Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011

Module Organiser, Igor Potapov

Module organiser: Dr. Igor Potapov.

Module website: http://www.csc.liv.ac.uk/∼igor/COMP309

Module Introduction

This module builds on the material that you learned in Comp202, Complexity of Algorithms.

In Comp108 and Comp202, you learned about asymptotic notation and using this notation to analyse algorithms. We will be using this notation, and the techniques that you learned in Comp202.

In Comp202, you learned basic algorithmic paradigms such as divide-and-conquer, the greedy method, and dynamic programming. We will be looking at the last two in more depth and seeing how to use them to efficiently solve computational problems.

You also learned some fundamental graph algorithms such as depth-first-search and minimum spanning tree algorithms. In COMP309 you will also study the all-pairs shortest paths problem.

In Comp202, you learned about the complexity classes P and NP. You learned about polynomial-time reductions and you were told about several NP-complete problems. In this module, we will do several polynomial-time reductions in detail and you will learn how to tell for yourself whether a computational problem is NP-complete.

We will also study the complexity of approximation problems. Given that an optimisation problem is difficult to solve exactly, we will want to know under what circumstances we can efficiently compute an approximate solution.

Module Overview. Part 1: Algorithmic Paradigms

The two paradigms that we will consider are greedy algorithms and dynamic programming.

Our study of greedy algorithms will focus on a scheduling problem called the activity-selection problem. The idea is that there are a collection of activities with associated start and finish times and we want to schedule as many of them as possible on shared resource.

Our study of dynamic programming will focus on the matrix-chain multiplication problem. In this problem, we are given a sequence of matrices and we want to figure out in which order to multiply pairs of them so that the overall multiplication can be done as quickly as possible.

Module Overview. Part 1: Algorithmic Paradigms

The two paradigms that we will consider are greedy algorithms and dynamic programming.

Our study of greedy algorithms will focus on a scheduling problem called the activity-selection problem. The idea is that there are a collection of activities with associated start and finish times and we want to schedule as many of them as possible on shared resource.

Our study of dynamic programming will focus on the matrix-chain multiplication problem. In this problem, we are given a sequence of matrices and we want to figure out in which order to multiply pairs of them so that the overall multiplication can be done as quickly as possible.

Module Overview. Part 1: Algorithmic Paradigms

The two paradigms that we will consider are greedy algorithms and dynamic programming.

Our study of greedy algorithms will focus on a scheduling problem called the activity-selection problem. The idea is that there are a collection of activities with associated start and finish times and we want to schedule as many of them as possible on shared resource.

Our study of dynamic programming will focus on the matrix-chain multiplication problem. In this problem, we are given a sequence of matrices and we want to figure out in which order to multiply pairs of them so that the overall multiplication can be done as quickly as possible.

We will also study the all-pairs shortest paths problem. In this problem, we are given a graph and we want to compute the shortest distance between every pair of vertices.

You will be given some exercises that give you more practice using these paradigms, and more examples arise later in the module.

Module Overview. Part 2: Pattern Matching

We will focus on an important application which arises in text editing, DNA mapping, and the development of algorithms for searching on the world-wide web.

The idea is that we have a pattern (a sequence of letters) and a text (a much longer sequence of characters) and we want to find all occurrences of the pattern in the text.

We will consider a simple, brute-force algorithm for this problem, then we will look at an alternative approach inspired by finite-state machines. Finally, we will study the algorithm of Knuth, Morris and Pratt.

Module Overview. Part 2: Pattern Matching

We will focus on an important application which arises in text editing, DNA mapping, and the development of algorithms for searching on the world-wide web.

The idea is that we have a pattern (a sequence of letters) and a text (a much longer sequence of characters) and we want to find all occurrences of the pattern in the text.

We will consider a simple, brute-force algorithm for this problem, then we will look at an alternative approach inspired by finite-state machines. Finally, we will study the algorithm of Knuth, Morris and Pratt.

Module Overview. Part 2: Pattern Matching

We will focus on an important application which arises in text editing, DNA mapping, and the development of algorithms for searching on the world-wide web.

The idea is that we have a pattern (a sequence of letters) and a text (a much longer sequence of characters) and we want to find all occurrences of the pattern in the text.

We will consider a simple, brute-force algorithm for this problem, then we will look at an alternative approach inspired by finite-state machines. Finally, we will study the algorithm of Knuth, Morris and Pratt.

Module Overview. Part 3: String Algorithms

We will consider some applications related to the basic pattern-matching application of Part 2.

We will study the problem of finding the longest common subsequence between two strings. This is useful for applications such as file comparison, and provides a further example of dynamic programming.

We will also consider text compression including Huffman coding (an example of a successful greedy algorithm).

Module Overview. Part 3: String Algorithms

We will consider some applications related to the basic pattern-matching application of Part 2.

We will study the problem of finding the longest common subsequence between two strings. This is useful for applications such as file comparison, and provides a further example of dynamic programming.

We will also consider text compression including Huffman coding (an example of a successful greedy algorithm).

Module Overview. Part 3: String Algorithms

We will consider some applications related to the basic pattern-matching application of Part 2.

We will study the problem of finding the longest common subsequence between two strings. This is useful for applications such as file comparison, and provides a further example of dynamic programming.

We will also consider text compression including Huffman coding (an example of a successful greedy algorithm).

Module Overview. Part 4: NP-Completeness

We will recall the basic definitions from Comp202, and then have a detailed look at polynomial-time reductions, showing that the following problems are NP-complete: 3-Conjunctive Normal Form Satisfiability (3-CNF), Clique, Vertex Cover, and Subset Sum.

Module Overview. Part 5: Approximation Algorithms and Complexity

We will consider optimisation problems in which the goal is to take a problem instance and find a feasible solution which is as good as possible with respect to some measure.

We will typically look at problems for which it is NP-hard to compute an optimal solution.

Module Overview. Part 5: Approximation Algorithms and Complexity

We will consider optimisation problems in which the goal is to take a problem instance and find a feasible solution which is as good as possible with respect to some measure.

We will typically look at problems for which it is NP-hard to compute an optimal solution.

Sequential vs. Parallel Algorithms

Many software companies have applications which are in use by their customers that have significant runtime and for which fast runtime is a necessity or a competitive advantage. There has always been the pressure to make such applications go faster.

We will consider how to compute things faster in parallel by breaking a problem into smaller pieces and solve larger problems without resorting to larger computers.

What kind of parallel speedup is required, what architecture should be used and whether a problem is amenable at all to a parallel attack

Sequential vs. Parallel Algorithms

Many software companies have applications which are in use by their customers that have significant runtime and for which fast runtime is a necessity or a competitive advantage. There has always been the pressure to make such applications go faster.

We will consider how to compute things faster in parallel by breaking a problem into smaller pieces and solve larger problems without resorting to larger computers.

What kind of parallel speedup is required, what architecture should be used and whether a problem is amenable at all to a parallel attack

Sequential vs. Parallel Algorithms

Many software companies have applications which are in use by their customers that have significant runtime and for which fast runtime is a necessity or a competitive advantage. There has always been the pressure to make such applications go faster.

We will consider how to compute things faster in parallel by breaking a problem into smaller pieces and solve larger problems without resorting to larger computers.

What kind of parallel speedup is required, what architecture should be used and whether a problem is amenable at all to a parallel attack

Learning Outcomes

At the conclusion of the course students should:

- Have an understanding of the role of algorithmics within computer science.
- Have expanded their knowledge of computational complexity theory.
- Be aware of current research-level concerns in the field of algorithm design.

Learning Algorithms

Learning about algorithms and complexity is difficult because it is not enough to understand the material in lecture.

Applying the ideas that we learn to a new problem takes some ingenuity and mathematical skill.

The way to acquire the skill is to practise – I will give you lots of example exercises, which you should solve yourself.

As in Comp202, we will describe algorithms in pseudocode rather than using the full syntax of any particular language. You should be able to implement the algorithms that we describe in a real programming language.

Pseudocode conventions

- Assignments will have the form $a \leftarrow b$ ($a = b$ refers to the comparison between a and b).
- Sometimes elementary instructions will be expressed in natural language (for instance “sort the array A in increasing order” will be an elementary instructions).
- **if-else**, **for**, and **while** control structures will be used for selection statements and loops.
- The instruction **return** e returns (i.e. “prints”) the value of e and exits from the program execution.
- Indentation will indicate block structure.

Pseudocode conventions

- Assignments will have the form $a \leftarrow b$ ($a = b$ refers to the comparison between a and b).
- Sometimes elementary instructions will be expressed in natural language (for instance “sort the array A in increasing order” will be an elementary instructions).
- **if-else**, **for**, and **while** control structures will be used for selection statements and loops.
- The instruction **return** e returns (i.e. “prints”) the value of e and exits from the program execution.
- Indentation will indicate block structure.

Pseudocode conventions

- Assignments will have the form $a \leftarrow b$ ($a = b$ refers to the comparison between a and b).
- Sometimes elementary instructions will be expressed in natural language (for instance “sort the array A in increasing order” will be an elementary instructions).
- **if-else**, **for**, and **while** control structures will be used for selection statements and loops.
- The instruction **return** e returns (i.e. “prints”) the value of e and exits from the program execution.
- Indentation will indicate block structure.

Pseudocode conventions

- Assignments will have the form $a \leftarrow b$ ($a = b$ refers to the comparison between a and b).
- Sometimes elementary instructions will be expressed in natural language (for instance “sort the array A in increasing order” will be an elementary instructions).
- **if-else**, **for**, and **while** control structures will be used for selection statements and loops.
- The instruction **return** e returns (i.e. “prints”) the value of e and exits from the program execution.
- Indentation will indicate block structure.

Pseudocode conventions

- Assignments will have the form $a \leftarrow b$ ($a = b$ refers to the comparison between a and b).
- Sometimes elementary instructions will be expressed in natural language (for instance “sort the array A in increasing order” will be an elementary instructions).
- **if-else**, **for**, and **while** control structures will be used for selection statements and loops.
- The instruction **return** e returns (i.e. “prints”) the value of e and exits from the program execution.
- Indentation will indicate block structure.

- (Unless otherwise stated) variables will be local to the given procedure.
- Arrays are denoted by $A[i]$, but also $A[i..j]$. length(A) denotes the number of elements of the array A.
- Parameters are always passed *by value*.

More conventions as we meet them.

- (Unless otherwise stated) variables will be local to the given procedure.
- Arrays are denoted by $A[i]$, but also $A[i..j]$. length(A) denotes the number of elements of the array A.
- Parameters are always passed *by value*.

More conventions as we meet them.

- (Unless otherwise stated) variables will be local to the given procedure.
- Arrays are denoted by $A[i]$, but also $A[i..j]$. length(A) denotes the number of elements of the array A.
- Parameters are always passed *by value*.

More conventions as we meet them.

- (Unless otherwise stated) variables will be local to the given procedure.
- Arrays are denoted by $A[i]$, but also $A[i..j]$. length(A) denotes the number of elements of the array A.
- Parameters are always passed *by value*.

More conventions as we meet them.

- (Unless otherwise stated) variables will be local to the given procedure.
- Arrays are denoted by $A[i]$, but also $A[i..j]$. length(A) denotes the number of elements of the array A.
- Parameters are always passed *by value*.

More conventions as we meet them.

Textbooks

- T. H. Cormen, C. E. Leiserson, R. L. Rivest **Introduction to Algorithms,** Second Edition. MIT Press (2001).
- G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, **Complexity and Approximation** Springer 2003.
- Lovasz and Plummer, **Matching Theory** North-Holland (1986).

Take notes.

Ask questions.