Introduction to Algorithms Fourth Edition

A notebook and summary

Michael Obernhumer

15.8.2024

Written by:

> Thomas H. Cormen
> Charles E. Leiserson
> Ronald L. Rivest
> Clifford Stein

Content

1 Foundations

2 The Role of Algorithms in Computing

2.1 Algorithms

2.1.1 Exercises:

2.1.1.1 1.1-1: Describe your own real-world example that requires sorting. Describe one that requires ûnding the shortest distance between two points.

2.1.1.2 1.1-2: Other than speed, what other measures of efûciency might you need to consider in a real-world setting?

2.1.1.3 1.1-3: Select a data structure that you have seen, and discuss its strengths and limitations.

2.1.1.4 1.1-4: How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

2.1.1.5 1.1-5: Suggest a real-world problem in which only the best solution will do. Then come up with one in which approximately the best solution is good enough.

2.1.1.6 1.1-6: Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

2.2 Algorithms as a technology

- Total system performance depends on choosing eficient algorithms as much as on choosing fast hardware

2.2.1 Exercises

2.2.1.1 1.2-1: Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

2.2.1.2 1.2-2: Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64n \log n$ steps. For which values of n does insertion sort beat merge sort?

2.2.1.3 1.2-3: What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

3 Getting Started

3.1 Insertion sort

3.1.1 The Algorithm

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

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

- The numbers to be sorted are also known as the keys.
- When we want to sort numbers, it's often because they are the keys associated with other data, which we call satellite data. Together, a key and satellite data form a record.

INSERTION-SORT $(A; n)$**:**

```
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Figure 1: insertion_sort_alg

Visualized:

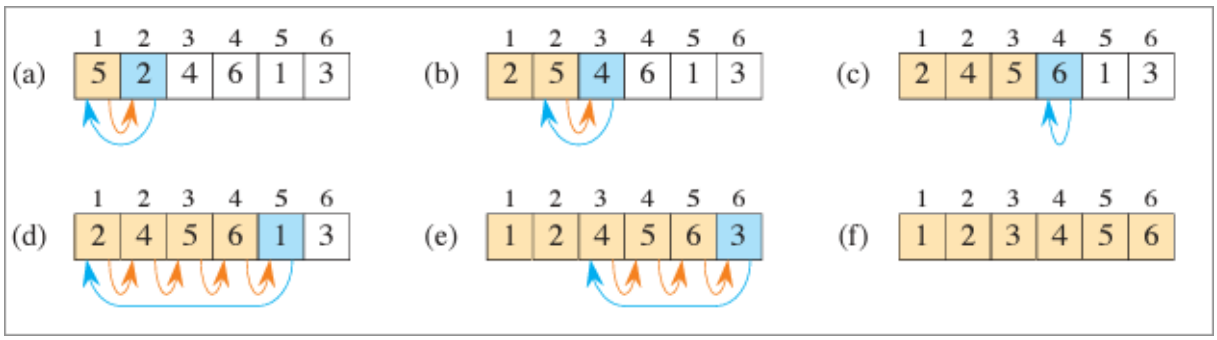

Figure 2: insertion_sort_visualized

3.1.2 What are loop invariants

Loop invariants help us understand why an algorithm is correct. When you're using a loop invariant, you need to show three things:

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: The loop terminates, and when it 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.

3.1.3 Insertion sort loop invariant

Initialization:

We start by showing that the loop invariant holds before the first loop iteration, when $i = 2$. The subarray $A[1 : i - 1]$ consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (after all, how could a subarray with just one value not be sorted?), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance:

Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving the values in $A[i - 1], A[i - 2], A[i - 3]$ and so on by one position to the right until it finds the proper position for $A[i]$ (lines 4-7), at which point it inserts the value of $A[i]$ (line 8). The subarray $A[1 : i]$ then consists of the elements originally in $A[1 : i]$, but in sorted order. `Incrementing` i (increasing its value by 1) for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5-7. Let's not get bogged down in such formalism just yet. Instead, we'll rely on our informal analysis to show that the second property holds for the outer loop.

Termination:

Finally, we examine loop termination. The loop variable i starts at 2 and increases by 1 in each iteration. Once $i's$ value exceeds n in line 1, the loop terminates. That is, the loop terminates once i equals $n+1$. Substituting $n+1$ for i in the wording of the loop invariant yields that the subarray $A[1:n]$ consists of the elements originally in $A[1:n]$, but in sorted order. Hence, the algorithm is correct.

3.1.4 Pseudocode conventions

Although many programming languages enforce 0-origin indexing for arrays (0 is the smallest valid index), we choose whichever indexing scheme is clearest for human readers to understand. Because people usually start counting at 1, not 0, most - but not all - of the arrays in this book use 1-origin indexing. To be clear about whether a particular algorithm assumes 0-origin or 1-origin indexing, we'll specify the bounds of the arrays explicitly.

3.1.5 Exercises

3.1.5.1 2.1-1: Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58\rangle$.

3.1.5.2 2.1-2: Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1:n]$.

```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

Figure 3: SUM-ARRAY

3.1.5.3 2.1-3: Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

3.1.5.4 2.1-4:

Consider the searching problem:

Input: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$ stored in array $A[1 : n]$ and a value x .

Output: An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A .

Write pseudocode for linear search, which scans through the array from beginning to end, looking for x . Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulûlls the three necessary properties.

3.1.5.5 2.1-5:

Consider the problem of adding two n-bit binary integers a and b , stored in two n-element arrays $A[0 : n-1]$ and $B[0 : n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] * 2^i$, and $b = \sum_{i=0}^{n-1} B[i] * 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0 : n]$, where $c = \sum_{i=0}^{n} C[i] * 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

3.2 Analyzing algorithms

3.3 2.3 Designing algorithms