runs about n*((n^2)/2 +1) time. But in big O we only care the fastest growing term → Big O(n^3)

```
                        y = y + a[k]; // O(1) It does not depend on array length

                        k = k + 1; // O(1) It does not depend on array length

                    }
                    j = j + 1; // O(1) It does not depend on array length

                }
                if (b[i] == y) {  // O(1) It does not depend on array length
                    x++;  // O(1) It does not depend on array length

                }
                i = i + 1; // O(1) It does not depend on array length

            }
            return x; // O(1) It does not depend on array length
```

→ For this method we only care the fastest growing term → Overall the running time of this method is **O (n^3)**

(3)

```
// n is the length of array a
// p is an array of integers of length 2
// initial call: method3(a, n-1, p)
// initially p[0] = 0, p[1] = 0
method3(int[] a, int i, int[] p)
    if (i == 0) {
        p[0] = a[0]; // O(1) It does not depend on array length
        p[1] = a[0];  // O(1) It does not depend on array length
    }
    else {
        method3(a, i-1, p); // Recursive call n-1 times → O(n)
        if (a[i] < p[0]]) {
         p[0] = a[i]; // O(1) It does not depend on array length
        }
        if (a[i] > p[1]]) {
         p[1] = a[i]; // O(1) It does not depend on array length
        }

    }
```

→ For this method we only care the fastest growing term → Overall the running time of this method is **O(n)**

(4)

```
// initial call: method4(a, 0, n-1) // n is the length of array a
public static int method4(int[] a, int x, int y)
{
  if (x >= y)   // O(1) It does not depend on array length
  {
      return a[x];  O(1) It does not depend on array length

  }
  else
  {
        z = (x + y) / 2; // integer division  // O(1)
        u = method4(a, x, z);  // Recursive call for the left half of array
O(log n)
        v = method4(a, z+1, y);  // Recursive call for the right half of array O(log
n)
        if (u < v) return u;  O(1) It does not depend on array length.

        else return v;   O(1) It does not depend on array length

  }
}
```

- The initial call for method4 will have length of array is n
- When the function step goes to u and v, it will start recursive calls:
  - the first calls: method4 will have length of array is n/2.
  - the second calls: method4 will have length of array is n/4 = n/2^2
  - the third calls: method4 will have length of array is n/8 = n/2^3
  - the fourth calls: method4 will have length of array is n/16 = n/2^4
  - and keeping going

n → n/2 → n/2^2 → n/2^3→n/2^4 → n/2^x = 1 (with x is number of the elements to check)
calculate x ➔ n = 2^x ➔ x = log2(n) = log(n) → Overall the running time of this method is **O(log n)**

In the (4): the running time is O(log n)

Problem 2 (20 points) This problem is about the stack and the queue data structures that are described in the textbook.

(1) Suppose that you execute the following sequence of operations on an initially empty stack. Using Example 6.3 in the textbook as a model, complete the following table.

Operation	Return Value	Stack Contents
push(10)	-	(10)
pop( )	10	()
push(12)	-	(12)
push(20)	-	(12,20)
size( )	2	(12,20)
push(7)	-	(12,20,7)
pop( )	7	(12,20)
top( )	20	(12,20)
pop( )	20	(12)
pop( )	12	()
push(35)	-	(35)
isEmpty( )	false	(35)

(2) Suppose that you execute the following sequence of operations on an initially empty queue. Using Example 6.4 in the textbook as a model, complete the following table.

Operation	Return Value	Queue Contents (first ← Q ← last)
enqueue(7)	-	(7)
dequeue( )	7	()
enqueue(15)	-	(15)
enqueue(3)	-	(15,3)
first( )	15	(15,3)
dequeue( )	15	(3)
dequeue( )	3	()
first( )	Null	()
enqueue(11)	-	(11)
dequeue( )	11	()
isEmpty( )	true	()
enqueue(5)	-	(5)

Problem 3 (60 points) The goal of this problem is: (1) practice of using and manipulating a doubly linked list and (2) practice of designing and implementing a small recursive method. Write a program named *Hw2_p3.java* that implements the following requirement:

- This method receives a doubly linked list that stores integers.
- It reverses order of all integers in the list.
- This must be done a ***recursive*** manner.
- The signature of the method must be:

```
public static void reverse(DoublyLinkedList<Integer> intList)
```

- You may want to write a separate method with additional parameters (refer to page 214 of the textbook).
- You may not use additional storage, such as another linked list, arrays, stacks, or queues. The rearrangement of integers must occur within the given linked list.

An incomplete *Hw2_p3.java* is provided. You must complete this program.

You must use the *DoublyLinkedList* class that is posted along with this assignment. You must not use the *DoublyLinkedList* class that is included in textbook's source code collection.

Note that you must not modify the given *DoublyLinkedList* class and the *DoubleLinkNode* class.

Deliverable

No separate documentation is needed for the program problem. However, you must include the following in your source code:

- Include the following comments above each method:
- Brief description of the method
- Input arguments
- Output arguments
- Include inline comments within your source code to increase readability of your code and to help readers better understand your code.

You must submit the following files:

- *Hw2_p1_p2.pdf*. This file must include the answers to problems 1 and 2
- *Hw2_p3*.java This file must include completed code for problem 3.

Combine all files into a single archive file and name it *LastName_FirstName_hw2.EXT*, where *EXT* is an appropriate archive file extension, such as *zip* or *rar*.

Grading

Problem 1 (20 points):

- Up to 4 points will be deducted for each wrong answer.

Problem 2-(1) (10 points):

- Up to 8 points will be deducted if your answer is wrong.

Problem 2-(2) (10 points):

- Up to 8 points will be deducted if your answer is wrong.

Problem 3 (60 points):

- If your program does not compile, 32 points are deducted.
- If your program compiles but causes a runtime error, 24 points are deducted.
- If there is no output or output is completely wrong, 20 points are deducted.
- If your program is partly wrong, up to 20 points are deducted