**1.** Given two functions, there is usually one point where one function is smaller than the other function. That point is n0.

When we plot the function over a graph, after the point n0, the bigger function is always above the smaller function.

N>n0 helps us determine which function is bigger.

O(n) is typically concerned with the behavior of functions for large values of n. It is mainly concerned about what happens to the function as n approaches infinity.

**2.** As the value of N approaches infinity, there is not much difference between the values of f1(N) and f2(N) and the definition of big O includes f1(N) >= f2(N). Hence,big O considers coefficients to be insignificant and both are bound by O(N).

**3.** a) f1(N)=2N f2(N)=3N

f1(5)=10 f2(5)=15

f1(10)=20 f2(10)=30

When N is doubled, the result is also doubled.

This happens because f1 (N) and f2 (N) are O (N) bound and it grows at a linear rate as f1(N) and f2(N) are linear functions.

b) f1(N)=2N\*N f2(N)=3N\*N

f1(5)=50 f2(5)=75

f1(10)=200 f2(10)=300

When N is doubled, the result increased fourfold.

This is because f1(N) and f2(N) are O(N^2) bound and it grows at a quadratic rate as f1(N) and f2(N) are quadratic functions.

**4.** N is the size of the input.

f(N) is the estimate of the amount of time it takes the function to output an answer for a size of N.

O(f(N)) is considered to be better than O(g(N)) if O(f(N)) grows slower.

Big O is mainly used to estimate the run time and instead of trying to find the exact run time we can use the Big O to give us a rough estimate.

For a certain constant less than n0, O(g(N)) might work better for smaller sizes of input.

Hence, we need Big O in algorithm analysis to know with certainty which function works best for large input sizes.

**5.** Let’s assume a few values for n.

For n=0, 2^n is 1 , n! is also 1.

For n=5, 2^n is 32 , n! is 120.

For n=10, 2^n is 1024 , n! is 3628800.

As we go to values over n=1000,2^n will only be multiplied by another 2 but n! will be multiplied with a number over 1000. Hence,n! grows faster.

**6.** a. O(n^5)

b. O(5^n)

c. O(n)

d. O(nlogn)

e. O(n^2)

**7.** Printing i+1 has the complexity of O(1),the loop runs N times.

The running time of the code = O(1) \* N.

The big O running time is O(N).

**8**. Printing ((i+1) \* (j+1)) has a complexity of O(1).

Size of the outer loop is N and the size of the inner loop is also N.

Running time of the code = O(1)\*N\*N.

The big O running time is O(N^2).

**9**. Size of the outer loop = N+1

Size of the inner loop=2\*N

Size of the print statement= O(1)

Running time of the code = O(1)\*2N\*(N+1)

We ignore the constants – 2 in 2\*N and 1 in N+1.

The big O running time for the code is O(N^2).

**10**. To calculate the big O running time for if-else statements we only take into account the larger of the two branches. Hence, we will only consider the if loop.

If(num<numItems) has run time of O(1).

Size of for loop is n.

Run time of print statement is O(1).

Run time of the code = O(1) \* O(N) \* O(1)

The big O running time of O(N).

**11**. The while loop halves the result each time it enters the loop. Since the loop is being reduces by a constant fraction in a constant amount of time, the big O running time is O(logN).

**12**. We will consider the larger of the two branches i.e. the else branch.

Whenever the control enters the else branch, it will make another call after it has been divided by 2. Since it is being halved by constant time after each call, it is O(logN).