Introductory lecture

David Rydeheard
Room 2.111
david.rydeheard@manchester.ac.uk
and
Milan Mihajlovic
and
Ian Pratt-Hartmann

Course Arrangements

- Course Aims 1: A practical introduction to computational thinking and developing algorithmic literacy.

  Two semesters: 1 lecture and 1 laboratory session every week.
- Course Aims 2: Introducing a further programming language - the language C.
- Course website is full of information: arrangements, resources, links to online courses/material including your laboratory exercises. You should all consult it as soon as possible:

  http://studentnet.cs.manchester.ac.uk/ugt/COMP26120/
- Course textbook - essential for the course:

  Algorithm design: foundations, examples and internet examples. Michael Goodrich and Roberto Tamassia. ISBN 0471383951, Wiley (2002).

  This book is now online from Blackboard and the course website

A computational problem

A computational problem:

Consider a list of positive integers. We are given a positive integer k and wish to find two (not necessarily distinct) numbers, m and n, in the list whose product is k, i.e. $m \times n = k$.

For example, $k = 72$, and the list is

$$[5, 24, 9, 5, 30, 6, 3, 12, 2, 10].$$

A computational problem

How do we do this in general, i.e. for all finite lists of integers?

We need an algorithm for this computational task.

An algorithm is a mechanical procedure which we can implement as a program.

DO IT!

How many algorithms can you suggest? What is the best performing algorithm?

Approaches to developing algorithms

There are many algorithmic techniques available.

For this problem, here are some possibilities:

- We may search the list directly.
- We may try to preprocess the list and then search.
- We may try to use the product structure of integers to make a more effective search.
- Others?....

A naive search algorithm

Let us try the simplest possible exhaustive search.

In pseudocode , using an **array** A of positive integers, we might write this as:

```
product-search(int A[])
  found <- false;
  { for (i from 0 upto length(A))
      { for (j from 0 upto length(A))
            { if ( A[i]*A[j] = k )
                then { found <- true, return; }
             } }
  }; return;
```

We could (usefully) return the found values!

Is this a good algorithm? How do we compare algorithms? What is a useful measure of the performance of an algorithm?

An algorithm using preprocessing

Consider an algorithm in which we first sort the array into (say) ascending order.

For example, the result of the sorting may be

$$[2, 3, 5, 5, 6, 9, 10, 12, 24, 30].$$

Can we search this list 'faster'?

Searching a sorted list: idea

Idea: search from both ends! Why? Let us see what happens...

If the product is too small, increment left position; if too large, decrement right position.

Let us try it on our example, to find two numbers with product 72 in the list

$$[2, 3, 5, 5, 6, 9, 10, 12, 24, 30].$$

Start with 2 and 30. Then $2 \times 30 = 60 < 72$ so move left position along one and try $3 \times 30 = 90 > 72$, so move right position down the list one and try $3 \times 24 = 72$, BINGO!

It worked on this list, but can we show it works for every list. That is we need a correctness argument.

Searching a sorted list - algorithm

This gives an algorithm. In pseudocode for arrays in ascending order:

```
product-search(int A[])
  found = false;
  i <- 0; j <- length(A);
  while (i =< j)
    { if ( A[i]*A[j] = k )
           then { found <- true; return }
           else if ( A[i]*A[j] < k )
                  then { i <- i+1 }
                  else { j <- j-1 }

     };
  return;
```

Is this algorithm (a) correct, and (b) any better than the first?

Searching a sorted list: correctness

We need to show that the above algorithm does not overlook any candidate pair of numbers.

Consider an array A of integers in ascending order, and suppose that the algorithm has reached a point where the left position is i and the right position is j.

Searching a sorted list: correctness (continued)

A correctness argument: Now, suppose that no element outside of the segment i to j (inclusive) is a member of a candidate pair.

- Suppose $A[i] \times A[j] = k$, then we are done.
- Suppose $A[i] \times A[j] > k$. Then the algorithm says we consider the segment i to $j - 1$. We show that no element outside this can be part of a candidate pair. We already know that no element outside i to j is part of a candidate pair (by assumption), so is j part of a candidate pair? If it is, its partner must be in i to j (inclusive). But all elements in this segment are greater than or equal to A[i] (as the array is in ascending order). But $A[i] \times A[j] > k$, so j cannot be in a candidate pair.

  Thus any candidate pair must lie within i to $j - 1$.
- Suppose $A[i] \times A[j] < k$. Then by the same argument, elements of a candidate pair must be in the segment from $i + 1$ to j.

Measures of performance and comparing algorithms in practice

What do we measure?

We count the number of operations required to compute a result.

Which operations?

- Operations should be significant in the running time of the implementation of the algorithm.
- Operations should be of constant time.

This is called the time complexity of the algorithm, and depends on the input provided.

Time complexity of the naive searching algorithm

How many operations does the first algorithm take?

Which operations? Either multiplication or equality (it doesn't matter).

Suppose the input is an array of length N.

Best case: It could find a result with the first pair, in which case we need just 1 operation.

Worst case: It could find the result as the last pair considered, or not find a result. Need N^2 operations (1 for each pair).

Time complexity of second algorithm using sorting

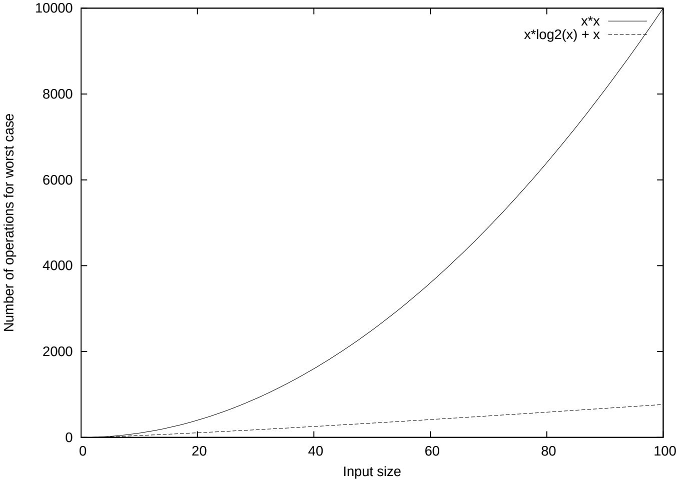
For second algorithm: Number of operations $=$
Number required for sorting $+$ number required for searching.

For sorting we can do this quite fast: For array length N, we can sort it in approx. $N \times \log_2(N)$ comparison operations (see later).

Note: $\log_2(N)$ is much smaller that N for most N, so $N \times \log_2(N)$ is much smaller than N^2.

How many operations for the searching? Answer: best case is 1 (again) and worst case is N (each operation disposes of one item in the array). So total worst case is: $\log_2(N) \times N + N$.

What about the average case? For these algorithms, the worst case is a good measure of the average case - but not always. So this algorithm is much better than the naive search using these measures.



Plot: Upper curve is the naive search, lower is the algorithm using sorting.

Course unit: Aims

The course unit has two aims:

1. To enable you to program in a different imperative language, C, with new structures and concepts, to help you become a skilled, knowledgeable and adaptable programmer.
2. To introduce you to the world of algorithms:
   How to devise algorithms for tasks, evaluate their appropriateness and measure their performance.
   How to implement algorithms in application software.
   An introduction to algorithmic literacy - what algorithms are available and how to use them.

Course unit: Structure

The website: You MUST consult the course unit website (accessed from syllabus page):

- General procedures - eg absence from labs, who to contact, etc.,
- A week-by-week calendar of activities,
- All resources: the lecture slides, sample exams and their answers, maths notes, exam and revision guidance, resit rules, etc.,
- The laboratory exercises - get them!,
- Assessment rules and details,
- An on-line C Course and other C resources.

Course unit: Structure

Course textbook: All should use the course textbook - it is necessary for success on this course unit. It is available online:

Algorithm design: foundations, examples and internet examples. Michael Goodrich and Roberto Tamassia. ISBN 0471383951, Wiley (2002).

Lecturers: Milan Mihajlovic, Ian Pratt-Hartmann and David Rydeheard.

Calendar: Full year (both semesters): 1 lecture per week, 1 laboratory session per week.

Assessment: 50% Laboratory marks (25% each semester), 50% Exam marks (15% first semester, 35% second).

Warning 1

This course unit is rather different from others. It is important that you take the following into account:

- You will be expected to read a considerable amount of material outside the lectures - this includes the course textbook and other material too.
- Lectures will give some guidance about course content and topics, laboratory exercises, etc., but much of the material of the course unit is not lectured!

Warning 2

- You need to learn the language C. The first few C laboratory sessions are crucial to the course: make sure you attend them and learn how to write C code as soon as possible. If you need help - make sure you get it (from lecturers, teaching assistants, on-line material, books, etc).
- Laboratory schedule: If you attend a laboratory on the day of a deadline, you will get an automatic extension usually to the start of your next lab session (one week hence). If you fail to attend, then your exercise will normally be considered late. There is a deadline for marking: You must get your exercises marked within 2 weeks of the extended deadline. Laboratory exercises are subject to plagiarism testing.
- This is a practical course and you need to apply yourself throughout to the practical material.

ENJOY!