Lecture notes for Week 1: Introduction

by Ken Clowes

Table of contents

1. Topics

1. Course management.
2. Algorithm examples.
3. Algorithm analysis.
4. Labs.

1.1. Textbook portions covered

Introduction to Algorithms (Cormen et al.)

Chapter 2 (p. 15-40)

Engineering Algorithms...(Clowes "online book")

Chapter 1 (Sections 1.1-1.3) (p. 2-15) (Section 1.4 is covered in labs.)

2. Lecture 1 (Friday, 7 January 2005)

2.1. Administrivia

Professor:

- Name: Ken Clowes
- Office: ENG 449
- Phone: (416) 979-5000 ext. 6099
- Email: `kclowes@ee.ryerson.ca`
- Home page: `http://www.ee.ryerson.ca/~kclowes` (http://www.ee.ryerson.ca/~kclowes)
- Counselling hours:
  - Tuesday 10-Noon
  - Wednesday 10-Noon
  - Thursday 11-noon

Course home page:

The home page is `http://www.ee.ryerson.ca/~courses/ele428` (http://www.ee.ryerson.ca/~courses/ele428) .

The course web site has been totally redesigned this year. In particular, animations of many of the algorithms covered in the course are included. These animations should help you to understand how the algorithms work.

Course directory:

The home directory for the course is `/home/courses/ele428`.

The directory and its subdirectories include a wide variety of files including:

- Lab *zip* files.
- Old exams/tests/problem sets.
- Source code examples.

Local Access Only

Unlike the web site, the course directory can only be accessed when you are logged onto a departmental computer. Furthermore, you cannot create or modify any files in this directory. (For example, if you attempt to compile a C source code file, it will fail because you cannot create the generated object file.) However, you can copy any file to a directory you own (where, for example, you could compile it.)

2.2. Announcements

- Labs start next Monday (Jan 10).

2.3. Course Management

- Hard copy distributed in class.
- Also available on the course home page.

2.4. General Remarks

- This is a *theory* course, NOT a programming course.
- The theory examined includes the description and analysis of various well-known algorithms and data structures.
- The topics are discussed in a generic way without reference to any particular programming methodology or language.
- The lab portion of the course is designed to give you practical experience with some of the algorithms studied using the C programming language. (Some of the later labs may also have a Java component.)

2.5. Introduction to algorithms

- Informally, an algorithm is a *systematic procedure to solve a problem*.
- A problem may have several different algorithms that solve the problem. Differences between such algorithms may include:
  1. Efficiency (how fast the problem is solved).
  2. Complexity (how tricky or easy it is to implement in software).
  3. Resources (how many memory or other resources the algorithm needs).

2.5.1. Algorithm Examples (informal)

1. Let's look at some algorithms to sort a collection of objects.

2. Let's also analyze the complexity of each sort algorithm. Assume that the basic operation is the comparison of two objects. Ideally, we want to know the total number of comparisons required as a function of the size of the problem. The size of the sorting problem is *n* (the number of objects to sort).

2.5.1.1. Selection Sort

1. Informal description: Move the smallest value in the unsorted collection to the the sorted collection. Keep on doing this until there is nothing left to sort.
2. The "move smallest" step decrements the size of the unsorted collection. Since this step is repeated until all items have been sorted, it is done *n* times.
3. The "move smallest" step has to find the smallest value in the unsorted collection. If the unsorted collection has *p* objects, we have to do *p-1* comparisons.
4. In general, we need to do *n-1* comparisons on the first pass. The subsequent passes require *n-2*, *n-3*...*0* comparisons.
5. Hence the total number of comparisons is:
   n-1 + n-2 ... +1 = n(n-1)/2 = n^2/2 - n/2
6. This analysis indicates that the number of comparisons to sort a collection of *n* objects using the *Selection Sort* algorithm is a quadratic equation.
7. Hence, we say that SelectionSort is a **quadratic** algorithm.

2.5.1.2. Merging sorted collections

1. **Problem:** Given 2 sorted collections, combine (*merge*) them into a single sorted collection.
2. **Algorithm (informal):** Move the smallest item from either sorted partial collection to the fnal sorted (merged) collection. (*Note:* Since each partial collection is sorted, the smallest value in each is always "at the top"—there is no need to seach for it.)
3. **Analysis:** Each comparison adds one item to the final sorted collection. Consequently, at most *n* comparisons are required to merge two sorted collections with a total of *n* items into a single totally sorted collection.
4. Since the total number of comparisons needed to merge two sorted collections into a single sorted collection is proportional to *n*, we say that the *Merge* algorithm has **linear** complexity.

2.5.1.3. MergeSort

1. **Problem:** Sort *n* items.
2. **Algorithm (informal):**

1. If there are less than 2 items in the collection, then the collection is sorted. STOP.
2. Otherwise, split the collection in 2 and sort each sub-collection by using this algorithm.
3. Finally, merge the two sub-collections and STOP.

3. **Analysis (informal):** Since our analysis only requires that we determine the number of comparisons, we need only be concerned with the *Merge* operation since the other steps in the *MergeSort* algorithm do not compare the values of items in the collection.

   The question then becomes: "How many times do we merge sub-collections and how many items are merged?"

   This question can be answered as follows:

   - To create the final sorted collection with *n* items, we merge 2 *n/2* collections using *n* comparisons.
   - Each *n/2* sorted collection is obtained by merging two *n/4* collections where each merge requires *n/2* comparisons. Since there are 2 such merges, a total of *n* comparisons are needed.
   - Similarly, the 4 *n/4* sorted collections are obtained by merging 8 *n/8* collections. Each merge requires *n/4* comparisons; the total number of comparisons for 4 such merges is *n*.
   - In general, there are *p n/(2^p)* sorted collections that are obtained by merging *2p* collections. The total number of comparisons to perform all these merges is *n*.
   - We stop splitting when the sub-collection size is 1 (or 0). If *n = 2^p*, we need to split *p* times.
   - Since each split requires *n* comparisons to obtain the *n/2^(p-1)* splits, the total number of comparisons for all of the merge operations is $n \times p = n \log n$.

4. **Analysis (summary):** #comparisons = n log n

2.6. Questions asked in lecture

Question:
The analysis of *MergeSort* ignores operations (e.g. time to split a collection). Don't we have to include all operations when analyzing an algorithm?
Answer:

Good question!

YES, in principle, we need to account for all operations used in an algorithm.

HOWEVER, if our goal is to characterize an algorithm of complexity *linear*, *quadratic*, *n*

log n, etc., we can ignore some of the operations.

For now, "trust me": # comparisons is all we need to determine a sort algorithm efficiency.

In the next 2 weeks, we will analyze this algorithm (and others) in a mathematically rigorous way (including ALL operations).

3. Lecture 2/3 (Tuesday, January 11, 2005)

3.1. Announcements

- Postscript and pdf versions of on-line book available in:
  `/home/courses/ele428/Book`

3.2. How to describe an algorithm

There are (at least) 3 ways commonly used:

English:

What we've used so far.

Step-by-step:

Used in *Engineering Algorithms...(Clowes "online book")*.

Pseudocode:

Used in *Introduction to Algorithms (Cormen et al.)*.

3.2.1. Example: InsertionSort pcode

References: *Introduction to Algorithms (Cormen et al.)* p.17

Pseudocode format

The pseudocode format used in *Introduction to Algorithms (Cormen et al.)* (cf. p.. 19-20) use an algol-like syntax (which is popular with mathematicians.) I will usually use pseudocode that is more closely related to C.

3.2.2. Example: SelectionSort step-by-step

References: *Engineering Algorithms...(Clowes "online book")* p. 6

3.2.3. Algorithm Analysis

3.2.3.1. General Technique

1. Assume each step takes constant time.
2. Determine how many times each step is performed in the worst case as a function of *n* (problem size).
3. Derive *T(n)* (time to perform algorithm) by adding up the total time for all steps. (The total time for a single step is the constant time for doing it once times the number of times it is performed.)

3.2.3.2. Big Theta notation

1. For large *n*, the value of *T(n)* is mainly determined by the fastest growing term.
2. *BigTheta* notation is simply the fastest growing term with the constant multiplier simplified to 1.
3. **Example:** If *T(n) = 3n^2 + 1234n + 567*, the quadratic term dominates and we say: *T(n) = BigTheta(n^2)*.

3.2.3.3. Example: InsertionSort

Done in class. Refer to *Introduction to Algorithms (Cormen et al.)* (p. 22-26)

3.2.3.4. Example: MergeSort

1. Derived in class: *T(n) = 2 T(n) + n*
2. Refer to *Engineering Algorithms...(Clowes "online book")* (Section 1.3.4).
3. Refer to *Introduction to Algorithms (Cormen et al.)* (p. 33-36)

Introduction to Recurrences

1. An equation like *T(n) = 2 T(n) + n* is called a **recurrence**: the value of the function is defined in terms of the function value for smaller arguments.
2. Solving a recurrence means converting the equation into an ordinary formula that is not self-referential.
3. A recurrence can only be solved if there are one (or more) known *base cases*. (This is similar to the situation for differential equations where *boundary conditions* are required.)
4. The closed form solution for *T(n) = 2 T(n) + n* where *T(1) = 0* is *T(n) = n* lg *n*.
5. References:
   - *Introduction to Algorithms (Cormen et al.)* (p 32-36)
   - *Engineering Algorithms...(Clowes "online book")* (p 12-14)

3.3. Software Tools Used in Course

All of the software tools used in the course are available at no cost and work in all major Operating Systems (including Microsoft Windows, Linux, MacOS, Solaris, etc.) The tools include:

Java SDK:

The Software Development Kit includes the Java runtime environment and the development tools (such as compiler, debugger, javadoc, etc.). It can be downloaded from:
www.java.sun (http://www.java.sun)

jedit Editor:

This general-purpose text editor can be downloaded from:
www.jedit.org (http://www.jedit.org)

cygwin Tools (including gcc C compiler):

This set of tools allows much of the standard Unix environment to run under Microsoft Windows. It includes compilers (C, C++, fortran, etc.), utilities, an XWindow implementation (XFree86) and command line shells. It can be downloaded from:
www.cygwin.com (http://www.cygwin.com)

4. Suggested Problems

Introduction to Algorithms (Cormen et al.)

- 2-2
- 2-4

Engineering Algorithms...(Clowes "online book")

- 1.1
- 1.2
- 1.3
- 1.4
- 1.5
- 1.9
- 1.20
- 1.22