Complexities and Searching

Complexities

Understanding and analysing the complexity of an algorithm allows us to select the best algorithm for the job.

What is an Algorithm?

An algorithm is a step-by-step procedure for solving a problem. The properties of an algorithm are:

- Each step of an algorithm must be exact.
- An algorithm must terminate.
- An algorithm must be effective.
- An algorithm should be general.

Analysis of Algorithms

The analysis of algorithms is the analysis of the efficiency of a program, and this provides us with a metric to compare different methods of solution. In this case, the efficiency is equated to the complexity.

A comparison of algorithms should focus on significant differences in the efficiency of the algorithms, and should not consider reductions in computing costs due to clever coding tricks. Tricks may reduce the readability of an algorithm.

The goal is to evaluate rigorously the resources (time and space) needed by an algorithm and represent the result of the analysis with a formula. We will emphasize more on the time requirement rather than space requirement, and the time requirement of an algorithm is also called its time complexity.

Big-O Notation

What is the best way to compare algorithms' time complexity?

1. Measure Run Time?

Will not work, as the run time depends on the compiler, the computer used, and the current work load of the computer. We also want to be able to compare two algorithms that are coded in different languages, using different data sets or running on different computers.

2. Count the Number of Operations

The time taken by an algorithm is related to the number of operations required. We can simplify this analysis further by ignoring the different types of operations and different number of operations in a statement, and simply count the number of statements required.

We also only need a simple term to indicate how efficient an algorithm is. The dominating term of a formula gives us the approximate performance. This is called **asymptotic analysis** of the algorithm.

Asymptotic analysis is an analysis of algorithms that focuses on:

- Analysing the problems of large input size
- Considering only the leading term of the formula
- Ignoring the coefficient of the leading term

This is where the Big-O notation comes in.

Definition of the Big-O Notation
Given a function f(n), we say g(n) is an (asymptotic) upper bound of f(n), denoted as $f(n) = O(g(n))$, if there exist a constant $c > 0$, and a positive integer n_0 such that $f(n) \leq c*g(n)$ for all $n \geq n_0$.

- f(*n*) is said to be bounded from above by g(*n*).
- O() is called the "big O" notation.

Based on the definition, $2n^2$ and $30n^2$ have the same upper bound n^2. This is why:

$2n^2$
$f1(n) = 2n^2$; $g(n) = n^2$
Let $c=3$ and $n_0=1$, since $2n^2 \leq cn^2 \; \forall \; n \geq n_0$
Hence f1(n) = O(g(n))

$30n^2$
$f2(n) = 30n^2$; $g(n) = n^2$
Let $c = 31$ and $n_0 = 1$, since $30n^2 \leq cn^2 \; \forall \; n \geq n_0$
Hence f2(n) = O(g(n))

They only differ in terms of c.

With this definition, any bound that's "bounds" the algorithm is technically a valid complexity. However, we are usually only interested in the tightest bound.

These are the common complexities:
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n2) < O(n3) < O(2n) < O(n!)$

How to find the complexity?
Basically just count the number of statements executed.

- If there are only a small number of simple statements in a program – O(1)
- If there is a 'for' loop dictated by a loop index that goes up to n – O(n)
- If there is a nested 'for' loop with outer one controlled by n and the inner one controlled by m – O(n*m)
- For a loop with a range of values n, and each iteration reduces the range by a fixed constant fraction (eg: 1⁄2)
    - O(log n)
- For a recursive method, each call is usually O(1). So
    - if n calls are made – O(n)
    - if n log n calls are made – O(n log n)

Common Examples:
Non-recursive Binary Search: O(log n)

Recursive Fibonacci: $O(2^2)$ or more precisely $O(\Phi^2)$

Worst Case vs Best Case vs Average Case

Worst-Case Analysis

- Interested in the worst-case behaviour.
- A determination of the maximum amount of time that an algorithm requires to solve problems of size n

Best-Case Analysis

- Interested in the best-case behaviour
- Not useful

Average-Case Analysis

- A determination of the average amount of time that an algorithm requires to solve problems of size n
- Have to know the probability distribution
- The hardest

Examples from Tutorial

	$O(\log [(\log n)^2]) = O(\log \log n) <$		
Logarithmic Time	$O(\log_3 n) = O(\log n) <$		
Sublinear Time	$O(n^{0.00000001}) <$	$O(100\ n^{2/3}) <$	
Linear Time	$O(20n) <$		
Linearithmic Time	$O(n \log n) = O(\log n!) <$		
Quadratic Time	$O(4n2) <$		
Polynomial Time	$O(n^{2.5}) <$		
Exponential Time	$O(2^n) = O(2^{n+1}) <$	$O(3^n) <$	$O(2^{2n}) = O(4^n) <$
Factorial Time	$O((n-1)!) <$	$O(n!) <$	
!!!	$O(n^n)$		

Mathematical Equations
Prepared by Aaron Tan, Edited by Chong Ket Fah

Here are some equalities that are useful in analyzing algorithms:

Arithmetic series

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \quad \ldots (1)$$

More generally, if $a_n = a_{n-1} + c$, where *c* is a constant, then

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n = \frac{n(a_n + a_1)}{2} \quad \ldots (2)$$

Geometric series

$$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 \quad \ldots (3)$$

More generally, if $a_n = ca_{n-1}$, where $c \neq 1$ is a constant, then

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n = a_1 \frac{c^n - 1}{c - 1} \quad \ldots (4)$$

If $0 < c < 1$, then the sum of the infinite geometric series is

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1 - c} \quad \ldots (5)$$

Harmonic series

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(i+1) \quad \ldots (6)$$

Sum of squares

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \quad \ldots (7)$$

Logarithms

$$\log_b a = \frac{1}{\log_a b} \quad \ldots (8)$$

$$\log_a x = \frac{\log_b x}{\log_b a} \quad \ldots (9)$$

$$\log_2(n!) = n \log_2(n) \quad \ldots (10)$$

Searching

Sequential Search vs Binary Search

- Sequential Search is the linear iteration of some data to find the data required
- Binary Search is the searching of sorted data in a way that halves the amount of data we are searching through with every step, until the item is found

Binary Search

- By having items in increasing order, we can first look at the middle item, and determine if the item that we want is in the first half or the second half
- We repeat this process with the half that we want, such that we continuously narrow down the range we are searching through
- This repeats until we actually find the item that we want as the middle item, or when the range of items to search next becomes 0 i.e. item not found

Efficiency of Sequential Search (data not sorted)

Worst case: O(*n*) – when item is the last item or unsuccessful search
Average case: O(*n*) – item averages out to be around n/2 position
Best case: O(1) – when the first item is the one we need

Efficiency of Binary Search (data sorted)

Worst case: O(log n) – when item is found during the last step or unsuccessful search
Average case: O(log n)
Best case: O(1) – meaningless as the chance of this happening is very low

O(1) Searching?

There is also the concept of hashing, where we hash (key, value) pairs into a HashTable. This will be covered subsequently in Data Structures, but this allows us to achieve near O(1) searching and retrieving.