The Design and Analysis of Algorithms

Lecture 1

Lei Wang

Dalian University of Technology

September 23, 2015

Get Started

Course Info

- Instructor: Lei Wang, lei.wang@dlut.edu.cn, tel: 6227 4394
- Textbook: **Introduction to Algorithms**, by Thomas Cormen *et al.*, aka, CLRS.
- Courseware: MIT Open Coursework
  `http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/index.htm`
- Call for volunteer TAs
- LaTeX is strongly recommended

Goals

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of "divide and conquer" in the context of merge sort.

The sorting problem

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

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

Solution

Algorithm: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Insertion sort

- A good algorithm for sorting a small number of elements.
- It works the way you might sort a hand of playing cards.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

Correctness

We often use a ***loop invariant*** to show the correctness of an algorithm.

Loop invariant for INSERTION-SORT

At the start of each iteration of the "outer" **for** loop—the loop indexed by j—the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in **sorted** order.

Three parts to prove correctness

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like mathematical induction

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.
- We can show the three parts in any order.

Example for insertion sort

The three parts for insertion sort

- **Initialization:** Just before the first iteration, $j = 2$. The subarray $A[1..j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.
- **Maintenance:** we note that the body of the inner while loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.
- **Termination:** The outer for loop ends when $j > n$; this occurs when $j = n+1$. Thus, $j-1 = n$. Plugging n in for $j-1$ in the loop invariant, $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order. The entire array is sorted!

Random-access machine (RAM) model

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers.
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling; shift left/shift right (good for multiplying/dividing by 2^k).
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return.

  Each of these instructions takes a constant amount of time.

Running time

On a particular input, ***the running time*** is the number of primitive operations (steps) executed.

- Want to define steps to be machine-independent.
- Each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line i takes the same amount of time c_i.
- This is assuming that the line consists only of primitive operations.

The running time of an algorithm is:

$$\sum_{\text{all statements}} \text{(cost of statement)} \times \text{(\# of times statement is executed)}$$

Analysis of insertion sort

Pseudocode with cost line by line

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

The running time of Insertion-Sort

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1) \end{aligned}$$

Analysis of insertion sort, (cont.)

Best case:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\ &= an + b. \end{aligned}$$

Worst case:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= an^2 + bn + c. \end{aligned}$$

Worst-case and average-case analysis

We usually concentrate on finding the ***worst-case running time***: the longest running time for ***any*** input of size n.

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it's often about as bad as the worst case.

Divide and conquer

Another common approach.

- **Divide** the problem into a number of subproblems.
- **Conquer** the subproblems by solving them recursively.
  - **Base case:** If the subproblems are small enough, just solve them by brute force.
- **Combine** the subproblem solutions to give a solution to the original problem.