Lecture 1
Introduction to Course

COMP26120

Giles Reger, Milan Mihajlovic and Lucas Cordeiro

September 2018

Staff

Giles Reger
Room 2.62
giles.reger@manchester.ac.uk

Milan Mihajlovic
milan.mihajlovic@manchester.ac.uk

Lucas Cordeiro
Room 2.28
lucas.cordeiro@manchester.ac.uk

Course Aims

The course unit has four main aims:

1. Help you become confident with a range of data structures and algorithms and able to apply them in a realistic way
2. Give you the tools required to analyse a problem and decide which algorithms or algorithmic techniques to apply
3. Emphasise practical concerns rather than mathematical analysis
4. Introduce you to C by applying it to solve new problems

Our focus is on Algorithms and you will learn C as a (helpful) side-effect.

Course Arrangements

Full year (both semesters): 1 lecture and 1 laboratory session per week

Practical concerns:

This is a practical course. Therefore, we have swapped lecture time for lab time as the best way to learn is by doing.

Labs in C:

We assume you are all competent (Java) programmers and introduce C as a second programming language. You may need to refer to resources beyond the lectures and labs to help you. Learning a new programming language is an important skill.

Course Website

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

http://studentnet.cs.manchester.ac.uk/ugt/COMP26120/

- 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,
- Links to other C resources.

Course Textbook(s)

We have a main course text book and a secondary course textbook.

Main 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).

Second textbook: This text is very useful but we will not rely on it. It is available from the library:

Introduction to Algorithms. Cormen, Leiserson, Rivest and Stein. ISBN 978-0-262-03384-8, MIT Press (2009).

Course Assessment

Coursework (50%):

- 25% each semester
- In each semester
  - 20% from coding exercises
  - 5% from 3 online quizzes (see website for when)
  - Quizzes provide early feedback on conceptual issues

Exam (50%):

- 15% first semester, 35% second semester
- Material (general concepts) covered in labs is examinable
- Note: new staff means exams likely to be a little different in style

Health Warning 1

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!

Health Warning 2

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).

- This is a practical course and you need to apply yourself throughout to the practical material.

Laboratory Arrangements

- Deadlines: The submission deadline is the start of the lab following the last lab for an exercise. The marking deadline is two weeks and two hours after the submission deadline (exception: end of semester).

- Submission System: This year we are introducing a new system for submission and marking. General points:
  - The first step is the same: run `submit`
  - Next you can login to the system (see course page for link) to see a report generated from running tests etc.
  - TAs will use this report to mark your submission. You can see if tests are failing and resubmit – you should pass these tests.
  - In the labs use the system to add yourself to the marking queue - the system knows if you're in the right lab!
  - You should receive an email after being marked (check)

- Remember - Exercises are subject to plagiarism testing

Course Structure (Subject to Change)

	Semester 1		Semester 2 (subject to reordering)	
	Lecture	Lab	Lecture	Lab
W1	Introduction	-	Graphs	*Marking*
W2	From Java to C	Pseudocode	Graphs	Graphs
W3	Pointers in C	C one	Graphs	Testing small
W4	Linked Lists in C	C two	Tractability etc	↳ world hypothesis
W5	Complexity	LinkedList	↳	↳
W6	Reading Week		Greedy Alg.	Graph C to SAT
W7	Divide and Conq.	Iterative sort	Dynamic Prog.	Knapsack
W8	Trees	Recursive sort	Designing Alg.	↳
W9	More Trees	Pangolins	Number	↳
W10	Hash Tables	↳	↳ theoretic	Public key
W11	Skip Lists	Spellchecker	*Invited Lecture*	↳ cryptograhpy
W12	Revision	↳	Revision	*Marking*

End of Admin

Most importantly,

ENJOY!

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

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

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!
(remember to bring pen+paper to lectures)

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

Where do algorithms come from?

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:

```
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 found
```

We could also (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? Is it always the same?

A naive search algorithm

Let us try the simplest possible exhaustive search.

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

```
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 found
```

We could also (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? Is it always the same?

A naive search algorithm

Let us try the simplest possible exhaustive search.

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

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

We could also (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? Is it always the same?

A naive search algorithm

Let us try the simplest possible exhaustive search.

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

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

We could also (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? Is it always the same?

A naive search algorithm

Let us try the simplest possible exhaustive search.

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

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

We could also (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? Is it always the same?

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: 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: 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: 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[])
  i := 0; j := length(A);
  while (i =< j)
     if ( A[i]*A[j] = k )
          then return true
          else if ( A[i]*A[j] < k )
                  then  i := i+1
                  else  j := j-1

  return false
```

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 (and that it terminates).

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.

Time complexity measures

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 $\frac{1}{2}N^2$ operations (1 for each pair, ignoring symmetries).

Time complexity of second algorithm using sorting

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

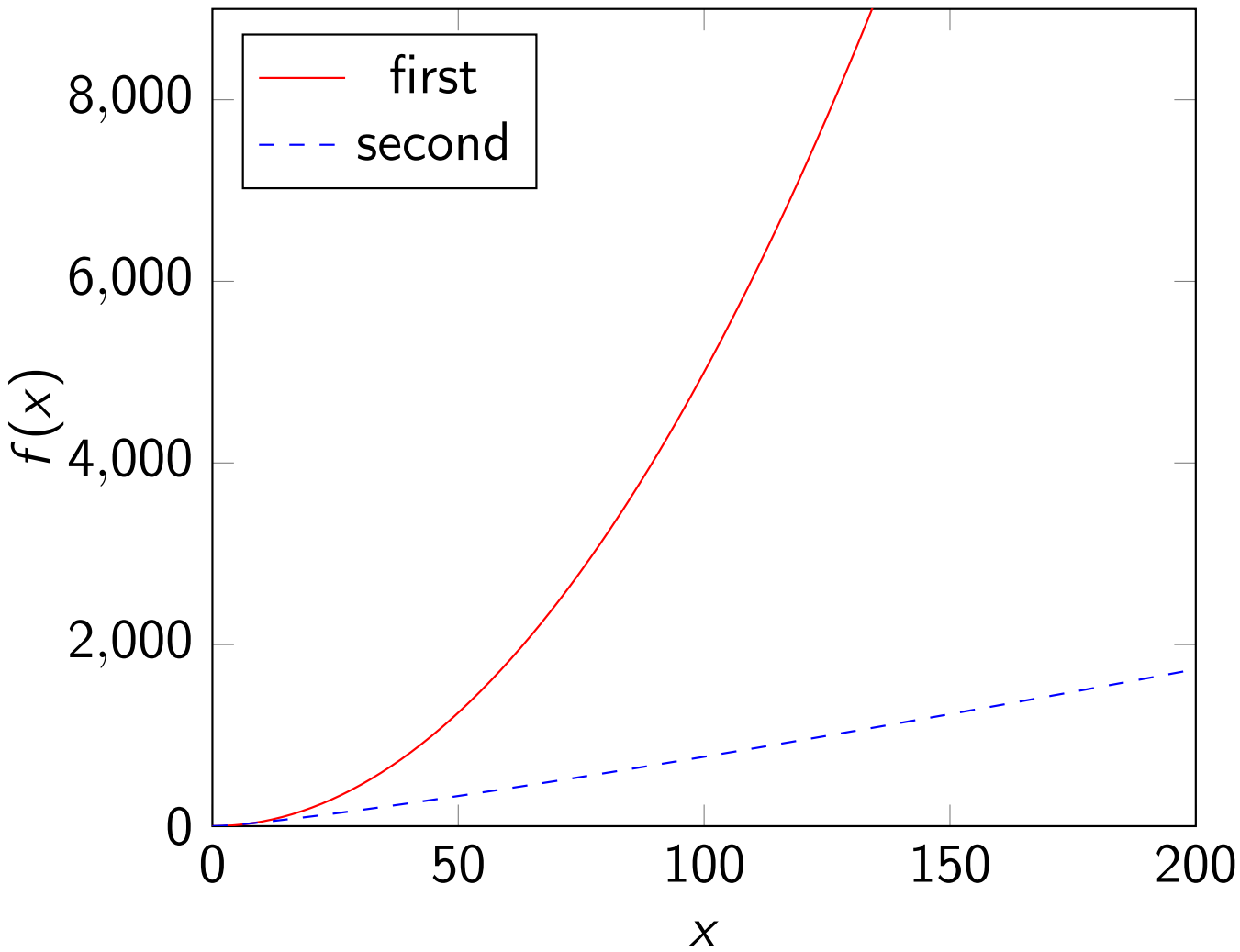
For sorting we can do this quite fast: For array length N, we can sort it in approx. $N \times \log_2(N)$ comparison operations (see later).

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.

A Graph



Summary

Course Admin - see the website!

Coming up with algorithms requires creativity but also knowing about common techniques (e.g. preprocesing with sorting)

Algorithms need to be correct

Need to be able to compare different solutions