1. (10 Points) Fill in the blanks by selecting the statements that can be true based on the statement in the first column.

	g(n) grows slower than f(n)	g(n) grows the same rate as f(n)	g(n) grows faster than f(n)
f(n)=O(g(n))		T	T
f(n)=o(g(n))			T
f(n)=Ω(g(n))	T	T	
f(n)=ω(g(n))	T		
f(n)=θ(g(n))		T	

2. (10 Points) Fine an arrangement of the following functions f1, f2, …, f10 so that f1=O(f2), f2 = O(f3), …, f(9)=O(f10). Also indicate which functions grow at the same asymptotic rate.

lg(n!), ln(n), n, $2^{(2n)}$, $2^{(n+1)}$, nlg(n), lg(n), n^2, 1, $lg^2(n)$

Answer: Functions are arranged in increasing growth rate from top to bottom. Functions grow at the same rate are listed in the same row.

1) 1
2) ln(n), lg(n)
3) $lg^2(n)$
4) n
5) lg(n!), nlg(n)
6) n^2
7) $2^{(n+1)}$
8) $2^{(2n)}$

3. (20 Points) Provide best-case and worst-case running time and space complexity analysis in Big-Oh notation for the following **sort** method. For each case, provide an example input array and brief explanation.

	Big-O Notation	Input	Explanation
Best-Case Running Time	O(n)	array in sorted order, such as [1, 2, 3, 4, 5]	The condition test for embedded for loop is always false for each i value, so the method only does (n-1) comparisons, thus has O(n) running time
Worst-Case Running Time	O(n^2)	array in reverse sorted order, such as [5, 4, 3, 2, 1]	For each i, it requires i swaps, the total number of swaps is 1 + 2 + 3 + … + (n-1) = (n-1)*n/2 = O(n^2)
Best-Case Space Complexity	O(1)	Any input, such as [5, 3, 1, 4, 2]	Three int variables are declared, which takes O(1) space. Each of swap and isLessThan methods uses O(1) space, thus in total space complexity is O(1)
Worst-Case Space Complexity	O(1)	Any input, such as [5, 3, 1, 4, 2]	Same as the above.

```java
public class InsertionSort {
	/**
	 * Sort the input array into non-decreasing order
	 * @param a Input array, assume not null
	 */
	public static <T extends Comparable<T>> void sort(T[] a) {
		int n = a.length;
		for (int i = 1; i < n; i++) {
			// Insert a[i] into sorted section: 0, 1, ..., a[i-1]
			for (int j = i; j > 0 && isLessThan(a[j], a[j - 1]); j--) {
				swap(a, j, j - 1);
			}
		}
	}

	public static<T extends Comparable<T>> boolean isLessThan(T v, T w) {
		return v.compareTo(w) < 0;
	}

	public static<T> void swap(T[] a, int i, int j) {
		T t = a[i];
		a[i] = a[j];
		a[j] = t;
	}
}
```

4. (20 Points) Provide best-case and worst-case running time and space complexity analysis in Big-Oh notation for the following **pow**_2 method. For each case, provide an example input pair and brief explanation.

	Big-O Notation	Example Input	Explanation
Best-Case Running Time	O(lgn)	x=2 and n=31, where n is always an odd number (except base cases) in each pow_2 call	With this kind of input, we have these running time functions t(n)=1, when n<=1 t(n) = t(n/2) + C, when n>1 That gives us t(n) = O(lgn)
Worst-Case Running Time	O(n)	x=2 and n=32, where n is always an even number (except base cases) in each pow_2 call	With this kind of input, we have these running time functions t(n)=1, when n<=1 t(n) = 2t(n/2) + C, when n>1, assume n=2^k, t(n) = 2^kt(n/2^k) + (k+1)C That gives us t(n) = O(n)
Best-Case Space Complexity	O(lgn)	Any input pair, such as x=2 and n=31	The function uses at most lgn+c (c is a constant) stack frames on the system stack due to recursion, thus has O(lgn) for space complexity
Worst-Case Space Complexity	O(lgn)	Any input pair, such as x=2 and n=32	Same as the above

```java
public static long pow_2(long x, int n) {
        if (n == 0)
                return 1;
        if (n == 1)
                return x;
        if (n % 2 == 0) {
                return pow_2( x, n / 2 ) * pow_2( x, n / 2 );
        } else {
                return pow_2(x * x, n / 2) * x;
        }
}
```

Submission Note

1) For written part of the questions:
    a) Write your answers inside a text document (in plain text, MS Word, or PDF format)
    b) Name the file as firstname.lastname.assignment1.txt(doc, docx, or pdf) with proper file extension
2) Due Sep 16th, 11:59 PM