Lecture Notes for Chapter 2: Getting Started

Chapter 2 overview

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.

Insertion 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 $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The sequences are typically stored in arrays.

We also refer to the numbers as ***keys***. Along with each key may be additional information, known as ***satellite data***. *[You might want to clarify that "satellite data" does not necessarily come from a satellite.]*

We will see several ways to solve the sorting problem. Each way will be expressed as an ***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.

Expressing algorithms

We express algorithms in whatever way is the clearest and most concise.

English is sometimes the best way.

When issues of control need to be made perfectly clear, we often use ***pseudocode***.

- Pseudocode is similar to C, C++, Pascal, and Java. If you know any of these languages, you should be able to understand pseudocode.
- Pseudocode is designed for *expressing algorithms to humans*. Software engineering issues of data abstraction, modularity, and error handling are often ignored.
- We sometimes embed English statements into pseudocode. Therefore, unlike for "real" programming languages, we cannot create a compiler that translates pseudocode to machine code.

Insertion sort

A good algorithm for sorting a small number of elements.

It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Pseudocode

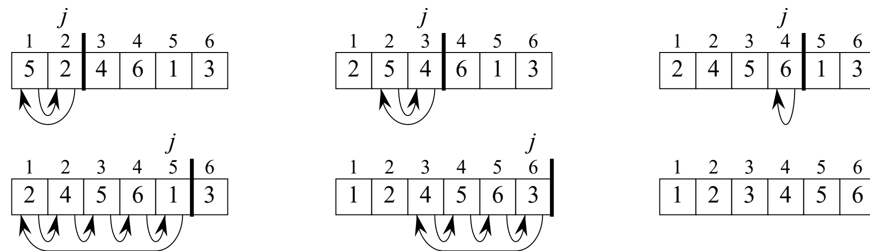
We use a procedure INSERTION-SORT.

- Takes as parameters an array $A[1 \,..\, n]$ and the length n of the array.
- As in Pascal, we use ". ." to denote a range within an array.
- *[We usually use 1-origin indexing, as we do here. There are a few places in later chapters where we use 0-origin indexing instead. If you are translating pseudocode to C, C++, or Java, which use 0-origin indexing, you need to be careful to get the indices right. One option is to adjust all index calculations in the C, C++, or Java code to compensate. An easier option is, when using an array $A[1 \,..\, n]$, to allocate the array to be one entry longer—$A[0 \,..\, n]$—and just don't use the entry at index 0.]*
- *[In the lecture notes, we indicate array lengths by parameters rather than by using the* length *attribute that is used in the book. That saves us a line of pseudocode each time. The solutions continue to use the* length *attribute.]*
- The array A is sorted ***in place***: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

```
INSERTION-SORT(A, n)                                   cost  times
 for j = 2 to n                                        c_1   n
     key = A[j]                                        c_2   n - 1
     // Insert A[j] into the sorted sequence A[1..j-1]. 0     n - 1
     i = j - 1                                         c_4   n - 1
     while i > 0 and A[i] > key                        c_5   sum_{j=2}^{n} t_j
         A[i+1] = A[i]                                 c_6   sum_{j=2}^{n} (t_j - 1)
         i = i - 1                                     c_7   sum_{j=2}^{n} (t_j - 1)
     A[i+1] = key                                      c_8   n - 1
```

[Leave this on the board, but show only the pseudocode for now. We'll put in the "cost" and "times" columns later.]

Example



[Read this figure row by row. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the "current card" being inserted into the hand. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1 \mathinner{.\,.} j]$—from the part of the array that is unaffected by this iteration—$A[j+1 \mathinner{.\,.} n]$. The last part of the figure shows the final sorted array.]

Correctness

We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer. Here's the loop invariant for INSERTION-SORT:

> **Loop invariant:** At the start of each iteration of the "outer" **for** loop—the loop indexed by j—the subarray $A[1 \mathinner{.\,.} j-1]$ consists of the elements originally in $A[1 \mathinner{.\,.} j-1]$ but in sorted order.

To use a loop invariant to prove correctness, we must show three things about it:

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.

For insertion sort

Initialization: Just before the first iteration, $j = 2$. The subarray $A[1 \mathinner{.\,.} j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead 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 $j > n$, which occurs when $j = n+1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1 \mathinner{.\,.} n]$ consists of the elements originally in $A[1 \mathinner{.\,.} n]$ but in sorted order. In other words, the entire array is sorted.

Pseudocode conventions

[Covering most, but not all, here. See book pages 20–22 for all conventions.]

- Indentation indicates block structure. Saves space and writing time.
- Looping constructs are like in C, C++, Pascal, and Java. We assume that the loop variable in a **for** loop is still defined when the loop exits (unlike in Pascal).
- **//** indicates that the remainder of the line is a comment.
- Variables are local, unless otherwise specified.
- We often use ***objects***, which have ***attributes***. For an attribute *attr* of object x, we write $x.attr$. (This notation matches $x.attr$ in Java and is equivalent to x->*attr* in C++.) Attributes can cascade, so that if $x.y$ is an object and this object has attribute *attr*, then $x.y.attr$ indicates this object's attribute. That is, $x.y.attr$ is implicitly parenthesized as $(x.y).attr$.
- Objects are treated as references, like in Java. If x and y denote objects, then the assignment $y = x$ makes x and y reference the same object. It does not cause attributes of one object to be copied to another.
- Parameters are passed by value, as in Java and C (and the default mechanism in Pascal and C++). When an object is passed by value, it is actually a reference (or pointer) that is passed; changes to the reference itself are not seen by the caller, but changes to the object's attributes are.

- The boolean operators "and" and "or" are ***short-circuiting***: if after evaluating the left-hand operand, we know the result of the expression, then we don't evaluate the right-hand operand. (If x is FALSE in "x and y" then we don't evaluate y. If x is TRUE in "x or y" then we don't evaluate y.)

Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time.

In order to predict resource requirements, we need a computational model.

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). Also, 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.

The RAM model uses integer and floating-point types.

- We don't worry about precision, although it is crucial in certain numerical applications.
- There is a limit on the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n$ is a very frequently used shorthand for $\log_2 n$.)
  - $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
  - c is a constant $\Rightarrow$ the word size cannot grow arbitrarily.

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.