COMP 460 Lecture Notes

R. I. Greenberg

1 Administrivia

We'll start by going over some points on the syllabus. Read it, e.g., the parts about collaboration! Also note there is a handout 0 that you should return right away to get on the class email list and get access to the web site.

Note that COMP 460 has COMP 363 or COMP 388 (Foundations of CS) as a prerequisite. Past experience has shown that people who try going straight to COMP 460 without an undergraduate algorithms background usually do poorly. In the first week, we will go over the first four! chapters of the book, which should be review, and also cover chapter six.

2 Introduction

2.1 Some Terminologies

- A *computational problem* specifies an input/output relationship, e.g.:

  sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A reordering $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- A particular input meeting the problem spec. is an *instance* of the problem, e.g., $\langle 5, 10, 8 \rangle$ for the sorting problem. (Corresponding output is $\langle 5, 8, 10 \rangle$.)

- An *algorithm* is a computational procedure that produces the correct output for each problem instance. (Actually, sometimes we only demand that an algorithm is "usually" correct.)

  $$\text{"algorithm"} \neq \text{"program"}$$

  A program is an implementation of an algorithm in a particular programming language.

  We will generally specify algorithms in a pseudocode similar to C, Pascal, and Algol.

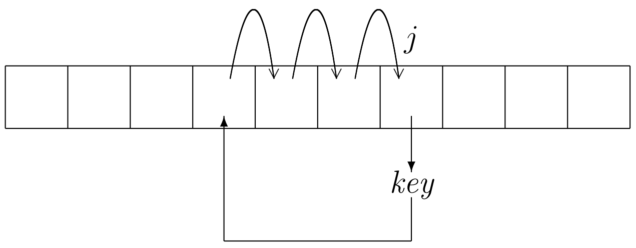
2.2 An Example Algorithm: Insertion Sort

Input is array A with a number of elements denoted by length$[A]$.

(I will use a slightly different pseudocode style than the book to avoid relying *only* on indentation to indicate block structure. You may use either style.)

```
  INSERTION-SORT($A$)
1 for $j \leftarrow 2$ to length$[A]$
2     $key \leftarrow A[j]$
3     $i \leftarrow j - 1$
4     while $i > 0$ and $A[i] > key$
5         $A[i+1] \leftarrow A[i]$
6         $i \leftarrow i - 1$
7     endwhile
8     $A[i+1] \leftarrow key$
9 endfor
```

We can see how this algorithm works by noting that lines 2.2 through 2.2 insert $A[j]$ into the sorted nondecreasing sequence $A[1..j-1]$:



2.3 Analysis of Algorithms

- Running time generally increases with input size and is, therefore, expressed as a function of input size. (Sometimes running time also differs for different inputs of the same size.)

  There are various possible measures of input size. In most contexts, we can think of it as the number of items in the input, e.g., the size n of a sequence input to the sorting problem. Other times, it could be, e.g., the number of bits in the binary representation of two numbers whose gcd we want to find.

- Ex.: Let's add up the time for insertion sort. For each j, let t_j be the number of times the **while** loop test is executed for that value of j.

	statement	cost(time per exec.)	#times executed
1	**for** $j \leftarrow 2$ **to** length$[A]$	1	n
2	$\quad$ *key* $\leftarrow A[j]$	1	$n-1$
3	$\quad i \leftarrow j-1$	1	$n-1$
4	$\quad$ **while** $i > 0$ and $A[i] >$ *key*	1	$\sum_{j=2}^{n} t_j$
5	$\qquad A[i+1] \leftarrow A[i]$	1	$\sum_{j=2}^{n}(t_j-1)$
6	$\qquad i \leftarrow i-1$	1	$\sum_{j=2}^{n}(t_j-1)$
7	$\quad$ **endwhile**	1	$\sum_{j=2}^{n}(t_j-1)$
8	$\quad A[i+1] \leftarrow$ *key*	1	$n-1$
9	**endfor**	1	$n-1$

We assign a cost of 1 to each statement here, because each is one or a few simple RAM ops. (e.g., reading or writing a variable, addition, subtraction, etc.) unlike, e.g., exponentiation or some other complicated operation.

The total time $T(n)$ is the sum of the products of the "cost" and "times" columns. (Later we may see pseudocode with a line that cannot be counted as unit cost, e.g., "Sort the array A.")

The best-case running time is when $t_j = 1 \forall j$. Then, $T(n) = an + b$ for some constants a and b, i.e., $T(n)$ is *linear in n.*

The worst-case running time is when $t_j = j \forall j$. Then, $T(n) = an^2 + bn + c$ for some constants a, b, and c, i.e., $T(n)$ is *quadratic in n.*

- We generally concentrate on worst-case time, i.e., an upper bound on time for all inputs of size n.

  Sometimes we'll look at average-case. (Often average-case is roughly as bad as worst-case.)

- We also generally concentrate on the leading term (an^2) of a cost like $an^2 + bn + c$. Furthermore the constant (a) is not as important as the exponent on n.

- Returning to our example, we say the worst-case time for insertion sort is $\Theta(n^2)$, and the best-case time is $\Theta(n)$.

  For large enough n, a $\Theta(n^2)$ algorithm (worst-case running time) is faster than, e.g., a $\Theta(n^3)$ algorithm in the worst case.

COMP 460 Lecture Notes

R. I. Greenberg

1 Divide-and-Conquer Approach to Algorithm Design

Insertion sort used an incremental approach. Now we look at another approach in which an algorithm solves a problem by recursively calling itself to solve related subproblems.

Ex.: Merge Sort:

- Divide n-element sequence into two subsequences of $n/2$ elements each.
- Sort the two subsequences recursively using merge sort.
- Merge the two sorted subsequences to produce the answer.

The recursion "bottoms out" when we encounter a sequence of length 1; then, just return it.

We can do the merging procedure in $\Theta(n)$ time.

We can write a *recurrence* for the running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{for } n = 1 \\ \underset{\substack{\text{sorting}\\ \text{2 subarrays}}}{2T(n/2)} + \underset{\text{merging}}{\Theta(n)} & \text{for } n > 1 \end{cases}$$

(Assume for now that n is a power of 2, so we don't have to worry about uneven divisions.)

We will see $T(n)$ is $\Theta(n \lg n)$. So merge sort is better than insertion sort in the worst case.

2 Asymptotic Notation

- Functions that are $O(g(n))$ are defined by:

$$O(g(n)) = \{\, f(n) : \exists c, n_0 > 0 \text{ s.t. } 0 \le f(n) \le cg(n) \forall n \ge n_0 \,\}$$

- Functions that are $\Omega(g(n))$ are defined by:

$$\Omega(g(n)) = \{\, f(n) : \exists c, n_0 > 0 \text{ s.t. } 0 \le cg(n) \le f(n) \forall n \ge n_0 \,\}$$

- $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

- Exs. relating to insertion sort: It is easy to see that insertion sort has $O(n^2)$ worst-case running time:

  - At most n values of j, and, for each one, at most n values of i.
  - Constant amt. of work for each of the n^2 pairs of values for i and j.

  $O(n^2)$ worst-case implies $O(n^2)$ in general.

  $\Theta(n^2)$ worst-case does not imply $\Theta(n^2)$ always. (We saw $\Theta(n)$ best case.)

  Best case is $\Omega(n)$ implies always $\Omega(n)$.

  Our book will not say insertion-sort is $\Omega(n^2)$, because $\exists$ inputs for which time is $\Theta(n)$. But can still say *worst-case* running time is $\Omega(n^2)$, and many writers make such statements with "worst-case" being implicit.

- Other notations:

  $f(n)$ is $o(g(n))$ means $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ (and f, g asymp. nonneg.)

  $f(n)$ is $\omega(g(n))$ means $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ (and f, g asymp. nonneg.)

  We can also define these two notations in the style used above for O and Ω by just changing "$\exists c$" to "$\forall c$" (still with $\exists n_0$).

3 Recurrences

Three solution techniques:

- Substitution: Guess a soln., and prove it by induction.

- Recursion-tree method (referred to as "iteration" in CLR 1st ed.)

  An example appears in the text.

- Master method for recurrences of the form

$$T(n) = aT(n/b) + f(n) \qquad a \ge 1, b > 1\ .$$

  (The method also works with "$T(\lfloor n/b \rfloor)$" or "$T(\lceil n/b \rceil)$" instead of "$T(n/b)$".)

  Three cases (incorporating result of a textbook exercise into case 2):

1. If $\exists \varepsilon > 0$ such that $f(n) = O(n^{\log_b a - \varepsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \lg^k n)$, with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If $\exists \varepsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \varepsilon})$, then $T(n) = \Theta(f(n))$.

(In case 3, we also need a regularity condition, but as a practical matter, it will not be an issue for the types of functions we'll look at in this course.)

Note: Sometimes none of the 3 cases apply; then another solution method must be used.