**Chapter 2**

1. In the definition of Big-O, why is the "for N >= n0" needed?

When we look over the definition of Big-O, it is that "T(N) = O(f(N)) means that T(N) <= cf(N) for some constant c and for N >= n0. " n0 is the start points where the equation is true from n0 to infinity. For example, there are two algorithms named f and g. f is lower than g before the point n0. However, after from n0 to infinity, f becomes greater than g. n0 is such a point. So, to make sure the algothm analysis, "for N >= n0" is needed.

1. If f1(N) = 2N and f2(N) = 3N, why are they both O(N), since 3N is larger than 2N for N>=1?

f1(N) = 2 \* N and F(N) = N

f1(N) = O(F(N)) 🡪 2\*N <= c \* F(N) for some constant c >= 2 for N >=1

f1(N) = O(F(N)) 🡪 O(N)

f2(N) = 3 \* N and F(N) = N

f2(N) = O(F(N)) 🡪 3\*N <= c \* F(N) for some constant c >= 3 for N >=1

f2(N) = O(F(N)) 🡪 O(N)

So, both of O-notation is O(N), and also, co-efficient is ignored in Big-O

1. a) For f1(N) = 2N and f2(N) = 3N: calculate f1(5) and f2(5), then f1(10) and f2(10). When N was doubled in each case, what happened to the result? Explain why this happens.

f1(5) = 10, f1(10) = 20

f2(5) = 15, f2(10) = 30

When N was doubled, the result is also doubled.

This is becasue both of f1(N) and f2(N) are O(N), So they increase linearly.

b) For f1(N) = 2N\*N and f2(N) = 3N\*N: calculate f1(5) and f2(5), then f1(10) and f2(10). When N was doubled in each case, what happened to the result? Explain why this happens.

f1(5) = 50, f1(10) = 200

f2(5) = 75, f2(10) = 300

When N was doubled, the result is that both of f1(10) and f2(10) is four times of f1(5) and f2(5). It means that grow at a quadratic rate (2^2 = 4). So, it increased quandratic rate and the O notation is O(n^2)

1. Since Big-O notation is a mathematical tool for functions like f(N) or g(N), how is it applicable to algorithm analysis?

Big-O is a mathematical tool, which is defined as Upper-bound(possibly equal) For the algorithm analysis, N is the size of input and it is the main consideration. And O(f(N)) = T (n) is the time taking to run N data. We analyze worst-case performance since that tells us the limit of poor performance.

1. Which grows faster, 2^n or n! ? Explain why.

I think n! is faster than the 2^n.

n! = n \* (n-1) \* (n-2) \* ... \* 5 \* 4 \* 3 \* 2 \* 1

and 2^n = 2 \* 2 \* 2 \* ... \* 2

2^n is 2 multiplied by n times, but n! is multiplied by 1 from n. When the n is less than or equal to 3, 2^n is bigger than n!. However, when I apply 100 to n, 2^100 is 2 multiplied by 100 times and 100! is multiplied by 1 from 100.

So, 2^100 = 2 \* 2 \* 2 \* ... \* 2 --> 100 times

100! = 100 \* 99 \* 98 \* ... \* 1 ---> 100 times

Both of 2^n and 100! is multiplied by components 100 times, but from 100 to 3 (n!) is bigger than 2, so it is natural that n! is bigger than 2^n.

So, n! is faster than 2^n when n >=4

1. Give the Big-O notation for the following expressions:

a. 4n^5 + 3n^2 – 2 🡪 O(n^5)

b. 5^n - n^2 + 19 🡪 O(5^n)

c. (3/5)\*n 🡪 O(n)

d. 3n \* log(n) + 11 🡪 O(nlogn)

e. [n(n+1)/2 + n] / 2 🡪 O(n^2)

1. What is the Big-O running time for this code? Explain your answer.

for (int i=0; i<numItems; i++)

System.out.println(i+1);

O(N) 🡪 This is simple for-loop with numItems(n). Number iterations (n) times the statements inside. The for-loop runs n times. So, the O-notation is O(N)

1. What is the Big-O running time for this code? Explain your answer.

for (int i=0; i<numItems; i++)

for (int j=0; j<numItems; j++)

System.out.println( (i+1) \* (j+1) );

O(N^2) 🡪 This is nested for-loop. Outside for-loop runs numItems(n) times, and the inside for-loop runs numItems(n) times, but they are nested. So, n\*n= n^2. So, the O notation is O(N^2)

1. What is the Big-O running time for this code? Explain your answer.

for (int i=0; i<numItems+1; i++)

for (int j=0; j<2\*numItems; j++)

System.out.println( (i+1) \* (j+1) );

O(N^2) 🡪 This is nested for-loop. Outside for-loop runs numItems(n) times, and the inside for-loop runs 2\*numItems(2n) times, but they are nested. So, n\*2n = 2n^2, but we do not care the coefficients. So, the O-notation is O(N^2)

1. What is the Big-O running time for this code? Explain your answer.

if ( num < numItems )

for (int i=0; i<numItems; i++)

{

System.out.println(i);

}

else

System.out.println("too many");

O(N) 🡪 Under the else statement, O-notation is O(c), and under the if statement, O-notation is O(N) because there is single for-loop, whichi runs n times (numItems) inside the if statement. So, the O-notation is O(N)

1. What is the Big-O running time for this code? Explain your answer.

int i = numItems;

while (i > 0)

i = i / 2; // integer division will eventually reach zero

O(logN) 🡪 if (repeatedly) it takes constant time to reduce the problem size by a fraction. In this case, due to i = i/2, the O-notation is O(logN), and due to while statement, it runs repeatedly. So, the O-notation is O(logN)

1. What is the Big-O running time for this code? Explain your answer.

(You do not need to work out a recurrence formula).

public static int div(int numItems)

{

if (numItems == 0)

return 0;

else

return numItems%2 + div(numItems/2);

}

This is O(logN). The reason is that this is recursion, and it repeatedly runs due to recurstion. It takes constant time to reduce the problem size by a fraction under the else statement. So, the O-notation is O(logN)