Analysis - Insertion Sort

Outline

1. The Sorting Problem
2. Insertion Sort: An Incremental Strategy
3. Loop Invariants and Correctness of Insertion Sort
4. RAM Model; What do we count?
5. Analysis of Insertion Sort: Best and Worst Cases
6. Worst Case Rate of Growth and Θ (Theta)

Modeling a Problem: The Sorting Problem

Problem Formulation:

Clear and unambiguous definition of what to be solved in terms of:

- Input of the problem
- Output of the problem
- Assumptions in the problem

Descriptions in a problem formulation must be declarative (not procedural). All assumptions concerning input and output must be explicit. The problem formulation provides the requirements for an algorithm.

Problem Formulation for Sorting:

Input: A sequence σ of n real numbers xi ($1 \leq i \leq n$)

Assumptions:

1. n is a positive integer.
2. The real numbers xi ($1 \leq i \leq n$) are not necessarily distinct.

Output: A permutation $\pi = x'1\ x'2\ \ldots\ x'n$ of the given sequence σ such that $x'j \leq x'j+1$ for every j ($1 \leq j < n$)

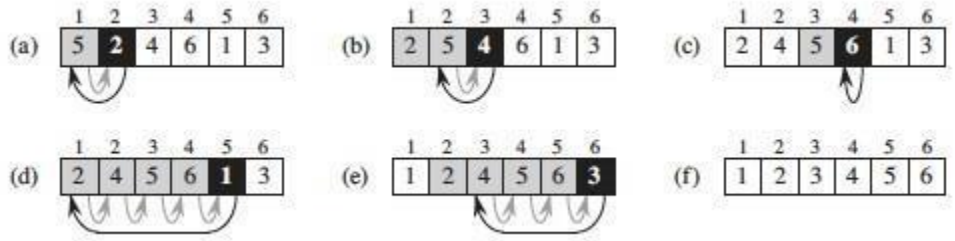
- The numbers are referred to as **keys**.
- Additional information known as **satellite data** may be associated with each key.
- Sorting is hugely important in most applications of computers. We will cover several ways to solve this problem in this course.

Insertion Sort: An Incremental Strategy:

Insertion sort takes an **incremental strategy** of problem solving: pick off one element of the problem at a time and deal with it. Our first example of the text's pseudocode:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      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          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Here's a step by step example:



Loop Invariants and Correctness of Insertion Sort

Loop Invariants:

A loop invariant is a formal property that is (claimed to be) true at the start of each iteration. We can use loop invariants to prove the correctness of iteration in programs, by showing three things about the loop invariant:

Initialization: It is true prior to the first iteration.

Maintenance: If it is true prior to a given iteration, then it remains true before the next iteration.

Termination: When the loop terminates, the invariant (and the conditions of termination) gives us a useful property that helps to show that the algorithm is correct.

Notice the similarity to mathematical induction, but here we have a termination condition.

Correctness of Insertion Sort:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      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          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop Invariant: At the start of each iteration of the outer `for` loop at line 1, the subarray A[1 .. *j*-1] consists of the elements originally in A[1 .. *j*-1] but in sorted order.

Initialization: We start with *j*=2. The subarray A[1 .. *j*-1] is the single element A[1], which is the element originally in A[1] and is trivially sorted.

Maintenance: A precise analysis would state and prove another loop invariant for

the `while` loop. For simplicity, we'll note informally that at each iteration the elements A[*j*-1], A[*j*-2], A[*j*-3], etc. are shifted to the right (so they remain in the sequence in proper order) until the proper place for key (the former occupant of A[j]) is found. Thus at the next iteration, the subarray A[1 .. j] has the same elements but in sorted order.

Termination: The outer `for` loop ends when j=n+1. Therefore j-1=n. Plugging n into the loop invariant, the subarray A[1 .. n] (which is the entire array) consists of the elements originally in A[1 .. n] but in sorted order.

RAM Model: What do we count?

If we are going to tally up time (and space) requirements, we need to know what counts as a unit of time (and space). Since computers differ from each other in details, it is helpful to have a common abstract model.

Random Access Machine (RAM) Model:

The RAM model is based on the design of typical von Neumann architecture computers that are most widely in use. For example:

- Instructions are executed one after the other (no concurrent operations).
- Instructions operate on a small number (one or two) of data "words" at a time.

- Data words are of a limited, constant size (cannot get arbitrarily large computation done in one operation by putting the data in an arbitrarily large word).

Categories of Primitive Operations:

We identify the primitive operations that count as "one step" of computation. They may differ in actual time taken, but all can be bounded by the same constant, so we can simplify things greatly by counting them as equal.

Data Manipulation:

- Arithmetic operation: +, -, *, /, remainder, floor, ceiling, left/right shift
- Comparison: $<, =, >, \leq, \geq$
- Logical operation: $\wedge, \vee, \neg$

These assume bounded size data objects being manipulated, such as integers that can be represented in a constant number of bits (e.g, a 64-bit word), bounded precision floating numbers, or boolean strings that are bounded in size. Arbitrarily large integers, arbitrarily large floating point precision, and arbitrarily long strings can lead to nonconstant growth in computation time.

Flow Control:

- Branch: case, if, etc.
- Loop; while, for __ ← ___ to ___

Here we are stating that the time to execute the machinery of the conditional loop controllers are constant time. However, if the language allows one to call arbitrary methods as part of the boolean expressions involved, the overall execution may not be constant time.

Miscellaneous:

- Assignment: ←
- Subscription: [ ]
- Reference
- Setting up a procedure or function call (see below)
- Setting up an I/O operation (see below)

The time to set up a procedure call is constant, but the time to execute the procedure may not be. Count that separately. Similarly, the time to set up an I/O operation is constant, but the time to actually read or write the data may be a function of the size of the data. Treat I/O as constant

only if you know that the data size is bounded by a constant, e.g., reading one line from a file with fixed data formats.

Input Size: Time taken is a function of input size. How do we measure input size?

- It is often most convenient to use the number of items in the input, such as the number of numbers being sorted.
- For some algorithms we need to measure the size of data, such as the number of bits in two integers being multiplied.
- For other algorithms we need more than one number, such as the number of vertices and edges in a graph.

Analysis of Insertion Sort: Best and Worst Cases

We now undertake an exhaustive quantitative analysis of insertion sort. We do this analysis in greater detail than would normally be done, to illustrate why this level of detail is not necessary!!!

For each line, what does it cost, and how many times is it executed?

We don't know the actual cost (e.g., in milliseconds) as this varies across software and hardware implementations. A useful strategy when you do not know a quantity is to just give it a name ...

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

The ci are the unknown but constant costs for each step. The tj are the numbers of times that line 5 is executed for a given j. These quantities depend on the data, so again we just give them names.

Let T(n) be the running time of insertion sort. We can compute T(n) by multiplying each cost by the number of times it is incurred (on each line) and summing across all of the lines of code:

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

Best Case: When the array is already sorted, we always find that A[*i*] ≤ *key* the first time the while loop is run; so all *tj* are 1 and *tj-1* are 0. Substituting these values into the above:

5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$

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

As shown in the second line, this is the same as *an* + *b* for suitable constants *a* and *b*. Thus the running time is a **linear function of n.**

Worst Case: When the array is in reverse sorted order, we always find that A[*i*] > *key* in the while loop, and will need to compare *key* to all of the (growing) list of elements to the left of *j*. There are *j*-1 elements to compare to, and one additional test for loop exit. Thus, *tj=j*.

5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$

$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.

$\sum_{j=1}^{n} j$ is known as an *arithmetic series*, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.

Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

Plugging those values into our equation:

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

We get the worst case running time, which we simplify to gather constants:

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

T(n) can be expressed as $an^2 + bn + c$ for some *a*, *b*, *c*: *T(n)* is a **quadratic function of n**.

So we can draw these conclusions purely from mathematical analysis, with *no implementation or testing needed*: Insertion sort is very quick (linear) on already sorted data, so it works well when incrementally adding items to an existing list. But the worst case is slow for reverse sorted data.

Worst Case Rate of Growth and Θ (Theta): From the above example we introduce two key ideas and a notation that will be elaborated on later.

Worst Case Analysis:

Above, both best and worst case scenarios were analyzed. We usually concentrate on the worst-case running times for algorithms, because:

- This gives us a guaranteed upper bound.
- For some algorithms, the worst case occurs often (such as failing to find an item in a search).
- The average is often almost as bad as the worst case.

How long does it take on average to successfully find an item in an unsorted list of n items?

How long does it take in the worst case, when the item is not in the list?
What is the difference between the two?

Rate of Growth:

In the above example, we kept track of unknown but named constant values for the time required to execute each line once. In the end, we argued that these constants don't matter

- Their specific values don't matter because they all add up to summary constants in the equations (e.g., a and b).
- Even their presence does not matter, because it is the growth of the function of n that dominates the time taken to run the algorithm.

This is good news, because it means that all of that excruciating detail is not needed!

Furthermore, only the fastest growing term matters. In $an^2 + bn + c$, the growth of n^2 dominates all the other terms (including bn) in its growth.

Theta: Θ

We will use Θ notation to concentrate on the fastest growing term and ignore constants.

If we conclude that an algorithm requires $an^2 + bn + c$ steps to run, we will dispense with the constants and lower order terms and say that its growth rate (the growth of how long it takes as n grows) is $\Theta(n^2)$.

If we see $bn + c$ we will write $\Theta(n)$.

A simple constant c will be $\Theta(1)$, since it grows the same as the constant 1.

When we combine Θ terms, we similarly attend only to the dominant term. For example, suppose an analysis shows that the first part of an algorithm requires $\Theta(n^2)$ timeand the second part requires $\Theta(n)$ time. Since the former term dominates, we need not write $\Theta(n^2 + n)$: the overall algorithm is $\Theta(n^2)$