Algorithms and Imperative Programming

COMP26120

Todd Davies

December 21, 2014

Introduction

This is a two-semester practical introduction to algorithms and data structures, concentrating on devising and using algorithms, including algorithm design and performance issues as well as 'algorithmic literacy' - knowing what algorithms are available and how and when to use them.

To reflect the emphasis on practical issues, there are two practical (laboratory) hours to each lectured hour. Lectures serve to motivate the subject, orient students, reflect on practical exercises and impart some basic information. A range of practical applications of algorithms will also be presented in the lectures. Other information resources will be important, including a set textbook, which will provide essential support.

The course-unit starts with a 5-week primer on the C programming language, enabling students to become competent programmers in this language as well as in Java (and, possibly, in other languages). This teaching is supported by an on- line C course and extensive laboratory exercises.

There is a follow-up course unit on Advanced Algorithms in the Third Year. This presents the foundational areas of the subject, including (1) measures of algorithmic performance and the classification of computational tasks by the performance of algorithms, (2) formulating and presenting correctness arguments, as well as (3) a range of advanced algorithms, their structure and applications.

Aims

- To make best use of available learning time by encouraging active learning and by transmitting information in the most effective ways.
- To give students a genuine experience of C.
- To make students aware of the importance of algorithmic concerns in real-life Computer Science situations.
- To emphasise practical concerns, rather than mathematical analysis.
- To become confident with a range of data structures and algorithms and able to apply them in realistic tasks.

Additional reading

Algorithm design: foundations, analysis and internet examples - Goodrich, Michael T. and Roberto Tamassia

Contents

1 Algorithmic complexity and performance

Algorithmic complexity and the big-oh notation allows us to characterise the time and space requirements of an algorithm when it is given varying input data. The big-oh notation allows us to get a good approximation of the upper and lower bounds of an algorithm's complexity.

We can work out such an approximation by analysing (and generalising) the number of logical operations an algorithm might do, rather than inspecting it's performance in an implementation. This allows us to compare the merits of different algorithms irrespective of their implementation.

The big-oh notation is shown below:

$$O(growthrate)$$

The growth rate represents the rate at which the complexity of the algorithm will change with the size of the input.

Growth rates that are either exponential or factorial in nature (or are perhaps even worse than this) are said to be intractable , while algorithms with other computational complexities are said to be tractable.

Tractable (*Adjective*)
Easy to deal with.

Complexity	Growth rate	
$O(1)$	None	
$O(logn)$	Logarithmic	
$O(n^k)$	Polynomial	
$O(n)$	Linear	All of these are special cases
$O(n^2)$	Quadratic	of polynomials, n^1, n^2 and n^3
$O(n^3)$	Cubic	respectively
$O(k^n)$	Exponential	
$O(n!)$	Factorial	

Table 1: A number of common complexities and their equivalent growth rates

1.1 Simplifying Big-Oh expressions

In order to simplify the big-oh complexity of an algorithm you just isolate the fastest growing term in the equation (i.e. whatever term comes furthest down in Table 1). You then remove all the constants from the equation.

1.2 Analysing algorithmic complexity

There are two ways of finding the complexity of an algorithm, to inspect the psudo code for it, or by implementing the algorithm and experimentally determining the change in it's runtime with different input sizes.

Remember to check that your psudo code correctly implements the algorithm before you try this.

In order to analyse the psudo code to work out the complexity, you must look at how many primitive operations it will use for different sizes of input. Primitive operations are defined as memory accesses, arithmetic operations, comparisons and the like.

As a general rule, loops, recursion and other constructs for repeatedly performing operations will the best indicator as to the complexity of algorithms.

If an algorithm is reducing the data it has to work with every so often, then it may have a logarithmic runtime. For example, the algorithm utilises a binary chop (such as binary search), then the runtime probably has a log_2 inside. See section 1.2.1 for a bit more on logs.

In order to determine complexity experimentally, you must implement the algorithm in a programming language of your choice, then run the program for different input sizes and measure the runtime and the memory used. Plot the data on a graph and extrapolate as needed. From the curve of the graph, it is possible to predict the complexity of the algorithm.

1.2.1 A refresher on logs

A logarithm of a number is the exponent to which another number (the base) must be raised to produce that number:

$$\forall b, x, y \in \mathbb{Z}$$
$$y = b^x \Leftrightarrow x = log_b(y)$$

Henceforth, $2^4 = 16$ and $log_2(16) = 4$.

1.2.2 Finding the maximum input size

If we know the complexity and running space/time of an algorithm for a specific implementation and input, we might want to know the input size we could run the algorithm with for the specific machine in a specific time, or within a specific space limit.

The way you do this is by solving the big-oh equation for t instead of n. For example, if the algorithm takes 30 seconds to process 1000 kilobytes of data, how long will it take if we double the processing speed, given that the algorithm runs in $O(n^3)$ time?

$$t = n^3$$
$$\sqrt[3]{t} = n$$
$$\text{When we double } n \text{ and } t\text{:}$$
$$2n = \sqrt[3]{2t}$$
$$2n = 1.25992104989t$$
$$30/1.25992104989 \approx 24$$

1.3 The master theorem

See pages 268-270 in the course textbook for more information

The master method is a way of solving divide and conquer recurrence equations without having to explicitly use induction. It is used when an algorithm's complexity is of the form:

$$T(n) = \begin{cases} c & \text{if } n \leq d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Where $f(n)$ is a function that is positive when $n \geq d$ and:

$d \geq 1$
$a > 0$
$b > 1$
$c > 0$
$d \in \mathbb{Z}$
$a, b, c \in \mathbb{R}$

Such a recurrence relation occurs whenever an algorithm uses a divide and conquer approach. Such an algorithm will split the problem into a subproblems, each of size n/b before recursively solving them and merging the result back together. In this case, $f(n)$ is the time it takes to split the problem into subproblems and merge them back together after the solving is done.

The master theorem is defined by three cases:

1. If there is a small constant $\epsilon > 0$ such that $f(n)$ is $O(n^{log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{log_b a})$

   An example could be when the recurrence relation is

   $$T(n) = 4T(n/2) + n$$

   since $n^{log_b a} = n^{log_2 4} = n^2$, therefore $T(n) = \Theta(n^{log_2 4}) = \Theta(n^2)$.

2. Work out what's going on...

2 Algorithmic correctness

3 Data structures

4 Basic algorithms

4.1 Sorting

A sorting algorithm takes as an input, an array of keys, where there is a *total order* on the keys (each key can be compared to another), and produces as an output, the array where the keys are ordered according to their order.

A total order is a relation that is transitive ($a \leq b \wedge b \leq c \implies a \leq c$), anti-symmetric ($a \leq b \wedge b \leq a \implies a = b$), and unsuprisingly, total ($a \leq b \wedge b \leq a$).

If the ordering on the elements of the array isn't a total order then bad things can happen when you try and use an algorithm to sort the data. For example, if you were to try and sort an array of rocks, papers and scissors according to the rules of the traditional game, then you wouldn't have a transitive relation, and a sorting algorithm would probably loop infinately.

4.1.1 Define sorting

4.1.2 Quicksort

4.1.3 Merge sort

4.1.4 Radix sort and Bucket sort

4.1.5 Heap sort

4.1.6 The lower bound of sorting

4.2 Searching

4.3 Tree and graph traversal