Task 1: Hello, Definition (4 points)

This task consists of small problems involving working directly with the definition of Big-O. (Hint: One very simple solution for the first two problems below involves just taking limits.)

(1) Show, using either definition, that f (n) = n is O(n logn).

By using the definition of Big-O notation, $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n}{nlogn} = \frac{1}{logn} = \frac{1}{\infty} = 0.$$

Therefore, we can say that this function is bounded by g(n), O(nlogn).

(2) In class, we saw that Big-O multiplies naturally. You will explore this more formally here. Prove the following statement mathematically (i.e. using proof techniques learned in Discrete Math):
Proposition: If d(n) is O(f (n)) and e(n) is O(g (n)), then the product d(n)e(n) is O(f (n)g (n)).

Proof:
We want to show that d(n)e(n) is a product of O(f (n)g (n)). That is, by using direct proof method, $\frac{d(n)e(n)}{f(n)g(n)} = k_1 k_2$.

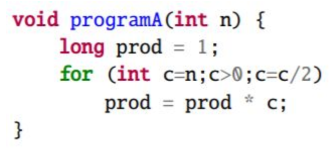
$$\lim_{n\to\infty} \frac{d(n)}{f(n)} = k_1$$

$$\lim_{n\to\infty} \frac{e(n)}{g(n)} = k_2$$

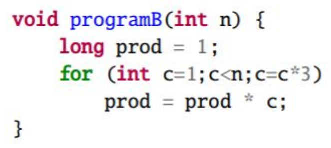
$$\lim_{n\to\infty} \frac{d(n)e(n)}{f(n)g(n)} = k_1 k_2$$

Hence, the product of d(n)e(n) is O(f (n)g (n)) by both of the constants.

(3) There is a function void fnE(int i, int num) that runs in 1000·i steps, regardless of what num is. Consider the following code snippet:



```
void programA(int n) {
    long prod = 1;
    for (int c=n;c>0;c=c/2)
        prod = prod * c;
}
```



```
void programB(int n) {
    long prod = 1;
    for (int c=1;c<n;c=c*3)
        prod = prod * c;
}
```

What’s the running time in Big-O of fnA as a function of n, which is the length of the array S. You should assume that it takes constant time to determine the length of an array.

Answer :

The outer loop is 1,2,3,4+...+(n-1)
Inner loop(fnE) is a for loop with j=i in range 1000*i.

Let f(i) be the time taken for the nested loop.

f(i) = (n-1-i) because values of j depend on i(j=i) which is also less than n.

$$\sum_{i=0}^{n} f(i) = \sum_{i=0}^{n} n - 1 - i$$

This summation is (n-1)+(n-2)+... (n-1-(n-1))which is identical to 1+2+3+...+ (n-1) which equals to $\frac{n(n-1)}{2}$; therefore, O(n^2).

(4) Show that $16n^2 + 11n^4 + 0.1n^5$ is not O(n^4)

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{16n^2+11n^4+0.1n^5}{n^4} = \infty$ which does not satisfy the definition of Big-O, $\frac{f(n)}{g(n)} < \infty$.

Task 2: Poisoned Wine (2 points)

Let the rats be the testers r
Let the no.of of wine be n

Suppose n = 15
Let's take note or mark the bottle of wine by the binary representation or bits.

E.g Wine bottle no.1 can be represented by 0001. Note: There can be a mixture of a drop of wine.

Claim: The number of rats needed in this case will be $log_2(n+1)$. $log_2(15+1)$=4.
Therefore, we need 4 mouses.

Proof:
We first get the 4-bit binary representation of the ***n***th bottle. If the ***i***th bit of the binary number is 1, we let the ***r***th rat drink the wine from that bottle. Note: 0 < ***n***th <15, 0 < ***i*** <4.

The reason why 4 mouses are enough is because with 4 testers each being dead or alive in the end, we can have a total of 16 combinations (2^4), 2 is the outcome of being dead or alive and 4 is no.of rats, now we can identify which bottle is poisoned from 15 bottles, on the day 31.

For example, the 1st and 3rd mouse are dead. Then, the bottle 5th is poisoned.
00000001
00000010
00000011
00000100
00000101 1st and 3rd rats drink the same bottle of poison.
00000110
00000111
00001000
00001001
00001010
00001011

Task 3: How Long Does This Take? (2 points)
For each of the following functions, determine the running time in terms of Θ in the variable n. Show your work. We're more interested in the thought process than the final answer.

```
void programA(int n) {
        long prod = 1; //  constant time
        for (int c=n;c>0;c=c/2) O(n)// rep n/2 times
                prod = prod * c;
}

void programB(int n) {
        long prod = 1;
        for (int c=1;c<n;c=c*3) rep n*3 times
                prod = prod * c;
}
```

Program A
Assume that when n = 4.

Count	Steps taken
4	Some constant c
2	Some constant c
1	Some constant c

n/2	Some constant c

Hence, the iteration count will be an input n.

Therefore, this iteration takes c(n/2) steps in all, which is O(n/2), O(n),in a worst case scenario.
The best case for this is when n=1; therefore, it will take O(1) running time.

Θ(n) because $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$. In this case, $\lim_{n\to\infty} \frac{n}{n/2} = 2 > 0$.

Assume that when n = 9.
Program B

Count	Steps taken
c=1	k
c=3	k
c=9	k
3^n	k

This loop will take $3^n k$ = O($log_3 n$) in the worst case.
In best case, it will take constant time O(1) when n<=3.

Θ ($log_3 n$) because $\lim_{n\to\infty} \frac{log_3 n}{log_3 n} > 0$.

Task 4: Halving Sum (4 points)
Our course staff has a signature process for summing up the values in a sequence (i.e., array). Let X be an input sequence consisting of n floating-point numbers. To make life easy, we'll assume n is a power of two—that is, n = 2 k for some nonnegative integer k. To sum up these numbers, we use the following process, expressed in a Python-like language:

```
def hsum(X): # assume len(X) is a power of two

while len(X) > 1: (1) allocate Y as an array of length len(X)/2
                  (2) fill in Y so that Y[i] = X[2i] + X[2i+1] for i = 0, 1, ..., len(X)/2
                  (3) X = Y
                  return X[0]
```

This task has two parts:

Part I: (2 points) Observe that the amount of work done in Steps (1)–(3) is a function of the length of X in that iteration. If z = X.length at the start of an iteration, how much work is being done in that iteration as a function z (e.g., 10z 5 + z logz or k1z 2 +k2z for some k1,k2 ∈ R+)? Don't use Big-O; answer in terms of k1 and k2. Let's make some assumptions here. For some k1,k2 ∈ R+:
• Allocating an array of length z costs you k1 · z.
• Arithmetic operations, as well as reading a value from an array and writing a value to an array, can be done in k2 per operation.
Your answer for this part should look like the following:

(1) $k_1 * \frac{z}{2}$
(2) $k_2 * \frac{z}{2}$
(3) k_2

$$k_1 * \frac{z}{2} + (k_2 * \frac{z}{2}) + k_2 = (\frac{k_1}{2})z + (k_2 + \frac{k_2 z}{2})$$

Part II: (2 points) Then, you'll analyze the running time of the algorithm (remember to explain how you get the running time you claim). To help you get started, make a table of how X.length changes over time if we start with X of length, say, 64. How does this work in general? (Hint: The geometric sum formula presented in class may come in handy.)

X.length	Steps taken
64	n
32	n/2
16	n/4
..	..
1	*logn*

Total running time is

$$\sum_{i=0}^{z/2} k_2 * (\tfrac{1}{2})^i$$

The summation is

$$(k_2 * \tfrac{1}{2}) + (k_2 * \tfrac{1}{4}) + ... + (k_2 * (1/2)^{z/2}) = (1/2)^{z/2+1} - 1$$

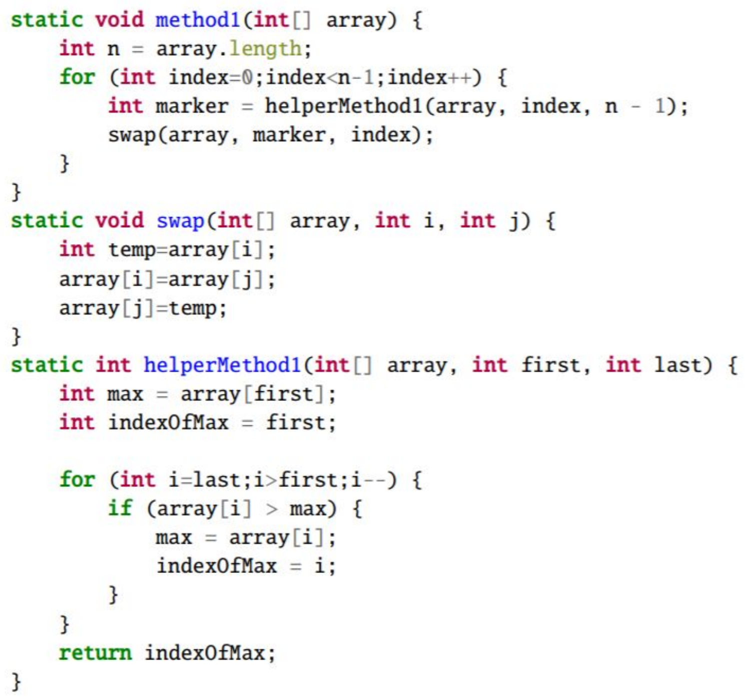
Total running time is

$$O(\log n * (½)^{\frac{n}{2}+1} - 1) = O(n\log n)$$

Task 5: More Running Time Analysis (6 points)

For the most part, we have focused almost exclusively on worst-case running time. In this problem, we are going to pay closer attention to these Java methods and consider their worst-case and best-case behaviors. The best-case behavior is the running time on the input that yields the fastest running time. The worst-case behavior is the running time on the input that yields the slowest running time.

(1) Determine the best-case running time and the worst-case running time of method1 in terms of Θ.



```java
static void method1(int[] array) {
    int n = array.length;
    for (int index=0;index<n-1;index++) {
        int marker = helperMethod1(array, index, n - 1);
        swap(array, marker, index);
    }
}
static void swap(int[] array, int i, int j) {
    int temp=array[i];
    array[i]=array[j];
    array[j]=temp;
}
static int helperMethod1(int[] array, int first, int last) {
    int max = array[first];
    int indexOfMax = first;

    for (int i=last;i>first;i--) {
        if (array[i] > max) {
            max = array[i];
            indexOfMax = i;
        }
    }
    return indexOfMax;
}
```

The worst case running time of method 1 is O(n^2). As it needs to compare everything in array.
The best case is when the length of the array is less than 2. O(1).

$$\lim_{n\to\infty} \frac{n^2}{n^2} = 1$$

Θ(n^2)

(2) Determine the best-case running time and the worst-case running time of method2 in terms of Θ.

static boolean method2(int[] array, int key) {
int n = array.length;
for (int index=0;index<n;index++) {
if (array[index] == key) return true;
}

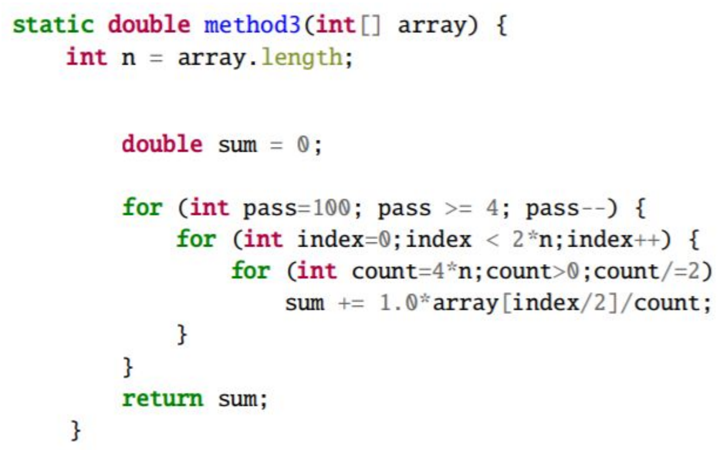
return false;
}

Best case:
The key on the first index. O(1)

Worst case:
When the key is on the last element of the array n length.
O(n)

The f(n) = n is $\Theta(n)$ because
$$\lim_{n\to\infty} \frac{n}{n} = 1$$

(3) Determine the best-case running time and the worst-case running time of method3 in terms of Θ.



```java
static double method3(int[] array) {
    int n = array.length;

        double sum = 0;

        for (int pass=100; pass >= 4; pass--) {
            for (int index=0;index < 2*n;index++) {
                for (int count=4*n;count>0;count/=2)
                    sum += 1.0*array[index/2]/count;
            }
        }
        return sum;
    }
```

Best case :
O(1). When pass =4.
Worst case:

```
for (int pass=100; pass >= 4; pass--) ← O(1)
    for (int index=0;index < 2*n;index++) ← O(n)
        for (int count=4*n;count>0;count/=2) ← O(logn)
```

The f(n) = nlogn is $\Theta(nlogn)$ because
$$\lim_{n\to\infty} \frac{nlogn}{nlogn} = 1$$

Task 6: Recursive Code (6 points)

For each of the following Java functions:

(1) Describe how you will measure the problem size in terms the input parameters. For example, the input size is measured by the variable n, or the input size is measured by the length of array a.
(2) Write a recurrence relation representing its running time. Show how you obtain the recurrence.
(3) Indicate what your recurrence solves to (by looking up the recurrence in our table).

```
// assume xs.length is a power of 2
int halvingSum(int[] xs) {
if (xs.length == 1) return xs[0]; ← O(1)
else {
int[] ys = new int[xs.length/2]; ← O(1)
for (int i=0;i<ys.length;i++) ←  O(n/2) → O(n)
ys[i] = xs[2*i]+xs[2*i+1]; ← O(1)
return halvingSum(ys); ←  T(n/2)
}
```

$T(n) = T(n/2) + O(n)$ solves to $O(n)$.

```
}
int anotherSum(int[] xs) {
if (xs.length == 1) return xs[0]; ← O(1)
else {
int[] ys = Arrays.copyOfRange(xs, 1, xs.length); ←  O(n-1) → O(n)
return xs[0]+anotherSum(ys); ← T(n-1)
}



}
int[] prefixSum(int[] xs) {
if (xs.length == 1) return xs; ← O(1)
else {
int n = xs.length; ← O(1)
int[] left = Arrays.copyOfRange(xs, 0, n/2); ← O(n/2)
left = prefixSum(left); ← T(n/2)
int[] right = Arrays.copyOfRange(xs, n/2, n); ← O(n/2)
right = prefixSum(right); ←  T(n/2)
int[] ps = new int[xs.length]; ← O(1)
int halfSum = left[left.length-1]; ← O(1)
for (int i=0;i<n/2;i++) ← n+1
```

```
{ ps[i] = left[i]; }← O(1)
for (int i=n/2;i<n;i++) n+1
{ ps[i] = right[i - n/2] + halfSum; }← O(1)
return ps;← O(1)
```
$T(n) = 2T(n/2) + O(n)$ solves to $O(n \log n)$.
```
}
}
```

Task 7: Counting Dashes (2 points)

Consider the following program:
```
void printRuler(int n) {
if (n > 0) {
printRuler(n-1);
// print n dashes
for (int i=0;i<n;i++) System.out.print('-');
System.out.println();
// --------------
printRuler(n-1);
}
}
```
We would like to know the total number of dashes printed for a given n. If we are to write a recurrence for that, we will get
$g(n) = 2g(n-1)+n$, with $g(0) = 0$,
where the additive n term stems from the fact that we print exactly n dashes in that function call.
It may seem hopeless to try to solve this recurrence directly, but you may recall that the number of instructions taken to solve tower of Hanoi on n discs is given by the recurrence
$f(n) = 2f(n-1)+1$, with $f(0) = 0$.
As you showed in a previous assignment, f (n) has a closed-form of f (n) = 2
n −1.
The two recurrences are strikingly similar. In this problem, we'll analyze g (n) using our knowledge of f (n). Since f (n) and g (n) have similar recurrences, differing only in an n term, we're going to guess that
$g(n) = a \cdot f(n)+b \cdot n +c$ (1)
The following steps will guide you through determining the values of a, b, and c—and verifying that our guess indeed works out. In your writeup, clearly show your work.
(i) We'll first figure out the value of c. What do you get when plugging in n = 0 into equation (1)? It helps to remember that f (0) = g (0) = 0. (Hint: c should be 0.)

$g(0) = a * f(0) + b * 0 + c$
$g(0) = c$
$C = 0$

(ii) To figure out the values of a and b, we'll plug in g (n) from equation (1) into the recurrence g (n) = 2g (n −1)+n. You should be able write it as
and solve for a and b such that P = 0 and Q = 0.
Keep in mind: Because f (n) = 2f (n −1)+1, we know that f (n)−2f (n −1) = 1.

Find a

$a * f(n) + bn = 2(a * f(n-1) + b(n-1)) + n$

$a * f(1) + b = 2\ (a(f(0)+b(0)) + 1$

$a * f(1) + b = 2(af(0)) + 1$

$a * (f(1) - 2f(0)) = 1 - b$

$a = 1 - b$

Find b

$a * f(n) + bn = 2(a * f(n-1) + b(n-1)) + n$

$a * f(2) + 2b = 2(a * f(1) + b(1)) + 2$

$a * f(2) = 2a * f(1) + 2$

$a * f(2) - 2\ af(1) = 2$

$a = 2$

$2 = 1 - b$

$b = -1$

(iii) Derive a closed form for g (n).

$g(n) = 2f(n) - n$

$= 2(2^n - 1) - n$

$= 2^{n+1} - 2 - n$

(iv) Use induction to verify that your closed form for g (n) actually works.

$P(n) := g(n) = 2^{n+1} - 2 - n$

Base case :
g(0) = 2-2-n=0 P(0) is true.
Inductive step:
Assume that $P(k)$ is true. g(k) $= 2^{k+1} - 2 - n$

Prove that P(k+1) is true. Note that g(0) = 0

$g(k+1) = 2^{k+2} - 2 - (k+1)$

$= 2 * 2^{k+1} - 2 - k - 1$

Let k+1 = z

$= 2^{z+1} - 2 - z$

Therefore, g(n) is true.