Analysis of Algorithms

An Analogy

Imagine the following scenario: You got a file on a hard drive and you need to send it to a friend who live across the country. You need to get the file to your friend as fast as possible. How should you send it ?

Most people's first thought would be email, FTP, or some other means of electronic transfer. That thought is reasonable but only half correct. If it's a small file, you're right, it would take 5-10 hours to get to an airport, hop on a flight, and then deliver it to your friend. But what if the file were really, really large ? Is it possible that it's faster to physically deliver it via plane ?

A one-terabyte (1 TB) file could take more than a day to transfer electronically. It would be much faster to just fly it across the country. If your file is that urgent, you might just want to do that.

We could describe the above data transfer "algorithm" runtime as:

- **Electronic Transfer:** O(s) where s is the size of the file. This means that the time to transfer the file increases linearly with the size of file.
- **Airplane Transfer:** O(1) with respect to the size of the file. As the size of the file increases, it won't take any longer to get the file to your friend.

Intuitive Explanation

$O(N^2)$ means for every element, you're doing something with every other element, such as comparing them. Bubble sort is an example of this.

*O(N*log(N))* means for every element, you're doing something that only needs to look at log N of the elements. This is usually because you know something about the elements that let you make an efficient choice. Most efficient sorts are an example of this, such as merge sort.

O(N!) means to do something for all possible permutations of the N elements. Traveling salesman is an example of this, where there are N! ways to visit the nodes, and the brute force solution is to look at the total cost of every possible permutation to find the optimal one.

clarification: Actually, Big-O says nothing about comparative performance of different algorithms at the same specific size point, but rather about comparative performance of the same algorithm at different size points:

What is an algorithm

An algorithm is a set of instructions to find the solution to a problem. It gives step-by-step operations to be performed that will take you from any valid input for the problem to an output.

Rules for Evaluating Complexity

How can you predict the complexity of a given algorithm? We can look for certain features to help us characterize it.

- Think of a name (often `n`) for the size of the input. If you have multiple inputs, like `arr1`, `arr2`, assign different names for each one (size of `arr1` is `n`; size of `arr2` is `m`).
- For consecutive statements, add the complexities of each.
- For branching statements (`if/else`), use the complexity of the worse branch.
- For loops, multiply the maximum number of times the loop can run by the complexity of the work inside the loop.
- Simplify: eliminate constant multiples within parentheses (`O(2n)` → `O(n)`), constant multiples of a single big-o family (`8*O(n)` → `O(n)`), and entire smaller terms (`O(n) + 3*O(1)` → `O(n)`). Don't remove smaller terms that use a different name for the input size: `O(n) + O(log(m))` doesn't simplify.

Common Big O Values

Input Size (N)	O(1)	O(log(N))	O(N)	O(N*log(N))	O(N^2)
1	1	0	1	1	1
8	1	3	8	24	64
30	1	~5	30	150	900
500	1	~9	500	4500	250,000
1000	1	~10	1000	10,000	1,000,000
16,000	1	~14	16,000	224,000	256,000,000
100,000	1	~17	100,000	1,700,000	10,000,000,000

Constant — statement (one line of code)

```
a += 1;
```

Growth Rate: **1**

Logarithmic — divide in half (binary search)

```
while (n > 1) {
  n = n / 2;
}
```

Growth Rate: **log(n)**

Linear — loop

```
for (int i = 0; i < n; i++) {
  // statements
  a += 1;
}
```

Growth Rate: **n**

The loop executes `N` times, so the sequence of statements also executes `N` times. If we assume the statements are `O(1)`, the total time for the for loop is `N * O(1)`, which is `O(N)` overall.

Quadratic — Effective sorting algorithms

```
Merge-sort, Quicksort, …
```

Growth Rate: **n*log(n)**

Quadratic — double loop (nested loops)

```
for (int c = 0; c < n; c++) {
  for (int i = 0; i < n; i++) {
    // sequence of statements
    a += 1;
  }
}
```

Growth Rate: **n^2**

The outer loop executes N times. Every time the outer loop executes, the inner loop executes `M` times. As a result, the statements in the inner loop execute a total of `N * M` times. Thus, the

complexity is `O(N * M)`.
In a common special case where the stopping condition of the inner loop is `J < N` instead of `J < M` (i.e., the inner loop also executes `N` times), the total complexity for the two loops is `O(N^2)`.

Cubic — triple loop

```
for (c = 0; c < n; c++) {
  for (i = 0; i < n; i++) {
    for (x = 0; x < n; x++) {
      a += 1;
    }
  }
}
```

Growth Rate: **n^3**

Exponential — O(2^n)

An O(2^n) algorithm requires double the resources for each additional input. This is an example of exponential growth - and it gets out of hand very quickly.

```
function fibonacci(num) {
  if (num <= 1){
    return num;
  } else {
    return fibonacci(num - 1) + fibonacci (num - 2);
  }
}
```

Growth Rate: **2^n**

If-Then-Else

```
if (condition) {
  block 1 (sequence of statements)
} else {
  block 2 (sequence of statements)
}
```

If `block 1` takes `O(1)` and `block 2` takes `O(N)`, the `if-then-else` statement would be `O(N)`.

Statements with function / procedure calls

When a statement involves a function/ procedure call, the complexity of the statement includes the complexity of the function/ procedure. Assume that you know that function/procedure `f` takes constant time, and that function/procedure `g` takes time proportional to (linear in) the value of its parameter `k`. Then the statements below have the time complexities indicated.

`f(k)` has `O(1)`
`g(k)` has `O(k)`

When a loop is involved, the same rule applies. For example:

```
for J in 1 .. N loop
  g(J);
end loop;
```

has complexity `(N2)`. The loop executes N times and each function/procedure call `g(N)` is complexity `O(N)`.

> Note: Those constant multiples can get really large, meaning sometimes an O(n) algorithm will run faster than an O(log (n)) algorithm. Hence depending on input there can be such cases.

Topic online resources

- OneNote Notes link