Lesson 38: Complexity Analysis (Big O)

Two types of complexity analysis:

Complexity analysis takes two forms. One form analyzes **how many steps** an algorithm takes to run… ultimately this means the **time that it takes to run**. The other type of complexity analysis has to do with how much space (in bytes) it takes to run the algorithm. With memory so abundant and inexpensive today, space analysis is not as important as it once was. We will confine our studies here to the time-analysis variety.

Introducing Big O:

Consider the following algorithm:

```
for(int j = 0; j < n; j++)
{…some code, no loops }
```

How many times do we go through the loop? The answer is *n*, of course, so we say that the time complexity is **of the order of n**. A short-hand equivalent is written as **O(n)**. This is known as **Big O** notation. Another valuable way to look at this is that the time it takes to run this algorithm is approximately proportional to *n*. The larger *n* is, the better the approximation. This is true of all Big O values.

A variety of Big O problems:

Now, let's look at some other specific examples and obtain Big O values for each.

Example 1:

```
for(j = 0; j < n; j++)
{
        for(k = 0; k < n; k++)
        { …some code… }
}
```

We go through the outer loop n times and for each of these iterations we go through the inner loop n times. The code designated as "…some code…" is executed n2 times so we assign a Big O value of **O(n^2)** to this algorithm.

Example 2:

```
for(j = 0; j < (n +50); j++)
{
        for(k = 0; k < n; k++)
        { …some code… }
}
```

We go through the outer loop n + 50 times and for each of these iterations we go through the inner loop n times. The code designated as "…some code…" is therefore executed $(n + 50)n = n^2 + 50n$ times. Recalling that Big O notation is an approximation for only very large n, we realize that 50n pales in comparison to n^2 so we **keep only the part with the highest exponent**. Finally, we assign a Big O value of **O(n^2)** to this algorithm.

Example 3:

```
for(j = 0; j < n ; j++)
{
        for(k = 0; k < (22 * n); k++)
        { …some code… }
}
```

We go through the outer loop *n* times and for each of these iterations we go through the inner loop 22n times. The code designated as "…some code…" is therefore executed n(22n) = $22n^2$ times. In Big O notation we ignore all coefficients, no matter their size, and assign a Big O value of **O(n^2)** to this algorithm.

Example 4:

Suppose by a time complexity analysis, we obtain $(3/5)n^3 + 15n^2 + (1/2)n$ $+5$, the corresponding Big O value would just simply be **O(n^3)**, since we **ignore all coefficients and use only the term with the highest exponent**.

Example 5:

```
public static int[] addStuff(int x[][])
{
        int row, col;
        int b[] = new int[x.length];
        for(row =0; row < x.length; row++)
        {
                for(col= 0; col < x[row].length; col++)
                {
                        b[row] += x[row][col];
                }
        }
        return b;
}
```

Yes, this one is a bit more complicated than the previous ones. Let's assume we call this method with the following code:

```
int dfg[][] = { {1,2,…},
                {0,4,…},
                … };      //Assume this array has r rows and c columns for a
int newArray[] = addStuff(dfg);        //total of rc = n elements
```

Studying the *addStuff* method, we note that we go through the outer loop *x.length* times which is the number of rows, *r*. For the inner loop we go through *x[row].length* which is the number of columns, *c*. Therefore, the total number of times through the code in the inner loop is *rc*, which in turn is just the number of elements in the entire matrix, *n*. We can write the Big O value as either **O(rc)** or **O(n)**.

Example 6:

```
for(j = 0; j < n; j+=5)
for(k =1; k < n; k*=3)
{ …some code… }
```

We go through the outer loop (1 / 5)n times and log3(n) times through the inner loop for a total of (1 / 5)n log3(n). The final Big O value is **O(n log(n))**. Notice that we have dropped the coefficient of 1 / 5 as is the custom with Big O. Also, we have dropped the base 3 on the log since all logs of the same quantity (but with various bases) differ from each other only by a mulplicative constant.

Example 7:

```
m = -9;
for(j = 0; j < n; j+=5)
{
        … some code …
        if (j < m)
        {
                for(k =0; k < n; k*=3)
                {… some code…}
        }
}
```

We go through the outer loop n/5 times and because *j* will never be less than *m*, we never go through the inner loop. Thus the Big O value is **O(n)**.

Example 8:

```
for(j = 0; j < n; j++)
{
for(k = j; k < n; k++)
{ …some code… }
}
```

On the first iteration of the outer loop, the inner loop iterates *n* times. On the second iteration of the outer loop, the inner loop iterates n – 1 times. On the third iteration of the outer loop, the inner loop iterates n - 2 times. This continues until the last iteration of the outer loop results in n – (n –1) iterations of the inner loop. Adding these we get

$$n + (n – 1) + (n – 2) + \dots + (n – (n –1))$$

Since the outer loop iterated *n* times we know there are *n* terms here. Simplifying, we get n(n –1) + constant = $n^2 – n$ + constant. The Big O value is therefore, **O(n^2)**.

Example 9:

Consider a **sequential search** (sometimes called a **linear search**) through an unordered list looking for a particular item. On the average we will need to search halfway through before we find the item. So, on the average, the Big O value should be O(n /2). Again, we drop the coefficient and get **O(n)**.

Example 10:

Consider a **binary search** on an ordered list. In fact, a binary search can only be done on an **ordered** list since this type of search works as follows. In the beginning we go halfway down the ordered list and ask, "Is this the item? If not, is the item above or below this point?" Then we take that indicated half of the list and cut it in half. The process repeats until we eventually find the item. Since we repeatedly cut the list by a factor of two, the run time is proportional to $\log_2(n)$ when n is the number of elements in the original list. We drop the base and write O(log(n)).

Example 11:

When we have just a simple block of code with no repetition, the Big O value is O(1). Consider the following block of code that yields a Big O value of **O(1)**:

```
x = 3 * Math.pow(p,2.1);
y = 46.1 * q/2.3;
d = Math.sqrt(Math.pow((x –x1), 2) + Math.pow((y –x1), 2));
```

Example 12:

This example spotlights the pitfall of excessive dependence on shortcuts. Consider the Big O value for the following code:

```
public int doStuff(int n)
{
        int sum = 0;
        for(int j = n; j > 0; j /= 2)
        {
                for(int k = 0; k < j; k++)
                {
                        sum += (2 * k * j) % n;
                }
        }
}
```

At first glance, the outer loop would seem to yield O($\log_2$(n)) according to shortcuts presented earlier. The inner loop is a bit of a mystery; however, most students would probably guess O(n). Putting these together and ignoring the base of the logarithm would yield a final value of O(n log(n)). Unfortunately, this is not the answer.

Let's go back to basics and forget about the shortcuts. The problem with the above analysis is the inner loop. To properly analyze this code, look at the first iteration of the outer loop and determine how many iterations there are of the inner loop. There are *n* iterations:

n

On the second iteration of the outer loop *j = n/2*, so there are *n/2* iterations of the inner loop:

n + n/2

On each successive iteration of the outer loop, *j* is half of its value on the previous iteration (and hence the number of iterations of the inner loop), so we have:

n + n/2 + n/4 + n/8 + … + 1

Searching for a simpler way to express this sum, we notice that this is a geometric series whose sum is given by the well known formula, (firstTerm – r * lastTerm)/(1 – r) where r is the common ratio (½ for this problem). So, our sum reduces to (n – ½ * 1)/(1 – ½) which, in turn algebraically simplifies to 2n – 1.

As per the conventions of time complexity analysis, drop the -1 and the coefficient of 2 to get a final value of O(n).

Calculating run times:

Example 1:

Suppose a certain algorithm has a Big O value of $O(n^2)$ and that when $n = 1000$ the run time is 19 sec. What will be he run time if $n = 10,000$?

We set up a proportion as follows:

$1000^2 /19 = 10000^2 / T$

$(10^3)^2 / 19 = (10^4)^2 / T$

$10^6 / 19 = 10^8 / T$, cross multiply to get
$T(10^6) = 19(10^8)$

$T = 1900$

Example 2:

Sometimes questions are worded such that we are looking for a ratio as in this problem. …Suppose a certain algorithm has a Big O value of $O(n^3)$. How many times slower is this algorithm when $n = 1500$ as compared to $n = 500$?

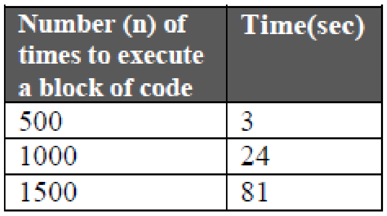
We set up a proportion as follows:

$500^3 / T_1 = 1500+ / T_2$, cross multiply and rearrange things so as to solve for the ratio T_2 / T_1

$T_2 / T_1 = 1500^3 / 500^3$
$T_2 / T_1 = (1500 / 500)^3$
$T_2 / T_1 = (3)^3$
$T_2 / T_1 = 27$, so the answer is that it's 27 times slower.

Example 3:

Consider the following table in which the number of times that a block of code executes is contrasted with the time it takes to run.



Number (n) of times to execute a block of code	Time(sec)
500	3
1000	24
1500	81

Table 38-1

What Big O value is represented by the data in this table?

When *n* doubles (going from 500 to 1000) the time is multiplied by 8. Since 2^3 is eight we conclude that the Big O value should be n^3. This is consistent with a comparison of the first and last rows of the table above. There, we notice that in tripling *n* when moving from an *n* value of 500 to 1500, that the time is multiplied by 27 (3 * 27 = 81). Since 3^3 is 27, we are, once again, led to n^3.

From fastest to slowest:

In summary, we will state what should be obvious by now. We will list in order, Big O values with the most efficient time-wise (fastest) at the top and the slowest at the bottom.

O(log n)
O(n)
O(n log n)
O(n^2)
O(n^3)
O(2^n)

Using graphs to compare Big O values:

Let's compare just two of the Big O values above so we can get a sense of why one is better than the other. In Fig. 38-1 below we will compare O(log n) and O(n).

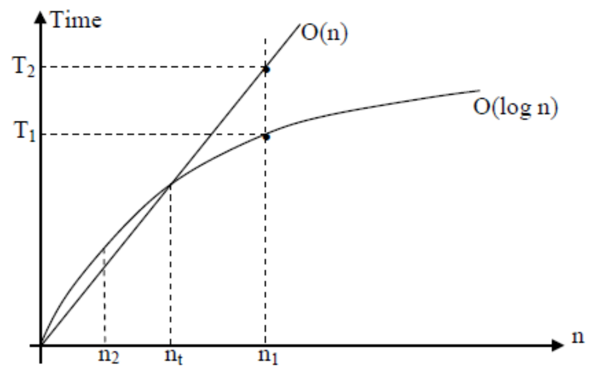


Fig 38-1 Comparison of O(n) and O(log n)

For a particular value of *n* (*n1*, for example) notice that O(log n) gives a corresponding run time of T_1 while n_1 corresponds to T_2 on the O(n) curve. A smaller run time is desired, so the O(log n) curve is superior for n_1 (and all values of *n* higher than n_t).

Beware of false statements:

When comparing Big O values, we must be careful in assuming something like the following:

An algorithm with a Big O value of O(log n) will **always** be faster than one with a Big O value of O(n).

This is generally a **false** statement. For **some** values of *n* (large ones) O(log n) **will**, indeed, be faster than O(n); however, we do not generally know this "transition" n_t value. For Fig 38-1 above this "transition" n_t value occurs at the intersection point of the curves. You would need to know specific details of each to know which is faster for any particular value of *n*. It is interesting to note that for the *n* value of n_2 in the drawing above, that the O(n) curve actually represents the fastest time.

The moral of all this is that Big O values are generally only valid for **large values of *n*** and are mostly useful for **relative** comparisons (For example, using O(n^2) to determine by what factor the run time increases when going from n = 10^3 to n = 10^5).

Best case, average case, worst case:

Some algorithms, especially sorting routines discussed in Lesson 40, have special circumstances in which they perform very poorly and sometimes very well. These are assigned "worst case" and "best case" Big O values.

Occasionally, "best case" is referred to as the **most restrictive** or **fastest executing** case. Similarly, "worst case" is referred to as the **least restrictive** or **slowest executing** case.