## Big O Justification

1. is

True. This equation could be rewritten as , log(n) is greater than 1, thus could be the lower bound of the equation.

1. is

False, since we already proved could be the lower bound of the equation, it could not be the upper bound of the equation.

1. is

True. The Omega function could be rewritten as , and since is the slowest, it times itself 5 times will make it even slower. As n increases, the Omega function will decrease, but will increase as n goes bigger, so the big Omega is the lower bound of the function.

1. is

True. The original function, as n increase by 1, the value should increase by approximately 4 times (speak to the largest term), the big O will increase by n times, thus as long as n is bigger than 4, the big O will be the upper bound of the function on the left hand.

1. is

True. As n gets bigger, the first term gets smaller, log(16) will not change based on n, and it is a constant number bigger than 1, thus 1 could be the lower bound of the equation.

1. is

False. The big O could be turned into , since the n! grows faster than , so big O is the lower bound of the function.

## List Analysis

* Give a tight big-O bound for the running time of insert in ArrayStringList

The tight big-O bound for the running time of insert in ArrayStringList is n. The tight big-O bound for the runtime for the expandCapacity() is n, and for other parts in the method insert() is 3n + 4, which means when they sum together will give the tight big-O bound for the running time of insert() as n.

* Give a tight big-O bound for the running time of insert in LinkedStringList

The worst case for the insert in LinkedStringList is when the index is the length of the arraylist, thus the runtime is 3n + 6, which give the tight big-O bound of n for the running time of insert in LinkedStringList.

* Give a tight big-O bound for the running time of remove in ArrayStringList

The calculated worst case runtime for the remove in ArrayStringList is 3n + 4, thus the tight big-O bound for the running time of remove in ArrayStringList is n.

* Give a tight big-O bound for the running time of remove in LinkedStringList

The worst case for the remove in LinkedStringList is when the index is the length of the arraylist, thus the runtime is 3n + 6, which give the tight big-O bound of n for the running time of remove in LinkedStringList.

* Give a tight big-O bound for the *best case* running time of add in ArrayStringList

The best case running time of add is expandcapacity() will only have a runtime of 1 because it only needs to check for the size of the array once and does not have to expand it, the rest part will be runtime of 2, so the tight big-O bound for the best case running time of add in ArrayStringList is 1.

* Give a tight big-O bound for the *worst case* running time of add in ArrayStringList

The worst case running time of add in ArrayStringList is when the expandCapacity() method does need to be called and has a runtime of n to copy everything in the original arraylist (which has a length of n) to the new arraylist with doubled size. The rest part of add will have a runtime of 2, thus the tight big-O bound for the worst case running time of add in ArrayStringList is n.

* Give a tight big-O bound for the *best case* running time of add in LinkedStringList

The best case of add in LinkedStringList is the while loop only run once, which is the front is actually the only element in the LinkedStringList, which will give a tight big-O bound of 1 for the best case running time of add in LinkedStringList.

* Give a tight big-O bound for the *worst case* running time of add in LinkedStringList

The worst case of add in LinkedStringList is when the while loop run until the end of the LinkedList, which will give a runtime of 2n + 4, and the tight big-O bound for the worst case running time of add in LinkedStringList is n.

## Mystery Functions

public static void f1(int n) {

int a = 0;-----------------1

int i = 0; -----------------1

while (i < n) { -----------------n

a = i; -----------------n

i++;-----------------n

}

}The runtime for f1 is 3n + 2 because the while loop will run for n times, and big-O bound is n. O(n)

public static void f2(int n) {

int a = 0; ----------------- 1

for(int i = 0; i < n; i += 3) {--------1 + n/3 + n/3

a = i; -----------------n/3

}

} The algorithm have a runtime of n + 2, since there is a for loop that will run for n/3 times, give the big-O bound is n. O(n)

public static void f3(int n) {

int a = 0; -----------------1

int i = 0; -----------------1

while (i < n) {-----------------n

for (int j = n; n > 0; n = n/10) {------log(n)

a = i + j; -----------------1

}

i++;-----------------n

}

} f3 has a for loop in a while loop, which the for loop will run for log(n) times and while loop will run for n times, thus the big-O bound for f3 is n\*log(n). O(n\*log(n))

public static void f4(int n) {

int a = 0;

for(int i = 0; i < n; i += 1) {

for(int j = i; j < n; j += 1) {

a = i + j;

}

}

} The algorithms will have a big-O bound for f4 as n^2. The outer loop will run n times, and the inner loop runtime is a arithmetic sequence, which equals to (n^2 + n) / 2, thus the big-O bound for f3 is n^2. O(n^2)

public static void f5(int n) {

int a = 0;

for(int i = 0; i < n \* n; i += 1) {

for(int j = 0; j <= n; j += 1) {

a = i + j;

}

}

} The algorithm will have a big-O bound for f5 as n^3, because the outer loop will run n^2 times, and the inner loop will run n times. O(n^3)

public static void f6(int n) {

int k = 1, a = 0;

for(int i = 0; i < n; i += 1) {

for(int j = 0; j <= k \* 2; j += 1) {

a = i + j;

}

k = k \* 2;

}

} The algorithm will have a big-O bound as 2^n, the outer loop will run n time, but it will not affect the inner loop, the inner loop will run 2^n time because the max for k is 2^n, thus the big-O bound for f6 is 2^n. O(2^n)