Asymptotic Analysis

Asymptotic Analysis is a core topic in Datastructures and Algorithms. It is the primary reason why we have so many datastructures to choose from. All datastructures can store data, so why can't we just use Arrays for everything? Although any program can produce the same correct output using any datastructure, the choice of datastructure greatly impacts the performance. So much so, that an operation that might be efficient with an appropriate Datastructure could take a very very very long time to run with a poorly choosen Datastructure.

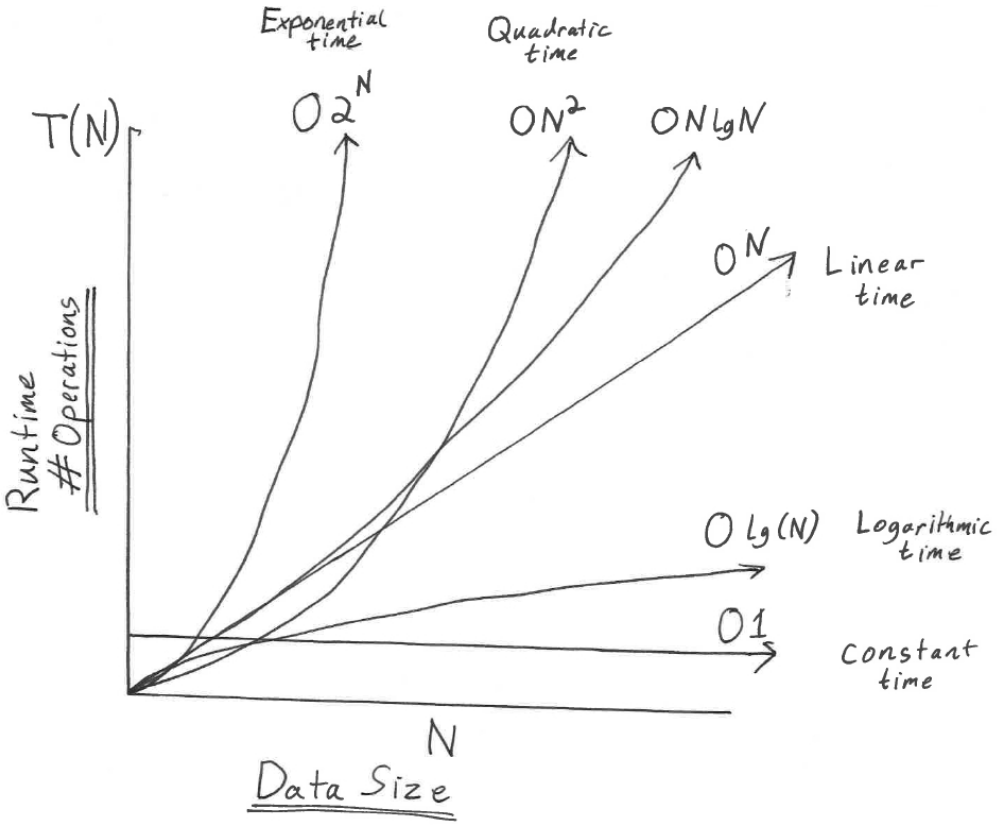
The time it takes a program to run, or **Runtime** ***T(N)*** is not necessarily linearly proportional to the size of the dataset ***N***. In fact, depending on the algorithm and/or datastructure.

Runtime ***T(N)*** could be any function of dataset size ***N***

Some common functions are shown below:

T(N) is **O(1)**	constant time
T(N) is **O(lg N)**	logarithmic time
T(N) is **O(N)**	linear time
T(N) is **O(N lg N)**	N lg N time
T(N) is **O(N^2)**	quadratic time
T(N) is **O(N^K)**	polynomial time
T(N) is **O(2^N)**	exponential time

We can see a plot of these functions in the following graph



Notice the Big-Oh used to describe each of these functions. The Big-Oh indicates (loosely) that the **constant factors** don't matter when we describe the function. A bit more precisely, the **T(N)** is **O(X)** if **T(N)** is at worst proportional to **X** the. We can **discard constant** factors when writing functions in Big-Oh notation.

So if a subroutine has runtime **T(N)** operations for a dataset size of **N**, then the following are all O(N) a.k.a. linear time:

(a) if T(N) = N then T(N) is O(N)
(b) if T(N) = 2N then T(N) is O(N)
(c) if T(N) = 0.1 N + 123 then T(N) is O(N)
(d) if T(N) = 1000 N + lg N + 4 then T(N) is O(N)

Clearly subroutine (b) is twice as slow as subroutine (a), but we would say that these are both O(N) or linear time.

The Precise definition of Big O is the following

$$T(N) \text{ is } O\,F(N) \quad \text{if and only if}$$

$$\forall N > N_0 \quad \exists k > 0 \quad \text{s. t.} \quad T(N) \leq k\,F(N)$$

Or in words:
T(N) is O F(N) if and only if
for all sufficiently large N, there exists some constant k, such that T(N) is less than k times F(N)

Example:
Prove T(N) = $3N^2 + 2N + 7$ is O N^2

Proof:
for all N > 1, $2N < 2N^2$ and $7 < 7N^2$

thus $3N^2 + 2N + 7 < 3N^2 + 2N^2 + 7N^2$ (for all N>1)

thus $3N^2 + 2N + 7 < 12\,N^2$ (for all N>1)

therefore: $\forall N > 1 \quad T(N) \leq 12\,N^2$ so T(N) is O N^2

If you encounter code that uses loops, we may *"multiply"* the code within the loops together to calculate the Big O

Example:

```
int A[N]
int B[N]
int C[N]
```

```
for i=1...N
        for j=1...N
                for k=1...N
                        print A[i] + B[i] + C[i]
```

The above example has a runtime of ON^3 because each of the nested loops is O(N)

If you have the following function:

```
int SelectionSort (int *A, int N)
{
        for (int i=0; i<N; i++) {

                for (int j=i; j<N; j++) {

                        if (A[j] < A[i]) {
                                int temp = A[i] ;
                                A[i] = A[j] ;
                                A[j] = temp ;
                        }
                }
        }
}
```

Notice that we cannot simply multiply the two loops together, because the inner loop does not run the exact same number of times every iteration.

Instead, when i = 0, it runs for N times.
When i = 1 it runs for N-1 times.
When i = 2, it runs for N-2 times,
When i = 3 it runs for N-3 times
etc.

So all in all, we need to represent it by a summation

$T(N) = N + (N-1) + (N-2) + (N-3) + \ldots + 1$

Which we can write in reverse order

$T(N) = 1 + 2 + 3 + \ldots + (N-2) + (N-1) + N$

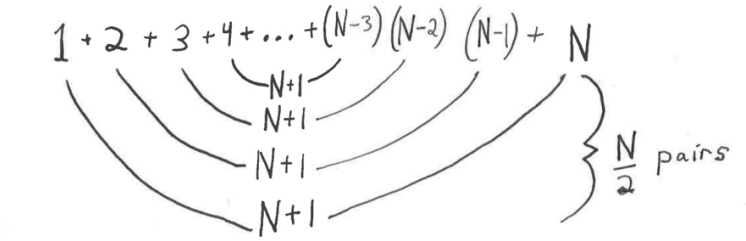
Which we can reformulate in terms of a summation

$$T(N) = \sum_{i=0}^{N-1} i$$

This particular summation is called the “arithmetic sum” the solution to the arithmetic sum is

$$T(N)=\frac{N}{2}(N+1) \quad \text{or} \quad T(N)=\frac{1}{2}N^2+\frac{1}{2}N$$

Why? Because there are N / 2 pairs, each of value N+1



Once we have T(N) in terms of a formula

$$T(N)=\frac{1}{2}N^2+\frac{1}{2}N$$

It is clear that T(N) is O N^2 we could prove this if we want by using the definition of Big O