CMPT 280 Tutorial: Timing Analysis

O Ω Θ

Name	Description
Big-O (O):	An upper bound on the growth rate of a function. It describes the *worst-case* scenario
Big-Omega (Ω):	A lower bound on the growth rate. It describes the *best-case* scenario
Big-Theta (Θ):	the function grows at the same rate in both the best and worst cases

Question 1

Use the statement counting approach to determine the number of statements executed by the following Java method in the **worst case** (use the statement-counting approach). Assume that all methods and constructors called by the given method are Θ(1).

```
StringBuilder t = new StringBuilder(s); // +1 (outside loop)

 for (int i = 0; i < s.length(); i++) {  // Loop structure:
     // For each iteration:
     // 1. Loop test:   +1
     // 2. If check:    +1
     // 3. Body:        +1 (worst case, always executes)
     if (Character.isLowerCase(s.charAt(i))) {
         t.setCharAt(i, (char)(s.charAt(i) - 'a' + 'A'));
     }
 }
 // After the loop: one extra test when i == s.length(): +1
```

Question 2

Determine the number of statements executed by the Java method in question 1 in the **best case**.

```
StringBuilder t = new StringBuilder(s); // +1 (outside loop)

 for (int i = 0; i < s.length(); i++) {  // Loop structure:
     // For each iteration:
     // 1. Loop test:   +1
     // 2. If check:    +1
     // 3. Body:        0 (best case, never executes)
     if (Character.isLowerCase(s.charAt(i))) {
         t.setCharAt(i, (char)(s.charAt(i) - 'a' + 'A'));
     }
 }
 // After the loop: one extra test when i == s.length(): +1

 return t.toString(); // +1 (outside loop)
```

Question 3

Express the worst and best case time complexities of the method in question 1 in Big-O and Big-Ω notation, respectively. If possible, express the overall time complexity of the method using Big-Θ notation.

worst case: O(n)
best case: Ω(n)
overall: since the best and worst case growth functions are the same, the method is Θ(n)

Question 4

Do the answers to question 3 change if the call to t.toString() is Θ(n) rather than Θ(1)? Why or why not?

no, because the loop is already O(n) in the worst case so adding an additional O(n) cost is 2O(n) which is just O(n). similarly for the best case, the cost would become 2Ω(n) which is just Ω(n).

Question 5

Determine the exact number of statements executed by the following method in the best- and worst-cases.

```
public Integer[][] flip(Integer[][] matrix) {
    int h = matrix.length;            // +1
    int w = matrix[0].length;         // +1
    Integer[][] T = new Integer[h][w]; // +1

    for (int i = 0; i < h; i++) {        // Loop structure:
        // For each iteration of outer loop:
        // 1. Outer loop test: +1
        // 2. Outer loop increment: counted as part of loop structure
        for (int j = 0; j < w; j++) {   // Inner loop structure:
            // For each iteration of inner loop:
            // 1. Inner loop test: +1
            // 2. Assignment in body: +1
            // 3. Inner loop increment: counted as part of loop structure
            T[j][i] = matrix[i][j];
        }
        // After inner loop: one extra inner test when j == w: +1
    }
```

Question 6

Express the overall time complexity of the method in question 7 using the appropriate notation(s) (big-O, big-Θ, big-Ω).

Θ(hw)

Question 7

Determine the exact number of statements executed by the following method in the best- and worst-cases.

```
public void lower_triangle_row_sum(Float[][] s) {
    // +1: if condition (best/worst case, always executed)
    if ( s.length != s[0].length )
        // +1: throw statement (best case only, stops execution)
        throw new RuntimeException("Array is not square!");

    // +1: float sum = 0;
    float sum = 0;

    // Outer loop (i from 0 to n-1):
    for(int i = 0; i < s.length; i++) { //
        // +1: float rowsum = 0.0;
        float rowsum = 0.0;

        // Inner loop (j from i down to 0):
        for(int j = i; j >= 0; j--) {
            // +1: rowsum += s[i][j]; (per iteration)
            rowsum += s[i][j];
            // +1: inner loop test (j >= 0) (per iteration + 1 for final test)
        }
        // +1: System.out.println(...) (per i)
        System.out.println("Sum of row " + i + ": " + rowsum);
        // +1: outer loop test (i < s.length) (per iteration + 1 for final test)
```

Question 8

Express the overall time complexity of the method in question 7 using the appropriate notation(s) (big-O, big-Θ, big-Ω).

The method is $O(n^2)$ (worst case) and $\Omega(1)$ (best case). Since the best and worst cases are not the same, no big- Θ statement can be made.

Question 9

Prove that the following code is $O(\log n)$ by calculating the exact number of statements executed and then expressing that function in big-O notation. Assume n is an `float` variable with a large positive value.

```
float i = n;                        // +1 (initialization)
while(i > 1.0) {                    // loop test counted per iteration (+1 per loop)
    float jump_dist = i - i / 2.0;            // +1 (per iteration)
    System.out.println("Jump " + jump_dist + " light years..."); // +1 (per iteration)
    i = i / 2.0;                    // +1 (per iteration)
    // total: 4 statements per loop iteration (including test)
}
// After the loop: one extra test when i <= 1.0 (+1)
System.out.println("You're now within 1 light year of your destination."); // +1
```

Number of statements per loop iteration: 4

Number of times the loop executes:

Question 10

Determine the time complexity of the following pseudocode snippet using the active operation approach.

```
Let A be a 2D array of size n by n.
Let T be an AVL tree with m items.

for each row r of A:                    // outer loop: n times
    for each column c of A:             // inner loop: n times per row, total n^2 times
        A[r][c] = A[r-1][c] + A[r][c-1] - A[r-1][c-1]   // O(1) per iteration
        T.insert(A[r][c])                                 // O(log m) per iteration (active op)
```

The active operation could be the assignment or the insert, since both are executed n^2 times, but only the insert is non-constant time.
Therefore, the insert operation is chosen as the active operation.

In the worst case, the active operation (insert) has a cost of $O(\log m)$ where m is the number of items in the AVL tree.

Question 11

Determine the time complexity of the following pseudocode snippet using the active operation approach (careful! This one is tricky!).

```
Let G be a weighted graph that is implemented with an adjacency list
    with |V| nodes and |E| edges.
Let H be a heap of edges ordered by edge weight (initially empty).

for each vertex v in G:
    if there is an edge from vertex v to vertex 0:
        insert the edge (v,0) to H
```

There are two possible active operations: Line 6 and line 7 since both execute the same number of times in the worst case and both have non-constant cost. Since it isn't immediately obvious which should be the active operation, we perform the analysis for both.

Question 12

Determine exact number of statements executed by the following pseudocode function in the worst case (this one is a bit tricky too!):

```
Algorithm treeconcat(root)

Let root be the root node of an m-ary tree containing n nodes where each
    node contains a string and each non-leaf node has exactly
    m children.
Let s be the empty string.

s <- root.string()                  // assign the string stored in root to s

if root is not a leaf node:         // Assume this line is O(1)
    for each child q of root:
        s <- s + treeconcat(q)      // string concatenation of s with the
                                    // return value, assume the concatenation
                                    // operation is O(1).

return s
```