**The Big O Notation**

The “O” in “Big O Notation” stands for ‘order’ as in the order of a function. When taking into account the O notation of a given function you consider it’s worst case outcome:

let v = vec![5, 6, 7, 8, 10, 1233] ;

for i in 0..v.len() {  
 if I == 5 { break }

}

The above would have a runtime complexity of O(n). For every element (n) in the vector, regardless that this case the loop will end on the first iteration, is how we are to consider the functions complexity. It will always be the maximum number of iterations a loop could possibly hold.

When considering runtime complexity for single instance items like:

let x = 10;

5 + 12;

match { x => , y => }

x == y;

This would have a runtime of O(1) which means no matter the input the runtime complexity is to be assumed as instantaneous. When single instance items occur within a loop then they are treated as multipliers per iteration:

for \_ in 0..10 { let x = 10; } // O(1 \* n)

for \_ in 0..10 { let x = 10; let y = x; } // O(2 \* n)

for \_ in 0..10 { let x == 10; let y = x; let z = 12 } // O(3 \* n)

When loops are nested within one another, loop within a loop, then the runtime grows exponentially:

for i in 0..10 {  
 for j in 0..10 {  
 let x = 10;

}

}

The above having a Big O of O(n²)

When dealing with Recursive functions they depending the runtime is based on the number of potential branching paths raised to the number of possible recursive iterations:

let mut x = 10;

loop\_back(&mut x);

fn loop\_back(x: i32) {  
 if !(x <= 0) {

x -= 1;

loop\_back(&mut x);

} else {

do\_something(x);

}

}

The above having a Big O of O(2^n) for each possible branch x can/will feed into.

**Complexity Classes**

|  |  |
| --- | --- |
| O(1) | Constant Time, which means everything will take the same amount of time. |
| O(log(n)) | Growth is defined by the logarithmic function (in general, base 2), which is better than linear growth. |
| O(n) | Linear time, which means that the solution performance depends on the input in a linear way. |
| O(n log(n)) | This is sometimes called quasilinear time and is the best achievable complexity for sorting. |
| O(n²) | The squared runtime is typical for the native implementation of search or sorting algorithms. |
| O(2^n) | This is among the most expensive classes and can often be found in really hard-to-solve problems. |

Most naïve/prototype solutions will have a O(n) runtime complexity as the goal is getting the program to work with optimization a future concern but it helps to know what you’ll be targeting for improvements when the time comes with this chart.

It is important to know the space your program will run on when determining which runtime is acceptable. If you have very little space for overhead an O(n²) may be better than an O(log(n)) solution because of the overhead required to implement the faster option. You should gauge the likely maximum of data processed and storage on hand you’re program will be expected to work with when determining which Big O notation is acceptable for gauging your algorithms.