A
L 1.3/1 b. The algorithm is stable because it uses only the less than operator when comparing the numbers. This effectively eliminates any possibility of counting a recurrence of an equal value multiple times, so the duplicate value will not be moved.
L 1.3/1 c. It is not an in-place sort because everything gets moved to a new array at the end

B
L 1.4/3 a. [a] [ab] [a] [ac] [acd] [ac]
L 1.4/3 b. [a] [ab] [b] [bc] [bcd] [cd]

C
L 2.1/7 a. it will take 8 times as long, because there are 2x as many problems, and 2^3 is 8.
L 2.1/7 b. the factor will increase by 10 because the cube root of 1000 is 10, and as shown in the previous question.

D
L 2.1/8 a. increase by a factor of 2
L 2.1/8 b. increase by a factor of 2
L 2.1/8 c. increase by a factor of 4
L 2.1/8 d. increase by a factor of 16
L 2.1/8 e. increase by a factor of 64
L 2.1/8 f. increase by a factor of 16

E
L 2.1/9 a. same
L 2.1/9 b. lower
L 2.1/9 c. same
L 2.1/9 d. lower
L 2.1/9 e. same
L 2.1/9 f. lower

F
L 2.2/2 a. true
L 2.2/2 b. true
L 2.2/2 c. false
L 2.2/2 d. true

G
L 2.3/1 a. 250,000
L 2.3/1 b. 2046
L 2.3/1 c. n-2
L 2.3/1 d. (n+1)(n+2)/2 - 3
L 2.3/1 e. (n-1)*(n)*(2n-1)/6 + (n-1)*n/2
L 2.3/1 f. 3n * (3^(n+1) - 3) / 2
L 2.3/1 g. ((n) * (n+1) / 2) ^ 2
L 2.3/1 h.

H
L 2.3/5 a. it calculates the range of an array
L 2.3/5 b. the basic operation is comparison
L 2.3/5 c. it operates 2(n-1) times
L 2.3/5 d. O(n)

L 2.3/5 e. the only way to improve the efficiency of the algorithm would be to give it a sorted array. given an array where we don't know whether or not it is sorted, we have to look at all items to get min and max values, so the best possible time efficiency is linear

I
L 2.4/1 a. x(1) = 0, x(2) = 5, x(3) = 10, x(4) = 15 … x(n) = 5*(n-1)
L 2.4/1 b. x(1) = 4, x(2) = 12, x(3) = 36, x(4) = 108 … x(n) = 4*3^(n-1)
L 2.4/1 c. x(0) = 0, x(1) = n, x(2) = 2n, x(3) = 3n, ….x(n) = n^2
L 2.4/1 d. x(1) = 1, x(2) = 1+n, x(4)
L 2.4/1 e. x(1) = 1, x(2) =

J
L 2.4/4 a. x(n) = x(n-1) + 2*n - 1, x(1) = 1
x(1) = 1, x(2) =
L 2.4/4 b.
L 2.4/4 c.