Notes - Algorithms & Data Structures - Big O Notation - Intro

- Dr Nick Hayward

A brief intro to Big O notation for algorithms.

Contents

Intro

Big O is special notation we may use to test and define the comparative performance of an algorithm.

For example, we commonly use this notation to test the performance of a third party algorithm. We may then compare and contrast various algorithms relative to project requirements.

Running time for algorithms

The first option for timing algorithms is simple search. In effect, 100 items has a potential maximum number of guesses of 100.

If we increase this number exponentially, the potential maximum time will continue to grow at the same rate. So, 4 billion items may take 4 billion guesses to reach the end of the list. This is known as *linear time*.

However, if we compare this performance with binary search, for example, we quickly see the performance benefits.

For example, for a list of 100 items we require at most 7 guesses. Larger datasets see a marked improvement in performance. 4 billion results will now require a maximum of 32 guesses.

So, we have a comparative result

- `O(n)` for linear time
- `O (log n)` for logarithmic time

A practical example

An example of choosing between simple and binary search.

n.b. this may seem like an obvious choice, but there may be contexts where *linear time* may be acceptable.

In many examples, we need an algorithm that is both fast and correct.

e.g. Landing on Mars...

We need to quickly choose an algorithm, usually in 10 seconds or less, to allow a spaceshup to land on Mars.

For this test, binary search will be quicker for most tests. However, simple search is easier to write, and may reduce errors due to its inherent simplicity.

As we're performing mission critical tasks, we can't have any bugs.

So, we begin by running each algorithm 100 times. Each task may take 1 millisecond to execute. If we run initial tests, we get the following results

- simple search = 100 ms (100 x 1ms)
- binary search = 7 ms ($\log_2$ 100 = 7)

100 ms vs 7 ms.

Whilst the real-world usage difference is minimal, the actual program will likely require a billion plus tasks and executions.

We may perform a quick initial scaling of timings, e.g.

- binary search = ~30 ms ($\log_2$ 1,000,000,000)

Binary search was initially ~15 times faster, so simple search will scale to `30 x 15`.

This seems reasonable, and is within tolerances for the program.

However, there's a major issue with this cursory calculation. It's based on an assumption that both search algorithms grow at the same rate.

Run times grow at different rates, thereby impacting performance relative to each dataset.

If we consider this specific example, Big O notation shows us that binary search is closer to 33 million times faster than simple search.

So, we cannot use simple search for our Mars lander.

Big O usage

Big O notation tells us the relative operations for each algorithm and dataset. In effect, how fast the algorithm is per task.

Big O notation defines binary search, for example, as `O log(n)`).

visualising big O

We may start with various algorithms to draw a grid of 16 squares.

We're effectively trying to determine the best algorithm to use. This decision is often predicated on many disparate conditions, which may reflect priorities in a given project.

For example, we may consider the following algorithm

- draw each box until we have the required 16 boxes...

- big O notation tells us that a linear pattern will produce 16 grid squares
- one box is one operation...

Big O notation produces the followibg results,

```
O(n)
```

However, there are better and more efficient options for algorithms.

For example, a second algorithm option uses folding to optimise the create of squares in the grid.

If we start by folding the a large square, a piece of paper for example, we immediately create two boxes. In effect, each fold is an operation in the algorithm.

We may continue folding the square, piece of paper, until we create our grid of 16 squares. This oly takes four operations to complete.

This algorithm produces a performance of `O(log n)` time.

common Big 0 runtimes

As we use various algorithms for projects, we may consider common performance times as calculated using Big O notation.

For example.

big O notation	description	algorithm
O(n)	also known as linear time	simple search
O(log n)	also known as log time	binary search
O(n * log n)	a faster sorting algorithm	e.g. quicksort
$0(n^{2)}$	a slow sorting algorithm	e.g. selection sort
O(n!)	a very slow algorithm	e.g traveling salesman problem

So, if we applied each algorithm to the above creation of a grid of 16 squares, we could choose the appropriate algorithm to solve the defined problem.

n.b. this is a tad simplified representation of Big O notation to the number of required operations.

We may consider the runtime and performance of an algorithm as follows,

- algorithm speed is measured using the growth in the number of required operations, and not time in seconds
- we consider the speed of increase in the runtime for an algorithm as the size of the input increase
- runtime for algorithms is defined using Big O notation
- `O(log n)` is faster than `O n`, and continues to get faster as the list of search items continues to increase

Traveling Salesman problem

An algorithm with a renowned bad running time.

This has become a famous problem in Computer Science, and many believe it will be very difficult to improve its performance.

The problem is as follows,

- the salesman has to visit 5 cities
- salesman wants to visit all cities in the minimum distance possible

Commonly, we might simply review each possible route and order to and from the cities.

For 5 cities, there are 120 permutations. Then, this will scale as possible

cities	permutations
6	720
7	5040
8	40320
15	1307674368000
30	265252859812191058636308480000000

For `n` items, it will take `n!` (n factorial) operations to compute the result.

This is what is known as *factorial time* or O(n!) time. So, as soon as the number of cities passes 100, we do not have enough time to calculate the number of permutations. The sun is forecast to collapse sooner!