HW4

To Turn in: please submit the questions and your answers below them in a pdf file on canvas.

Perform a time-complexity (Big-O) analysis for each of the next three problems (problems 1, 2, and 3). For full credit you should be able to produce a logical justification for your answer (a growth rate function can help demonstrate this – but is NOT required – so at least show in general why the Big-O is what it is). Equations you may need: (1) $1 + 2 + 3 + 4 + \ldots + n = (1 + n) * n / 2$; (2) $1 + a + a^2 + a^3 + \ldots + a^n = (a^{n+1} – 1) / (a-1)$.

1. (40 Points)

```
public static void two(int n)
{
   if(n > 0) (1)
   {
      System.out.println("n: " +n); (1)
      two(n - 1); (n)
      two(n - 1); (n)
   }
   else if (n < 0)
    {
       two(n + 1); // identical as above for negative numbers
       two(n + 1);
       System.out.println("n: " + n);
    }
}
```

Answer:

For each method call, the initial cost is T(n) = 2 + T(n-1) + T(n-1)

The next cost would be 2 + (2 + T(n-2) + T(n-2)) + (2 + T(n-2) + T(n-2))

Each of those recursive calls, bundled in parentheses, doubles the number of recursive calls. It will double every time the recursion is called, meaning the cost added onto the initial 2's is 2 * 2 * 2 * … * 2, until the n passed in is 0, meaning it does this n times, so the exponent above 2 is n. The calculation of the extra 2's which correspond to the if statement and print statement, it adds double the amount every recursive call, meaning the result is 2 + 2*n, n times, so the calculation is n(2 + 2*n) or 2n + 2n^2. However, the 2^n calls is still higher order, so **the big-oh notation for this is O(2^n)**

2. (30 Points)

```
public void three(int n)
{
  int i, j, k; (1)
  for (i = n/2; i > 0; i = i/2) (log2(n) - 1 + 1)
      for (j = 0; j < n; j++) ((log2(n) – 1)* n + 1)
            for (k = 0; k < n; k++) (((log2(n) – 1)* n) * n + 1)
                  System.out.println("i: " + i + " j: " + j+" k: " + k);
(((log2(n) – 1)* n) * n)

}  // end three
```

Answer:

The sum of the time cost is 1 + log(n) + nlog(n) – n + 1 + n^2log(n) – n^2 + 1 + n^2log(n) – n^2, combining like terms, the highest order term is n^2(log(n)) * 2, so **the big-oh notation is O(n^2(log(n)))**. This is due to embedded for loops, which multiply their time costs with each other THEN add them, as opposed to just adding them together. Each for loop statement executes the number of times of the previous for loop runs, plus 1 to consider for when it checks for a value that causes the loop to exit.

3. (30 points)

```
public static void four(int n)
{
    if (n > 1) (1)
    {
        System.out.println(n); (1)
        four(n-1); (n-1)
    }
    for (int i = 0; i < n; i++) (n+1)
      System.out.println(i); (n)
}
```

Answer:

During the first one, the time cost is T(n) = 2 + T(n-1) + n+1 + n or 3 + T(n-1) + 2n
The second time cost is 2 + (2 + T(n-2) + n+1 + n) + n+1 + n), which sums to be 6 + T(n-2) + 4n
This whole method recursively calls itself until n = 1, where it does the for loop one last time and stops, meaning it runs n-1 times.
Following the trend of the constant, which starts at 3 (or 2 + 1), it doubles every recursive call, so this portion equals 3(n-1) or 3n – 3
The last portion, 2n (or n + n) also doubles every recursive call, meaning it ends up running 2n(n-1) times, or 2n^2 – 2n.
Adding these two segments together gives us 2n^2 – 2n – 3n – 3. Combining like terms we get 2n^2 – 5n – 3, with 2n^2 being the highest order, so **the big-oh notation for this function is O(n^2)**