Constant Time Algorithms – O(1)

```
int n = 1000;
System.out.println("Output: " + n);
```

Clearly, it doesn't matter what n is. It takes a constant amount of time to run. It's not dependent on the size of n.

```
int n = 1000;
System.out.println("Output: " + n);
System.out.println("Next output: " + n);
System.out.println("One more output " + n);
```

The above example is also constant time.
Even if it takes 3 times as long to run, it doesn't depend on the size of the input, n. We denote constant time algorithms as follows: O(1). Note that O(2), O(3) or even O(1000) would mean the same.

We don't care about exactly how long it takes to run, only that it takes constant time.

Logarithmic Time Algorithms – O(log n)

Constant time algorithms are the quickest. Logarithmic time is the next quickest.

One common example of a logarithmic time algorithm is the binary search algorithm.

What is important here is that the running time grows in proportion to the logarithm of the input (in this case, log to the base 2):

```
for (int i = 1; i < n; i = i * 2){
    System.out.println("Output: " + i);
}
```

If n is 8, the output will be the following:

```
Output: 1
Output: 2
Output: 4
```

Linear Time Algorithms – O(n)

After logarithmic time algorithms, we get the next fastest: linear time algorithms.

It grows directly proportional to the size of its inputs.

```
for (int i = 0; i < n; i++) {
    System.out.println("Output: " + i);
}
```

N Log N Time Algorithms – O(n log n)

The running time grows in proportion to n log n of the input:

```
for (int i = 1; i < n; i++){
    for(int j = 1; j < n; j = j * 2) {
        System.out.println("Output: " + i + " and " + j);
    }
}
```

If n is 8, then this algorithm will run 8 * log(8) = 8 * 3 = 24 times.

Polynomial Time Algorithms – *O*(n^p)

These algorithms are even slower than n log n algorithms.

The term polynomial is a general term which contains quadratic *(n^2)*, cubic *(n^3)*, quartic *(n^4)*, etc. functions.

What's important to know is that ***O(n^2)*** is faster than ***O(n^3)*** which is faster than ***O(n^4)***, etc.

```
for (int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        System.out.println("Output: " + i + " and " + j);
    }
}
```

This algorithm will run $8^2 = 64$ times. Note, if we were to nest another for loop, this would become an *O(n^3)* algorithm.

Exponential Time Algorithms – *O*(k^n)

These algorithms grow in proportion to some factor exponentiated by the input size.

For example, *O(2^n)* algorithms double with every additional input. So, if *n = 2*, these algorithms will run four times; if n = 3, they will run eight times (kind of like the opposite of logarithmic time algorithms).

O(3^n) algorithms triple with every additional input, O(k^n) algorithms will get k times bigger with every additional input.

```
for (int i = 1; i <= Math.pow(2, n); i++){
    System.out.println("Output: " + i);
}
```

If n is 8 it will run 2^8 = 256 times.

Factorial Time Algorithms – O(n!)

```
for (int i = 1; i <= factorial(n); i++){
    System.out.println("Output: " + i);
}
```

where factorial(n) simply calculates n!. If n is 8, this algorithm will run 8! = 40320 times.

Functions ordered by growth rate:

Function	**Name**
1	**Growth is constant**
logn	**Growth is logarithmic**
n	**Growth is linear**
nlogn	**Growth is n-log-n**
n^2	**Growth is quadratic**
n^3	**Growth is cubic**
2^n	**Growth is exponential**
n!	**Growth is factorial**

- **$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$**

- **To get a feel for how the various functions grow with n, you are advised to study the following figs:**

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313 x 10^{33}