Algorithms and Imperative Programming

COMP26120

Todd Davies

January 4, 2015

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 of polynomials, n^1, n^2 and n^3 respectively
$O(n^2)$	Quadratic	
$O(n^3)$	Cubic	
$O(k^n)$	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:

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

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...

1.4 Amortized Analysis

> Amortized analysis is a technique for analysing an algorithm's running time. It is often appropriate when one is interested in understanding asymptotic behaviour over sequences of operations. For example, one might be interested in reasoning about the running time for an arbitrary operation to insert an item into a binary search tree structure. In cases such as this, it might be straightforward to come up with an upper bound by, say, finding the worst possible time required for any operation, then multiplying this by the number of operations in the sequence. However, many real data structures, such as splay trees, have the property that it is impossible for every operation in a sequence to take the worstcase time, so this approach can result in a horribly pessimistic bound! With a bit of clever reasoning about properties of the problem and data structures involved, amortized analysis allows a tighter bound that better reflects performance.
>
> (Rebecca Fiebrink, Amortized Analysis Explained, Princeton University)

The idea is based on the fact that some high cost operations may do lots of the work for later operations, which means that these later operations are low cost. In order to encapsulate this in our analysis, we need to come up with a 'potential' function that captures how operations with different costs affect our datastructure.

1.4.1 Aggregate analysis

This is the most simple method of amortized analysis. You find a bound on the sequence of operations, and then divide it by the number of operations to find the amortized cost. This only works when all operations have the same cost, such as pushing and popping items with a stack. If we want to push/pop n items, then the total cost would be n (assuming is costs 1 for each push or pop), so the amortized cost would be $\frac{n}{n} = 1$.

1.4.2 Banker's method

This method takes the view that each operation has a predefined cost that must be paid in order to execute it. You would need to insert coins into the computer for each operation based on it's cost. The cost does not necessarily equate to the complexity of the operation, in which case, any difference will be added to (or subtracted from) a 'bank account'.

This means that, operations that take a long time (say re-balancing a tree, or an insert into a heap that requires a lot of re-organisation), will be able to take money from the bank account in addition to their own fee in order to 'pay' for their time.

In this way, **the amount charged for each operation is the amortized cost for that operation**.

The hard bit with this method is to pick appropriate costs for each operation and show that they are sufficient to allow payment for any sequence of valid operations. If c_i is the actual cost of the ith operation, C is the cost we charge per operation, and we do n operations, then the following must hold for us to always have a positive bank account:

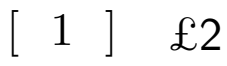
$$\sum_{i=1}^{n} C \geq \sum_{i=1}^{n} c_i$$

In order to reason about never falling into debt with a negative bank balance, it's a good idea to keep a track of which operations generated what credit, and store it in different 'sections' of the bank account. This could mean that each position in a stack ends up with having a certain amount of credit associated with it, or each node in a tree.

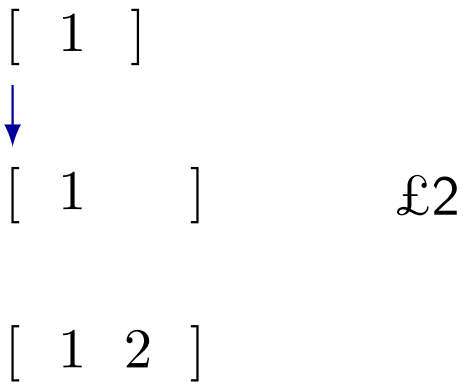
Lets analyse an extendable array. When it has reached it's limit, we create a new array of double the length and copy the current one across.

If we set a cost of 3 for every array write, then this should cover the cost of copying when the size limit is reached. Note, that on 'normal' writes (i.e. where we don't have to resize the array), we will only add 3 to the bank since £1 is taken up for the write operation. If we start with an empty array of size 0:

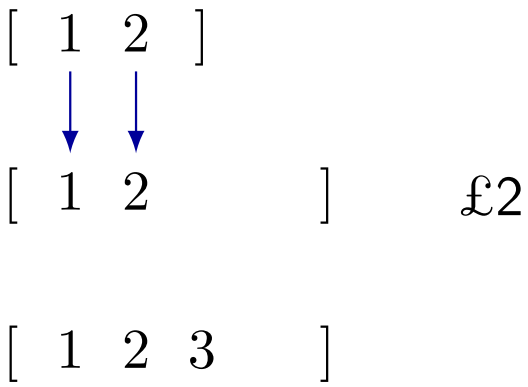
Now, lets add 1:



Now, in order to add 2, we'll need to resize the array, which costs £1 for copying the current item over, and then add the new 2 in, so we'll use up £2:

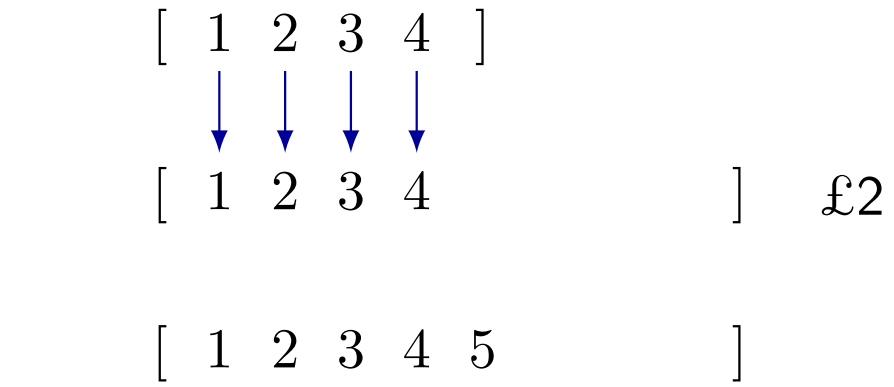


Now, in order to add 3, we'll need to resize the array, which costs £2 for copying the current ones over, and then add the new 3 in, so we'll use up £3:

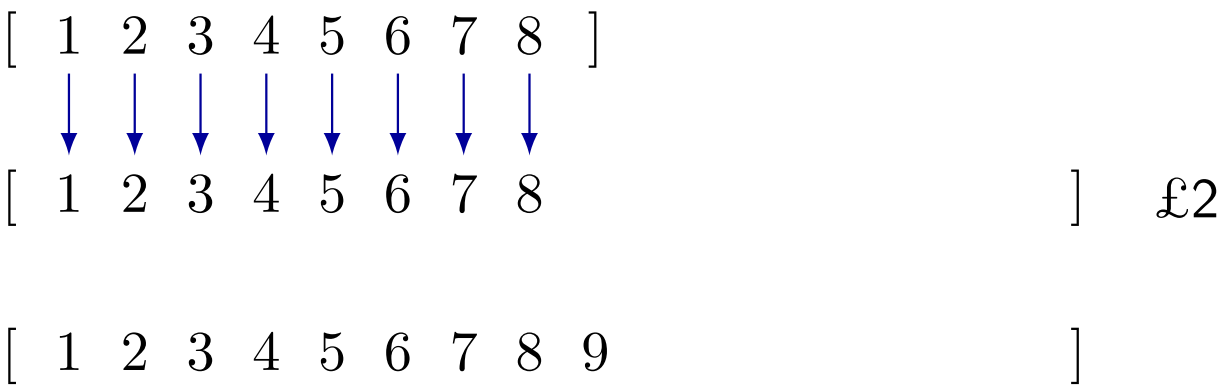


Adding 4 is normal:

But to add 5, we need another resize, costing $4 + 1 = 5$ (we also get an income of £3 too though):



Lets whizz forward to inserting 9. By this point, we'll have £8. We need to expand £8 doing copying, and another adding 9, but we get an income of £3, so we're left with £2 again:



You can see here that we are able to maintain a resizable array by charging a flat rate of £3 on every insert. This means we can pay for the insertion of n elements into our array with $3n$, which means our algorithm runs in $O(n)$ amortized time.

1.4.3 Potential method

I think the idea of 'potential' comes from potential energy in physics, which can be defined as "The energy stored in a system due to it's configuration".

The potential method is similar to the banker's method in that it requires virtual money for each operation and it keeps a bank account (of potential in this case) with which to finance expensive operations, however, **in the potential method, the amortized cost of the operation i is equal to the actual cost plus the increase in potential due to the operation**.

Mathematically, if we define Φ to be the potential function, which takes as an argument, the datastructure D at a specific state after the ith operation, and c_i to be the cost of operation i, then the cost of the operation will be:

$$C = c_i + (\Phi(D_i) - \Phi(D_{i-1})$$

We can derive the following equation (I won't try to here):

$$\sum_{i=1}^{n} C = \sum_{i=1}^{n} (c_i) + \Phi(D_n) - \Phi(D_0)$$

If $\Phi(D_i) \geq \Phi(D_0)$ for all i, then we know we will never run out of potential.

Lets say we have a stack structure, that implements `push` to push one element, and `popNPush`, which pops k elements, and pushes another. Our potential function could be:

$$\Phi(D_i) = \textit{The number of items in the stack after operation } i$$

This means that `push` will always have a cost of:

$$\begin{aligned} C_{\texttt{push}} &= 1 + (\Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + \text{\#items} + \text{\#items - 1} \\ &= 1 + 1 \\ &= 2 \end{aligned}$$

Lets work out the cost for `popNPush`:

$$\begin{aligned} C_{\texttt{popNPush(k)}} &= (k+1) + (n-k+1) - n \\ &= 2 + k - k + n - n \\ &= 2 \end{aligned}$$

We can conclude that the total amortized cost for both `push` and `popNPush` is 2, so they have a amortized running time of $O(1)$.

1.4.4 Comparing the banker's and the potential methods

The banker method focuses on each operations prepayment by charging each operation a fixed sum according to it's type, which will offset future expensive operations made possible by this operation. The potential method focuses on the current operation, and how it affects the datastructure/system now, and the corresponding change in opportunity for future operations.

Often, the banker method will store credit throughout the datastructure/system, maybe assigning each stack frame some credit, or maybe putting it in the branches of a tree. In contrast, the potential method takes a more holistic view and analyses the datastructure/system as a whole, maybe asking, how deep is the stack, or what is the depth of the tree.

It is harder to come up with a decent cost function Φ in the potential method than it is to work out the cost of operations in the banker method, but for the potential method, you only have to do it once, whilst in the banker method, you have to do it for each possible operation.

2 Basic algorithms

2.1 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.

2.2 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.

2.2.1 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.

2.3 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).

2.4 Searching

3 Code listings

```
1  import java.util.Arrays;
2  import java.util.*;
3
4  public class QuickSort {
5
6
7    private static <T extends Comparable<T>> void swap(T[] list, int i1,
8                                                       int i2) {
9      T temp = list[i1];
10     list[i1] = list[i2];
11     list[i2] = temp;
12   }
13
```

```
  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

```
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;
    }
  }
```

```
34
35    public static void main(String[] args) {
36      int[] toSort = {10,9,8,7,6,5,4,3,2,1,0};
37      System.out.println(Arrays.toString(toSort));
38      int[] sorted = mergeSort(toSort);
39      System.out.println(Arrays.toString(sorted));
40    }
41  }
```

Listing 2: MergeSort