G5BADS: first lecture

- In this lecture:
- Practical information
- What are data structures
- What are algorithms
- Aims and objectives of the course

Information

- Lecturer: Natasha Alechina
- web page: www.cs.nott.ac.uk/~nza/G5BADS
- Two lectures a week, no practical sessions
- 75 % exam, 25 % coursework
- one formal coursework
- some informal coursework

Books

- Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis, Java Edition*.
- Weiss, *Data Structures and Algorithm Analysis in Java.*
- Lafore, *Data Structures and Algorithms in Java.*
- Aho et al., *Data Structures and Algorithms*.
- Cormen et al., *Introduction to Algorithms.*

More textbooks

- Sahni, Data Structures, Algorithms, and Applications in Java
- Goodrich and Tamassia, Algorithm Design - Foundations, Analysis and Internet Examples
- Any other standard textbook you can find.

- Also useful (but not sufficient on its own):
- Harel, Algorithmics: The spirit of computing.
- Bailey, D. A. Java Structures.

Prerequisites

- PRG (Java)
- MC1 (Mathematics for Computer Scientists 1)

Mathematical Background

- Proofs by induction
- Recursion
- Logarithms (for next lecture)

- If you forgot any of those, see Chapter 2 (Mathematical Preliminaries) of Shaffer's book.

Knowledge of Java

- G51PRG (Introduction to Programming in Java) is a prerequisite for the course.
- I will use Java code for implementation examples.
- Coursework involves writing a Java program.
- If you don’t know Java, read Java tutorial in Shaffer's book or the Sun online Java tutorial on http://www.java.sun.com/tutorial (Getting Started and Learning the Java Language).

Content of the Course

- In a way, continues PRG, but not *about* Java
- Data structure: a way data is organised in computer memory; for example: array, list, list of lists, tree, table...
- Algorithm: a sequence of steps which provides a solution to a given problem. For example, an explanation how to use a telephone book to find somebody's telephone.

Abstraction

Algorithms are independent of a particular programming language. They get *implemented* in a particular program. The course is about analysing and solving problems on an abstract level before starting to program.

Issues in Data Structures

- Even if we end up using ready-made Java API structures, need to be able to decide when to use what.
- dynamic/static
- how much space it uses
- how easy it is to search
- etc. (does it make solving the problem easier)

Issues in Algorithms

- how much memory/time does it use (efficiency)
- does it actually solve the problem (correctness)

Aims and Objectives

- Aim of the course: understanding of issues involved in program design; good working knowledge of common algorithms and data structures
- Objectives:
- be able to identify the functionality required of the program in order to solve the task at hand;
- design data structures and algorithms which express this functionality in an efficient way;
- be able to evaluate a given implementation in terms of its efficiency and correctness.

2000/2001 coursework problem

Given:

- array of spam patterns,e.g. [“Make money fast”, “Make money in your spare time”, “Save money”]
- an email message

Design and implement a time-efficient algorithm which detects whether the message contains any of the spam patterns.

Basically a string-matching problem.

Most common solution

```
for (int i = 0; i < patterns.length; i++) {
   if(message.indexOf(patterns[i]) != -1) {
      return true;
   }
}
return false;
```

In other words, iterate through the patterns calling `indexOf()`.

What does `indexOf()` do?

Most common solution

What does `indexOf()` do?

It scans the message from the beginning looking for the first letter of the pattern. Then it starts trying to match the rest of the pattern. If this fails, it starts looking for the first letter again.

Most common solution

So, we have a nested loop:

```
for (int i = 0; i < patterns.length; i++) {
 for (int j = 0; j < message.length(); j++) {
   for (int k=0; k<patterns[i].length(); k++){
```

And in the worst case the outer loop iterates through all patterns, middle loop each time through all the message and inner loop through almost all of the pattern. This method is a brute force solution.

Illustration:

Let the message string be

`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

and the pattern

`aaaaaaaaaaaz`

Optimisations

- Use a better algorithm e.g. Boyer-Moore
- Reduce the number of patterns by removing superstrings
- Hash a list of patterns beginning with the same character under the key corresponding to this character. Scan the message once and for each character in the message check if there are any patterns starting with this character. If yes, try to match all of them in turn.

Serious optimisation

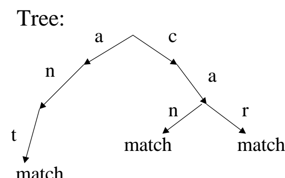
- Create a tree for holding the patterns
- A variant of Aho-Corasick algorithm (?) from Philippa Cowderoy's program: The speed does not depend on the number of patterns! (The size of the tree does).

String matching using a tree

Patterns:
ant
can
car

String to check:
a can

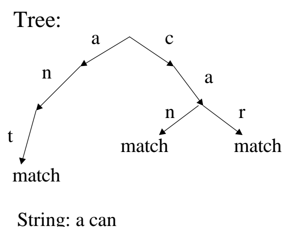
Tree:



String matching using a tree

- We move along the string taking one character at a time
- We maintain a list **`plist`** of potential matches (references to edges). The list is of size at most k (maximal pattern length)
- For each character in the string, update/remove items in **`plist`** and add at most one new item.

Tree:

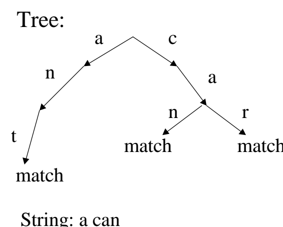


String: a can

String matching using a tree

- If we find character **`char`**:
  - if there is an edge from the root labelled **`char`** we add to **`plist`** a reference to that edge (start matching some pattern from the beginning)
  - for every item in **`plist`**, if there is an edge from there labelled **`char`**, replace the item by reference to that edge (continue matching some pattern)

Tree:

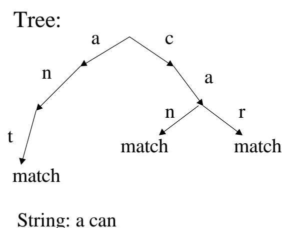


String: a can

String matching using a tree

- If we find character **`char`**:
  - if there is no edge labelled **`char`** from the item, remove it (could not match some pattern)
  - finally, if reached **`match`**, return **`true`** (successfully matched some pattern).
- When reached the end of string, return false

Tree:



String: a can

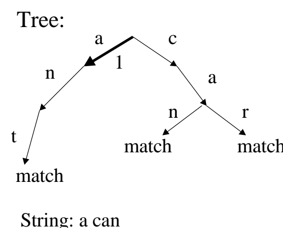
String matching using a tree

Trace:

a can

plist: 1

Tree:



String: a can

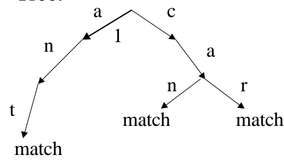
String matching using a tree

Trace:

a can

plist:

Tree:



String: a can

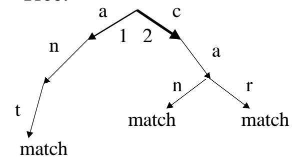
String matching using a tree

Trace:

a can

plist: 2

Tree:



String: a can

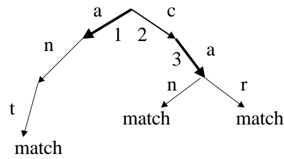
String matching using a tree

Trace:

a can

plist: 1,3

Tree:



String: a can

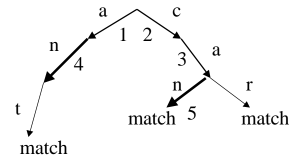
String matching using a tree

Trace:

a can

plist: 4,5

Tree:



String: a can

Lesson of the coursework

- The tree-based solution is more efficient
- In tests on large messages and large arrays of patterns, the difference in performance was many orders of magnitude (80ms vs 20 minutes)
- But can we analyse efficiency without testing, from first principles?
- Subject of next lecture.

Two Search Algorithms for Ordered Arrays

Linear search	Binary search
linearSearch(array, item) for each index in the array: if value at index equals to item, return true not found - return false	check value at the middle index of the array if item found, return true if value at middle index is greater than the item, search the first half (if empty return false) else search the second half (if empty return false)

Two Search Algorithms

Linear search:	Binary search
Al-Biqami	Al-Biqami
Alechina	Alechina
Allsebrook	Allsebrook
Armitage	Armitage
Ashman	Ashman
Backhouse	Backhouse
Beesley	Beesley
Beiley	***Beiley***
Belavkin	Belavkin
Belfield	Belfield
Benford	Benford
Blampied	Blampied

Two Search Algorithms

Linear search:	Binary search
Al-Biqami	Al-Biqami
Alechina	Alechina
Allsebrook	Allsebrook
Armitage	Armitage
Ashman	Ashman
Backhouse	*Backhouse*
Beesley	Beesley
Beiley	**Beiley**
Belavkin	Belavkin
Belfield	Belfield
Benford	Benford
Blampied	Blampied

Two Search Algorithms

Linear search:	Binary search
Al-Biqami	*Al-Biqami*
Alechina	*Alechina*
Allsebrook	*Allsebrook*
Armitage	*Armitage*
Ashman	*Ashman*
Backhouse	*Backhouse*
Beesley	Beesley
Beiley	**Beiley**
Belavkin	*Belavkin*
Belfield	Belfield
Benford	Benford
Blampied	Blampied

Two Search Algorithms

Linear search:	Binary search
Al-Biqami	*Al-Biqami*
Alechina	*Alechina*
Allsebrook	*Allsebrook*
Armitage	*Armitage*
Ashman	*Ashman*
Backhouse	*Backhouse*
Beesley	Beesley
Beiley	**Beiley**
Belavkin	*Belavkin*
Belfield	*Belfield*
Benford	*Benford*
Blampied	*Blampied*

Two Search Algorithms

Linear search:	Binary search
Al-Biqami	*Al-Biqami*
Alechina	*Alechina*
Allsebrook	*Allsebrook*
Armitage	*Armitage*
Ashman	*Ashman*
Backhouse	*Backhouse*
Beesley	*Beesley*
Beiley	**Beiley**
Belavkin	Belavkin
Belfield	*Belfield*
Benford	*Benford*
Blampied	*Blampied*

Two Search Algorithms

Linear search:	Binary search
Al-Biqami	*Al-Biqami*
Alechina	*Alechina*
Allsebrook	*Allsebrook*
Armitage	*Armitage*
Ashman	*Ashman*
Backhouse	*Backhouse*
Beesley	*Beesley*
Beiley	Beiley
Belavkin	*Belavkin*
Belfield	*Belfield*
Benford	*Benford*
Blampied	*Blampied*