G5BADS: Algorithms and Data Structures

Lecturer: Natasha Alechina
Email: nza@cs.nott.ac.uk
Web page for the course: http://www.cs.nott.ac.uk/~nza/G5BADS

Textbook

- Textbook website is http://ww0.java4.datastructures.net
- The textbook web site has pdf handouts for all the slides; I am not going to change much. If you want to get slides before the lectures, download them from there

Other textbooks

- The field of algorithms and data structures is well established, and there are lots of good textbooks which cover more or less the same material. For example:
  - 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.
  - Sahni, Data Structures, Algorithms, and Applications in Java

Pre-requisites

- PRG (Java)
  - I will use Java code for implementation examples.
  - Coursework involves writing a Java program
  - Revision: Goodrich and Tamassia or java.sun.com
- MCS (Mathematics for Computer Scientists 1)
  - Proofs by induction
  - Recursion
  - Logarithms
  - Revision: Goodrich and Tamassia

Assessment

- 75 % written exam, 25 % formal coursework
- One coursework, to be announced in the beginning of November, deadline end of November.
- Some informal coursework
- No tutorials or labs

More textbooks

- Any other standard textbook you can find.
- Also useful (but not sufficient on their own):
  - Harel, Algorithmics: The spirit of computing.
  - Bailey, D. A. Java Structures.
What are Algorithms and Data Structures

  For example:
  - a procedure for computing \( n! \) given \( n \)
  - an instruction for assembling a piece of furniture
  - various sorting and searching algorithms
- Data structure: a way data is organised in computer memory; for example: array, list, tree, table...

Aims and objectives of the course

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.

Plan of the course

- Algorithms analysis (two lectures)
  - Proving correctness of algorithms (one lecture)
  - Recursion, stacks and queues (two lectures)
- Lists (two lectures)
- Trees (one lecture)
- Priority queues and heaps (one lecture)
- Maps (one lecture)
- Search trees (two lectures)
- External searching in B-trees (one lecture)
- Sorting (three lectures)
- Graphs (three lectures)

Analysis of Algorithms

An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.

Running Time (§3.1)

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
- Easier to analyze
  - Crucial to applications such as games, finance and robotics

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- Plot the results
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation.
- Characterizes running time as a function of the input size, \( n \).
- Takes into account all possible inputs.
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment.

Pseudocode

- High-level description of an algorithm.
- More structured than English prose.
- Less detailed than a program.
- Preferred notation for describing algorithms.
- Hides program design issues.

Example: find max element of an array

```
Algorithm arrayMax(A, n)
Input array A of n integers
Output maximum element of A

currentMax ← A[0]
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]
return currentMax
```

Pseudocode Details

- Control flow:
  - if  
  - while
  - repeat
  - for
  - Indentation replaces braces

- Method declaration:
  - Algorithm method(arg[, arg]...)

- Method call:
  - var.method(arg[, arg]...)

- Return value:
  - return expression

- Expressions:
  - Equality testing
  - Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

- A CPU
- An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character.
- Memory cells are numbered and accessing any cell in memory takes unit time.

Seven Important Functions (§3.3)

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ \( \log n \)
  - Linear = \( n \)
  - N-Log-N = \( n \log n \)
  - Quadratic = \( n^2 \)
  - Cubic = \( n^3 \)
  - Exponential = \( 2^n \)

- Time \( T(n) \) vs. input size \( n \):

```
T(n) = 1, T(n) = \log n, T(n) = n
T(n) = n, T(n) = n^2
T(n) = n^2, T(n) = 2^n
```

Introduction; Analysis of Algorithms 13© 2004 Goodrich, Tamassia

Introduction; Analysis of Algorithms 14© 2004 Goodrich, Tamassia

Introduction; Analysis of Algorithms 15© 2004 Goodrich, Tamassia

Introduction; Analysis of Algorithms 16© 2004 Goodrich, Tamassia

Introduction; Analysis of Algorithms 17© 2004 Goodrich, Tamassia

Introduction; Analysis of Algorithms 18© 2004 Goodrich, Tamassia
Introduction; Analysis of Algorithms

Primitive Operations
- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Examples:
- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Counting Primitive Operations (§3.4)

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm `arrayMax(A, n)`

```plaintext
currentMax ← A[0]
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]
{ increment counter i }
return currentMax
```

<table>
<thead>
<tr>
<th># operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>2(n − 1)</td>
</tr>
<tr>
<td>2(n − 1)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
</tr>
<tr>
<td>8n − 3</td>
</tr>
</tbody>
</table>

Estimating Running Time

Algorithm `arrayMax` executes 8n − 3 primitive operations in the worst case. Define:
- \( a \) = Time taken by the fastest primitive operation
- \( b \) = Time taken by the slowest primitive operation

Let \( T(n) \) be worst-case time of `arrayMax`. Then
\[
\begin{align*}
    a \cdot (8n − 3) & \leq T(n) \\
    b \cdot (8n − 3) & \geq T(n)
\end{align*}
\]

Hence, the running time \( T(n) \) is bounded by two linear functions.

Growth Rate of Running Time

- Changing the hardware/software environment
  - Affects \( T(n) \) by a constant factor, but
  - Does not alter the growth rate of \( T(n) \)

The linear growth rate of the running time \( T(n) \) is an intrinsic property of algorithm `arrayMax`.

Big-Oh Notation (§3.4)

Given functions \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( O(g(n)) \) if there are positive constants \( c \) and \( n_0 \) such that
\[
f(n) \leq cg(n) \text{ for } n \geq n_0
\]

Example: \( 2n + 10 \) is \( O(n) \)
- \( 2n + 10 \leq cn \)
- Choose \( c = 2 \)
- \( n \geq 10 \)
- \( n_0 = 10 \)

Big-Oh Example

Example: the function \( n^2 \) is not \( O(n) \)
- \( n^2 \leq cn \)
- \( n \leq c \)
- The above inequality cannot be satisfied since \( c \) must be a constant

\[
\begin{align*}
    n^2 & \quad \text{for } n \geq 1 \\
    n & \quad \text{for } n \geq 1 \text{ (linear)}
\end{align*}
\]
More Big-Oh Examples

- **7n - 2**
  7n - 2 is \(O(n)\)
  need \(c > 0\) and \(n_0 \geq 1\) such that \(7n - 2 \leq cn\) for \(n \geq n_0\)
  this is true for \(c = 7\) and \(n_0 = 1\)

- **3n^2 + 20n + 5**
  3n^2 + 20n + 5 is \(O(n^2)\)
  need \(c > 0\) and \(n_0 \geq 1\) such that \(3n^2 + 20n + 5 \leq cn^2\) for \(n \geq n_0\)
  this is true for \(c = 4\) and \(n_0 = 21\)

- **3 \log n + 5**
  3 \log n + 5 is \(O(\log n)\)
  need \(c > 0\) and \(n_0 \geq 1\) such that \(3 \log n + 5 \leq c \log n\) for \(n \geq n_0\)
  this is true for \(c = 8\) and \(n_0 = 2\)

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement \(f(n) = O(g(n))\) means that the growth rate of \(f(n)\) is no more than the growth rate of \(g(n)\)
- We can use the big-Oh notation to rank functions according to their growth rate

<table>
<thead>
<tr>
<th>(f(n)) grows more</th>
<th>(g(n)) grows more</th>
<th>Same growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Another example

**Algorithm: alg**
- Input: positive integer \(n\), which is a power of 2
- Output: integer \(m\) such that \(2^m = n\)

\[
m \leftarrow 0 \\
\text{while } (n \geq 2) \\
\quad n \leftarrow n/2 \\
\quad m++ \\
\text{return } m
\]
Another example

Algorithm: alg
Input: positive integer n, which is a power of 2
Output: integer m such that $2^m = n$

```
m ← 0
while (n ≥ 2)
    n ← n/2
    m++
return m
```

all together $5 \log_2(n) + 2$

Computing Prefix Averages

We further illustrate asymptotic analysis with two algorithms for prefix averages.

1. The i-th prefix average of an array $X$ is average of the first $(i+1)$ elements of $X$:
   
   $$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i+1}$$

2. Computing the array $A$ of prefix averages of another array $X$ has applications to financial analysis.

For example, consider the following array $X$:

```
0 5 10 15 20 25 30 35
```

```
1234567
```

Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition:

Algorithm prefixAverages1($X$, n)

Input array $X$ of $n$ integers
Output array $A$ of prefix averages of $X$

```
A ← new array of $n$ integers
for $i$ ← 0 to $n-1$
do
    $s ← X[0]$
    for $j$ ← 1 to $i$
do
        $s ← s + X[j]$
    $A[i] ← s / (i+1)$
return $A$
```

The running time of prefixAverages1 is $O(1 + 2 + \ldots + n)$

The sum of the first $n$ integers is $n(n+1)/2$

Thus, algorithm prefixAverages1 runs in $O(n^2)$ time.

Prefix Averages (Linear)

The following algorithm computes prefix averages in linear time by keeping a running sum:

Algorithm prefixAverages2($X$, n)

Input array $X$ of $n$ integers
Output array $A$ of prefix averages of $X$

```
A ← new array of $n$ integers
$s ← X[0]$ 
for $i$ ← 0 to $n-1$
do
    $s ← s + X[i]$
    $A[i] ← s / (i+1)$
return $A$
```

Algorithm prefixAverages2 runs in $O(n)$ time.

Other way...

$1 + 2 + \ldots + (n-1) + n = ?$

Easier to compute the sum twice:

```
1 + 2 + \ldots + (n-1) + n
+ n + (n-1) + \ldots + 2 + 1
= (n+1) + (n+1) + \ldots + (n+1) = n(n+1)
```

...and divide by 2:

```
1 + 2 + \ldots + (n-1) + n = n(n+1)/2.
```
Math you need to Review

- Summations
- Logarithms and Exponents
  - **properties of logarithms:**
    - \( \log_b(xy) = \log_b x + \log_b y \)
    - \( \log_b \left( \frac{x}{y} \right) = \log_b x - \log_b y \)
    - \( \log_b a = \frac{\log_a x}{\log_a b} \)
  - **properties of exponentials:**
    - \( a^{b+c} = a^b a^c \)
    - \( a^{bc} = (a^b)^c \)
    - \( a^{b/c} = a^{b-c} \)
    - \( b = a^{\log_a b} \)
    - \( b^c = a^{c \log_a b} \)

Relatives of Big-Oh

- **big-Omega**
  - \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \)
- **big-Theta**
  - \( f(n) \) is \( \Theta(g(n)) \) if there are constants \( c' > 0 \) and \( c'' > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \) for \( n \geq n_0 \)

Intuition for Asymptotic Notation

- **Big-Oh**
  - \( f(n) \) is \( O(g(n)) \) if \( f(n) \) is asymptotically less than or equal to \( g(n) \)
- **big-Omega**
  - \( f(n) \) is \( \Omega(g(n)) \) if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)
- **big-Theta**
  - \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically equal to \( g(n) \)

Example Uses of the Relatives of Big-Oh

- \( 5n^2 \) is \( \Theta(n^2) \)
  - \( f(n) \) is \( \Theta(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \)
  - Let \( c = 1 \) and \( n_0 = 1 \)
- \( 5n^2 \) is \( \Theta(n) \)
  - \( f(n) \) is \( \Theta(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \)
  - Let \( c = 1 \) and \( n_0 = 1 \)
- \( 5n^2 \) is \( \Theta(n^2) \)
  - \( f(n) \) is \( \Theta(g(n)) \) if it is \( \Theta(n^2) \) and \( O(n^2) \). We have already seen the former, for the latter recall that \( f(n) \) is \( O(n^2) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq c \cdot n^2 \) for \( n \geq n_0 \)
  - Let \( c = 5 \) and \( n_0 = 1 \)

Informal coursework

- First informal coursework is on [http://www.cs.nott.ac.uk/~nza/G5BADS](http://www.cs.nott.ac.uk/~nza/G5BADS)
- Answers will be published next week.