1.
i.
Since rand() always returns a value between 0 and 1, so the best case is rand() = 1
The worst case is rand() < 0.5; and average case would be between 0 and 1.

a. Best case: 1(initialize) +1(increasement) + 2(increasement) = 4 operations
   Worst case: 1(initialize) +1(increasement) +6(increasement) = 8 operations
   Average case: 1(initialize) +1(increasement) +4(increasement) = 6 operations
b. Best case, worst case, average case: Ω (1), O(1), Θ(1)
c. f = O(1)
d. f = Ω(1)
e. f = Θ(1)

ii.
Best case would be N <= 0, so the loop doesn't run, worst and average case would be N>0 with rand() < 0.5

a. Best case: 1 + 1 + 1 = 3 operations
   Worst case: 1 + 1 + (n + 1) + n + n + n + n = 5n+3 operations
   Average case: (5n)/2 + 3
b. Best case: Ω(1)
   Worst case: O(n)
   Average case: Θ(n)
c. f = O(n)
d. f = Ω(1)

iii.
Best case would be N<=0 so the loop doesn't run
Worst case would be N>0 and unlucky == true

a. Best case: 1+1+1 = 3 operations
   Worst case: 1(Initial "count") +1(initial "i") +n(checks N) + 1(check unlucky)+1(assign "j") + (n^2+n)/2(checks) + 2(n^2 +n)/2(increasement) + (n^2 +n)/2(decrement) +n (increment) = 2n^2+4n+7/2 operations
   Average case: n^2 + 2n + 4
b. Best case: Ω(1)
   Worst case: O(n^2)
   Average case: Θ(n^2)
c. f = O(n^2)
d. f = Ω (1)

iv.
Best case would be unlucky == false so the loop doesn't run
Worst case would be lucky == true and N>0 so the loop can run

a. Best case: 1(Init "count) + 1(Init "I") + 1 (check unlucky) = 3 operations
   Worst case: 1(Init "count") + 1 (Init "I") + 1 (check unlucky) + n(check "i") + n(increment for "count") + n(decrement for "i") = 3n+3 operations
   Average case: 3n/n + 3 operations
b. Best case: Ω(1)
   Worst case: O(n)
   Average case: O(n)
c. f = O(n)

d. f = Ω(1)

v.

The best case would be N<=0 so that the count remains 0, the second loop also inactive
The worst case would be N> 0 then the 2 loops can run, rand() < 0.5

a. Best case: 1(init “count”) + 1(init “i”) +1(check “i”) + 1 (init “num”) + 1 (init “j”) + 1 (check “j”) = 6 operations
   Worst case: 1(init “count”) + 1(init “i”) + n (check “i”) + n (init “num”) + n(check “num”) + n(“count” increment) + n(“i” increment) + 1(init “num”) +1 (init “j”) + n(check “j”) + n(“count” increment) + n(“j” increment) = 8n +4 operations
   Average case: 4n+5 operations
b. Best case : Ω(1)
   Worst case: O(n)
   Average case: Θ(n)
c. f=O(n)
d. f = Ω(1)

vi.

the best case would be N<=1 so the loops doesn’t run
the worst case would be N> 1

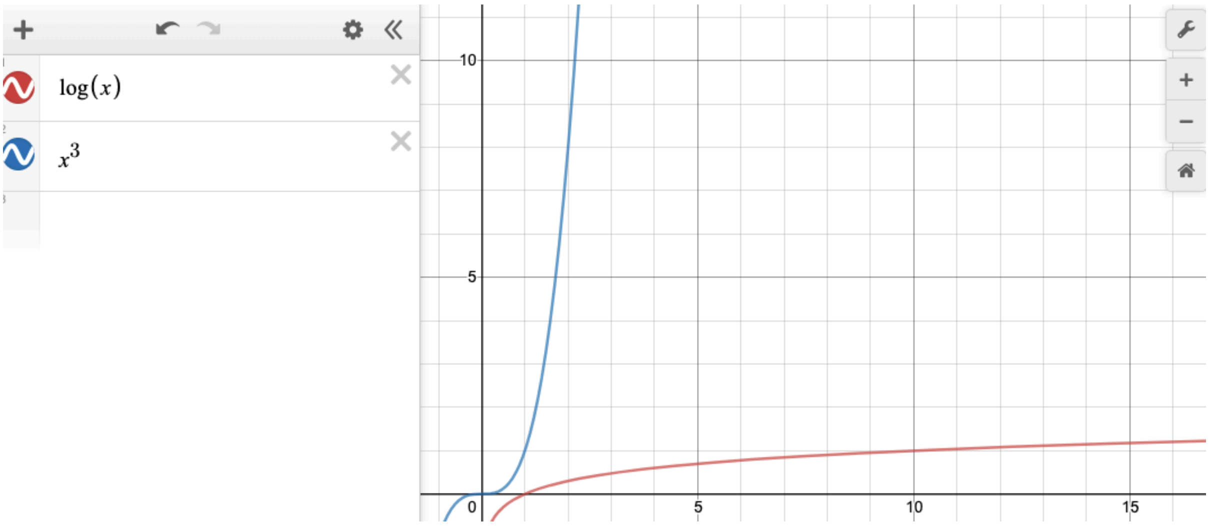
a. Best case: 1(init “i”) + 1(check) = 2 operations
   Worst case: 1(init “i”) + n-1(check) +n-1(init “j”) + n(n+1)/2-2 (check) + n(n+1)/2-1 (check) + n(n+1)/2-2 (swap) + n(n+1)/2-2(increment) n-1(increment) = 2x^2 + 5x -10 operations
   Average: (2n^2 + 5n -8)/2
b. Best case : Ω(1)
   Worst case: O(n^2)
   Average case: Θ(n^2)
c. f= O(n^2)
d. f = Ω(1)

f. The best way to describe the performance of algorithms is using big O notation, because it represent the worst case scenarios, which are regularly occurs during normal usage.

2.

Big O describe the worst-case scenario, can be used to describe the execution time or space used by algorithms.

3. Θ(n^3) always takes longer to run than Θ(log(n)) with every n>=1. Because when n<1, log(n)<0. In common usage, the size of input to algorithms is >0 so Θ(log(n)) will be faster.



4.
True, the highest power of variable n is 2, therefore O(n^2)
True, the highest power of variable n is 1, therefore O(n)
False, as n^4 is not a lower bound to n^3
True, because n is a lower bound for it