1. Provide O(f(N)) for the following

a) N + 1
b) 1 + 1/N
c) (1 + 1/N)(1 + 2/N)
d) $2N^3 + 15\ N^2 + \ N$
e) lg(2N)/lg N
f) $\lg(N^2 + 1) / \lg N$
g) $N^{100} / 2^N$

Answer:

a) O(N)
b) O(1)
c) O(1)
d) $O(N^3)$
e) O(1): lg(2N)/lg N = (lg2+lgN)/lgN=lg2/lgN + 1
f) O(1): $\lg(N^2 + 1) / \lg N \sim \lg(N^2) / \lg N = 2$, when N is very large
g) O(1): polynomial grows slower than exponential function

2. Which ones are correct according asymptotic notation definitions?

1) $3n^2 - 100n + 6 = O(n)$
2) $3n^2 - 100n + 6 = O(n^2)$
3) $3n^2 - 100n + 6 = O(n^3)$
4) $3n^2 - 100n + 6 = \Omega(n)$
5) $3n^2 - 100n + 6 = \Omega(n^2)$
6) $3n^2 - 100n + 6 = \Omega(n^3)$
7) $3n^2 - 100n + 6 = \Theta(n)$
8) $3n^2 - 100n + 6 = \Theta\ (n^2)$
9) $3n^2 - 100n + 6 = \Theta\ (n^3)$

Answer: The correct ones are: 2, 3, 4, 5, 8

3. Provide running time in big O notation for the following methods

```
29     public void a(int N) {
30         int sum = 0;
31         for (int i = N; i > 0; i /= 2)
32             for(int j = 0; j < i; j++)
33                 sum++;
34     }
```

Answer: Assume N = 2^m, Line 33 is executed about this number of times:

N + N/2 + N/4 + … + 2 + 1

= 2^m + 2^(m-1) + 2^(m-2) + … + 2 + 1

=2^(m+1) - 1

=2N - 1

= O(N) (this is for best-case, worst-case, and average-case)

For other N values, choose a closest power of 2 number that is greater than N to do approximation.

```java
36    public void b(int N) {
37        int sum = 0;
38        for (int i = 1; i < N; i *= 2)
39            for (int j = 0; j < i; j++)
40                sum++;
41    }
```

Answer:

Function b does the similar thing as a, the only difference is that in function a i goes from high to low, while in function b i goes from low to high. Assume N = 2^m, Line 40 is executed about this number of times:

1 + 2 + 4 + … + N/4 + N/2

= 1 + 2 + 2^2 + … + 2^(m-2) + 2^(m-1)

=2^(m) - 1

=N - 1

= O(N) (this is for best-case, worst-case, and average-case)

```java
    public void c(int N) {
        int sum = 0;
        for (int i = 1; i < N; i *= 2)
            for (int j = 0; j < N; j++)
                sum++;
    }
```

Answer:

The outer for statement runs about lgN times, and the inner for statement runs N times during each outer for loop iteration. Thus running time is O(NlogN) for best-case, worst-case, and average-case.

4. What is the best-case and worst-case running time for these methods? Assuming inputs are all valid.

```java
public static long factorial(int n) {
    if (n<=1) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Answer:

The running time function can be written as the following:

$t(1) = c1$, where c1 is a constant

$t(n) = t(n-1) + c2$, where c2 is a constant

Use substitution method, we have

$t(n) = t(n-1) + c2$

$t(n-1) = t(n-2) + c2$

$t(n-2) = t(n-3) + c2$

….

$t(3) = t(2) + c2$

$t(2) = t(1) + c2$

Adding all the above equations together gives us

$t(n) = t(1) + (n-1)*c2 = c1 + (n-1)*c2 = O(n)$, which is best-case and worst-case running time.

```java
    public static void whiteList(int[] a, int[] keys) {
        if (a==null || keys==null) return; // invalid inputs

        for (int key: keys) {
            if (binarySearchI(a, key)<0) {
                //print if not in whitelist a
                System.out.println(key);
            }
        }
    }
```

Answer:

Assume neither a nor keys is null, n=a.length and m=keys.length

For binary search, best-case running time is O(1) and worst-case running time is O(lgn). Since the for statement loops m times, thus best-case running time is O(m) and the worst-case running time is O(mlgn).

6. Space complexity analysis

a) For int[] with size n, what is the O-notation for its space requirement?

Answer: O(n)

b) For Integer[][] with size n*m, what is the O-notation for its space requirement?

Answer: O(n*m)

c) For LinkedList<Integer> with n nodes, what is the O-notation for its space requirement?

Answer: O(n)

7. Provide best-case and worst-case space complexity analysis for these methods.

```java
public static int binarySearchR(int[] a, int key, int low, int high) {
    if (low>high) {
        return -1;
    } else {
        int mid = (low+high)/2;
        if (a[mid]<key) {
            return binarySearchR(a, key, mid+1, high);
        } else if (a[mid]>key){
            return binarySearchR(a, key, low, mid-1);
        } else {
            return mid;
        }
    }
}
```

Answer: Asssume a is not null and a.length is n.

Best-Case	Worst-Case
For the best-case, search key is found at the first try. binarySearchR only declares one variable inside it, and the method is only called once, s(n) = c = O(1).	For the worst-case, search key does not exists in the array, binarySearchR will be called at most about lgn times, which uses at most about lgn stack frames at a particular moment. Since the method only declares one variable, thus s(n) = c*lgn = O(lgn)

```java
private void resize(int max) {
    E[] temp = (E[]) new Object[max];
    for (int i = 0; i < n; i++) {
        temp[i] = a[i];
    }
    a = temp;
}
```

Answer:

The method declares two variables temp and I, which uses O(max) and O(1) space respectively. In total the space complexity is O(max) for both best-case and worst-case.