Algorithms and Imperative Programming

COMP26120

Todd Davies

December 29, 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?

$$t = n^3$$
$$\sqrt[3]{t} = n$$

When we double n and t:

$$2n = \sqrt[3]{2t}$$
$$2n = 1.25992104989t$$
$$30/1.25992104989 \approx 24$$

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:

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

Where $f(n)$ is a function that is positive when $n \ge d$ 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 unsurprisingly, 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 infinitely.

4.2 The complexity of sorting

The most common sorting algorithms are $O(n^2)$ time (this includes $O(n \log n)$ algorithms too). Even though a polynomial sort time is good, sometimes we have billions of items to sort, and henceforth, a very long running time. In this situation, $n \log n$ sorts are much more preferable to n^2 sorts.

Since we often don't know in advance exactly what data any sorting function will be given in advance, its important to know both the upper and lower bounds on the complexity of the sorting algorithm we're using.

The **upper bound** is the worst case time complexity of the sort. For example the worst case complexity of Merge Sort is $O(n \log n)$.

The **lower bound** complexity is always at least $O(n)$ for sorting algorithms, since every item needs to be looked at once (if only to check that the list is already sorted). Bucket, radix and bubble sort all achieve this for certain inputs. No comparison based sort can ever achieve a lower bound with less than $O(log_2(n!))$ comparisons.

4.2.1 Worst case for comparison sorts

If we were to draw a decision tree for each possible path of a comparison sort, then we would get one with a depth of $log_2(n!)$, that produced all $n!$ permutations of the input. An asymptotically optimal comparison sort must travel down this tree in its quest to find the answer. MergeSort and HeapSort are optimal, while QuickSort is optimal for most inputs.

4.3 Sorting algorithms in detail

QuickSort

QuickSort is a **divide and conquer** algorithm, that uses a **comparison based** method to sort items. QuickSort can be implemented as an **in-place** sort, **stable**, though it is usually done so with recursion which gives it a space complexity of at least $O(log(n))$. This is a small inconvenience really though.

First the list is partitioned into two halves. To do this, a random pivot is chosen from the list, and of the two new lists, one has the items less than the pivot, and the other has the items greater or equal to it.

QuickSort is then applied to each sublist, so make them sorted, and then the start of the right list is joined to the end of the left list.

See Listing 1 for an example implementation.

MergeSort

MergeSort is another **divide and conquer** algorithm that also uses a **comparison based** approach. MergeSort can **not be implemented in place**, but it is **stable**.

It is similar to quicksort, except it's worst case run time is $O(n \log n)$. First the list is split down the middle, and the two halves are sorted using merge sort recursively. Then, the two lists are merged back into one sorted list in $O(n)$ time. See Listing 2 for an implementation.

BucketSort

BucketSort is a **stable**, **distribution based** sorting algorithm. You place elements into 'buckets' that describe a class of objects (you could order people by the first letter of their first name for example). When you've done that, you can either sort each bucket (using a different algorithm), then you simply return each bucket in order.

BucketSort has a complexity of $O(n + k)$ where k is the number of buckets you have. In the person name example, the number of buckets would be 26, which, depending on the size of n could be a very large, or very small factor.

RadixSort

A RadixSort is basically bucket sort with multiple iterations. When you have sorted the first digit/word etc, you apply the same sort to each bucket in turn, until all of the buckets have size 1. RadixSort has a time complexity of $O(n)$

RadixSort can be done **in-place**, and in a **stable** manner:

```
050, 731, 806, 014, 235
050, 731, 014, 235, 806
806, 014, 731, 235, 050
014, 050, 235, 731, 806
```

HeapSort

HeapSort is very simple, it iterates through the unsorted list, adding each element to a heap as it goes. When it has finished adding all the elements, it simply iterates through the heap and returns the order of iteration. Since adding an element to a (binary) heap is an $O(log(n))$ operation, and you need to do it n times, the runtime of HeapSort is $O(n \log n)$.

HeapSort can be implemented **in place** since the heap can be stored inside the input array (since a heap can be represented as an array). HeapSort isn't stable though, since the heap can be re-ordered while it's being created (if you're not sure on this, maybe it's time for a trip to wikipedia to learn about heaps).

4.4 Searching

4.5 Tree and graph traversal

5 Code listings

```java
import java.util.Arrays;
import java.util.*;

public class QuickSort {


  private static <T extends Comparable<T>> void swap(T[] list, int i1,
                                                      int i2) {
    T temp = list[i1];
    list[i1] = list[i2];
    list[i2] = temp;
  }

  public static <T extends Comparable<T>> void quickSort(T[] list) {
    quickSort(list, 0, list.length - 1);
  }

  public static <T extends Comparable<T>> void quickSort(T[] list, int start,
                                                          int end) {
    if(start >= end) {
     return; // 0 or 1 elements
    } else {
      int midPoint = start + ((end - start) / 2);
      T pivot = list[midPoint];
      int swapIndex = start;
      swap(list, midPoint, end);
      for(int i = start; i < end; i++) {
        if(pivot.compareTo(list[i]) > 0) {
          swap(list, i, swapIndex++);
        }
      }
      swap(list, end, swapIndex);
      quickSort(list, start, swapIndex - 1);
      quickSort(list, swapIndex + 1, end);
    }
  }


  public static void main(String[] args) {
    Integer[] testData = {4,3,2,6,7,56,4,3,7,8,6,4,23,56,6,4,23,2,2,1,6,5,34,8};
    System.out.println(Arrays.toString(testData));
    quickSort(testData);
    System.out.println(Arrays.toString(testData));
  }

}
```

Listing 1: QuickSort

```java
import java.util.Arrays;

public class MergeSort {

  private static int[] mergeSort(int[] input) {
    return mergeSort(input, 0, input.length - 1);
  }

  private static int[] mergeSort(int[] input, int start, int end) {
    if((end - start) == 0) {
      return Arrays.copyOfRange(input, start, end + 1);
    } else {
      int middle = start + ((end - start) / 2);
      int[] firstHalf = mergeSort(input, start, middle);
      int[] secondHalf = mergeSort(input, middle + 1, end);
      int[] output = new int[firstHalf.length + secondHalf.length];
      int l = 0, r = 0, o = 0;
      while(l < firstHalf.length || r < secondHalf.length) {
        if(l < firstHalf.length && r < secondHalf.length)  {
          if(firstHalf[l] < secondHalf[r]) {
            output[o++] = firstHalf[l++];
          } else {
            output[o++] = secondHalf[r++];
          }
        } else if(l < firstHalf.length) {
          output[o++] = firstHalf[l++];
        } else {
          output[o++] = secondHalf[r++];
        }
      }
      return output;
    }
  }

  public static void main(String[] args) {
    int[] toSort = {10,9,8,7,6,5,4,3,2,1,0};
    System.out.println(Arrays.toString(toSort));
    int[] sorted = mergeSort(toSort);
    System.out.println(Arrays.toString(sorted));
  }
}
```

Listing 2: MergeSort