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)) | False | True | True |
| f(n)=o(g(n)) | False | False | True |
| f(n)=Ω(g(n)) | True | True | False |
| f(n)=ω(g(n)) | True | False | False |
| f(n)=θ(g(n)) | False | True | False |

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), n2, 1, lg2(n)

1 = O[ln(n)] ln(n) =θ[lg(n)] lg(n)=O[lg2(n)] lg2(n)=O(n)

n=O[nlg(n)] nlg(n)=θ[lg(n!)] lg(n!)=O(n2) n2=O[2(n+1)]

2(n+1)=θ[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 | Example Input | Explanation |
| Best-Case Running Time | O(n) | 1,2,3,4,5,6,7,… | The swap function never needs to be run, so run time increases linearly with array length |
| Worst-Case Running Time | O(n2) | 7,6,5,4,3,2,1,0 | The swap function must move each number down the entire length of the array. Swap function must be performed n times per input, where n = total # of inputs |
| Best-Case Space Complexity | O(n) | 1,2,3,4,5,6,7,… | The size of the array = the total number of inputs. Size increases linearly with # inputs |
| Worst-Case Space Complexity | O(n) | 25,34,23,22,21,… | The size of the array = the total number of inputs. Size increases linearly with # inputs |

**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(1) | x=20, n =1(or 0) | N is small enough, we just return x (or 1) |
| Worst-Case Running Time | O(n) | x=4, n=200 | Each *pow\_2* call halves n, but worst case calls the *pow\_2* function gets called twice within itself. As n🡪0, n function calls are made. |
| Best-Case Space Complexity | O(1) | x=20, n=1(or 0) | N is small enough, we just return x (or 1) |
| Worst-Case Space Complexity | O(n) | x=4, n=4 | We have 2 variables to track, but at most n\*2 [or O(n)] stack frames will be used |

**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:

1. Write your answers inside a text document (in plain text, MS Word, or PDF format)
2. Name the file as firstname.lastname.assignment1.txt(doc, docx, or pdf) with proper file extension

2) Due Sep 16th, 11:59 PM