Due at 11:59PM, Friday, February 21st on Canvas. Submissions must be ***typed*** (except for questions labeled with a “ ” icon) and in PDF format. Show your work for full credit!

1. The ***formal definition*** of big-O notation states that $f(n) \in O(g(n))$ if there are real-valued constants $C, K > 0$ such that for all $n \geq K$, $f(n) \leq Cg(n)$. Using the formal definition, show that

   (a)
   $$7n^2 + 20n - 91 \in O(n^2),$$
   using $C = 8$. What's the ***smallest*** value of K you can choose?

   (b)
   $$7n^5 + 18n^4 - 3n^3 + 6n^2 - 2n + 5 = O(n^5),$$
   using whatever values of C and K you like.

2. ( ) In class, we gave an example of how polynomials grow slower than exponentials by showing, using a ***proof by induction***, that
   $$\lim_{x\to\infty} \frac{x^n}{e^x} = 0,$$
   for all integers $n \geq 0$. The same idea can be used to prove that powers of logarithmic functions (e.g. $(\log_2 x)^{10}$) grow slower than polynomials. Give another ***proof by induction*** of the fact
   $$\lim_{x\to\infty} \frac{(\ln x)^n}{x} = 0,$$
   for all integers $n \geq 0$. Just like in the proof from class, you should be applying l'Hopital's rule ***exactly once*** in the inductive step.

3. Using the guidelines from the slides, convert the following Java method into pseudocode:

```
public static int binarySearch(int[] array, int target){
    int n = array.length;
    int low = 0;
    int high = n-1;
    while (low <= high){
        int mid = (low + high) / 2;
        if (array[mid] == target){
            return mid;
        }
        else if (array[mid] < target){
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return -1;
}
```

Your answer should be in the following form:

BINARYSEARCH($A[1,\ldots,n]$, *target*):
    // your code here

(Note: be careful, the array A written above starts at 1, not 0. The math-y way of rounding down numbers is "$\lfloor x \rfloor$", but you can also just write something like "FLOOR(x)".)

4. For each of the algorithms below, do the following steps:

   (i) Write down a big-O estimate, as a function of m and/or n, for
   - the runtime of any subroutine call (this only applies to ALG2), and
   - the maximum number of iterations each loop can run for.

   (ii) Using (i), write down an expression for the total runtime of the algorithm.

   (iii) Simplify the expression from (ii) to get a simple function $O(f(n))$ or $O(f(m, n))$.

   Try to make your function as tight as possible to the original expression (e.g. $5n \log_2 n = O(n \log n)$, not $O(n^2)$ or $O(n)$).

```
ALG1(n):
    count := 0
    for a = 1, ..., n
        for b = 1, ..., n
            for c = b, ..., n
                for d = b, ..., c
                    count := count + 1
        for e = n^3, ..., n^4
            f := 1
            while f < n
                f := 3c
                count := count + 1
    return count

ALG2(m, n):
    count := 0
    for a = 5, ..., 5n
        for b = 1, ..., ⌊ln m⌋
            c := 1
            while c ≤ m
                c := c + 1
                d := m
                while d > 1
                    d := ⌊d/2⌋
                    count := count + 1
        count := count + ALG1(m)
    for e = 1, ..., 3m
        for f = 1, ..., 7n^2
            for g = 100, ..., 1000
                count := count + 1
    return count
```

(Note: if the function were

SAMPLE(n):
  $count := 0$
  for $a = 1, \ldots, n$
    BUBBLESORT(n)
    for $b = a, \ldots, n$
      $count := count + 1$
  return $count$

where BUBBLESORT takes $\Theta(n^2)$ time to sort an array of size n, then the solution would be:

(i) BUBBLESORT: $O(n^2)$, a: $O(n)$, b: $O(n)$

(ii) $O(n)[O(n^2) + O(n)]$

(iii) $O(n)[O(n^2) + O(n)] = O(n)O(n^2) = O(n^3)$.

)