Chapter 2 Getting Started

2.1 Insertion Sort

Exercise 2.1-1

Using Figure 2.2 as a model, illustrate the operation of InsertionSort on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Answer:

$$\begin{array}{cccccc}
31 & 41 & 59 & 26 & 41 & 58 \\
31 & \mathbf{41} & 59 & 26 & 41 & 58 \\
31 & 41 & \mathbf{59} & 26 & 41 & 58 \\
\mathbf{26} & 31 & 41 & 59 & 41 & 58 \\
26 & 31 & 41 & \mathbf{41} & 59 & 58 \\
26 & 31 & 41 & 41 & \mathbf{58} & 59
\end{array}$$

Exercise 2.1-2

Rewrite the InsertionSort procedure to sort into nonincreasing instead of nondecreasing order.

Answer:

(See `+srt/insertionsort_dec.m`.)

Exercise 2.1-3

Consider the ***searching problem***:

- **Input**: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.
- **Output**: An index i such that $v = A[i]$ or the special value nil if v does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Answer:

(See `+srch/linear_search.m`.)

loop invariant: the subarray $A[1..i-1]$ doesn't contain the value v.

Exercise 2.1-4

Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$-element array C. State the problem formally and write pseudocode for adding two integers.

Answer:

(See `+arthm/binary_add.m`.)

2.2 Analyzing algorithms

Exercise 2.2-1

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

Answer:

$$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$$

Exercise 2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case runnning times of selection sort in Θ-notation.

Answer:*

(See `+srt/selectionsort.m`.)

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1..j-1]$ consists of the $j-1$ smallest elements in the array $A[1..n]$, and this subarray is in sorted order. After the first $n-1$ elements, the subarray $A[1..n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.

Exercise 2.2-3

Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

Answer:

$$\begin{aligned} \frac{n+1}{2} &= \Theta(n) \\ n &= \Theta(n) \end{aligned}$$

Exercise 2.2-4

How can we modify almost any algorithm to have a good best-case running time?

Answer:*

Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

2.3 Designing algorithms

Exercise 2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

Answer:

$$\begin{array}{cccccccc} 3 & 9 & 26 & 38 & 41 & 49 & 52 & 57 \\ 3 & 26 & 41 & 52 & 9 & 38 & 49 & 57 \\ 3 & 41 & 26 & 52 & 38 & 57 & 9 & 49 \\ 3 & 41 & 52 & 26 & 38 & 57 & 9 & 49 \end{array}$$

Exercise 2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

Answer:

(See `+srt/mergesort_alt.m`.)

Exercise 2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Answer:*

The base case is when $n = 2$, and we have $n \lg n = 2 \lg 2 = 2 \cdot 1 = 2$.

For the inductive step, our inductive hypothesis is that $T(n/2) = (n/2) \lg(n/2) + n$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2) \lg(n/2) + n \\ &= n(\lg n - 1) + n \\ &= n \lg n - n + n \\ &= n \lg n, \end{aligned}$$

which completes the inductive proof for exact powers of 2.

Exercise 2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrrence for the running time of this recursive version of insertion sort.

Answer:*

Since it takes $\Theta(n)$ time in the worst case to insert $A[n]$ into the sorted array $A[1..n-1]$, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Although the exercise does not ask you to solve this recurrence, its solution is $T(n) = \Theta(n^2)$.

Exercise 2.3-5

Referring back to the searching problem (Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Answer:*

Procedure BinarySearch takes a sorted array A, a value v, and a range $[low..high]$ of the array, in which we search for the value . The procedure compares to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or nil if no entry of $A[low..high]$ contains the value . The initial call to either version should have the parameters $A, v, 1, n$.

(See `+srch/binary_search.m and +srch/binary_search_rec.m`.)

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $low > high$) and terminate it successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

Exercise 2.3-6

Observe that the **while** loop of lines 5–7 of the InsertionSort procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Answer:*

The **while** loop of lines 5–7 of procedure InsertionSort scans backward through the sorted array $A[1..j-1]$ to find the appropriate place for $A[j]$. The hitch is that the loop not only searches for the proper place for $A[j]$, but that it also moves each of the array elements that are bigger than $A[j]$ one position to the right (line 6). These movements can take as much as $\Theta(j)$ time, which occurs when all the $j-1$ elements preceding $A[j]$ are larger than $A[j]$. We can use binary search to improve the running time of the search to $\Theta(\lg j)$, but binary search will have no effect on the running time of moving the elements. Therefore, binary search alone cannot improve the worst-case running time of InsertionSort to $\Theta(n \lg n)$.

Exercise 2.3-7 ★

Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Answer:*

The following algorithm solves the problem:

1. Sort the elements in S.
2. Form the set $S' = \{z : z = x - y \text{ for some } y \in S\}$.
3. Sort the elements in S'.
4. Merge the two sorted sets S and S'.
5. There exist two elements in S whose sum is exactly x if and only if the same value appears in consecutive positions in the merged output.

To justify the claim in step 4, first observe that if any value appears twice in the merged output, it must appear in consecutive positions. Thus, we can restate the condition in step 5 as there exist two elements in S whose sum is exactly x if and only if the same value appears twice in the merged output.

Suppose that some value w appears twice. Then w appeared once in S and once in S'. Because w appeared in S', there exists some $y \in S$ such that $w = x - y$, or $x = w + y$. Since $w \in S$, the elements w and y are in S and sum to x.

Conversely, suppose that there are values $w, y \in S$ such that $w + y = x$. Then, since $x - y = w$, the value w appears in S'. Thus, w is in both S and S', and so it will appear twice in the merged output.

Steps 1 and 3 require $\Theta(n \lg n)$ steps. Steps 2, 4, 5, and 6 require $O(n)$ steps. Thus the overall running time is $O(n \lg n)$.

A reader submitted a simpler solution that also runs in $\Theta(n \lg n)$ time. First, sort the elements in S, taking $\Theta(n \lg n)$ time. Then, for each element y in S, perform a binary search in S for $x - y$. Each binary search takes $O(\lg n)$ time, and there are at most n of them, and so the time for all the binary searches is $O(n \lg n)$. The overall running time is $O(n \lg n)$.

Another reader pointed out that since S is a set, if the value $x/2$ appears in S, it appears in S just once, and so $x/2$ cannot be a solution.

Problems

Problem 2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should we choose k in practice?

Answer:*

a.

Insertion sort takes $\Theta(k^2)$ time per k-element list in the worst case. Therefore, sorting n/k lists of k elements each takes $\Theta(k^2 n/k) = \Theta(nk)$ worst-case time.

b.

Just extending the 2-list merge to merge all the lists at once would take $\Theta(n \cdot (n/k)) = \Theta(n^2/k)$ time (n from copying each element once into the result list, n/k from examining n/k lists at each step to select next item for result list).

To achieve $\Theta(n \lg(n/k))$-time merging, we merge the lists pairwise, then merge the resulting lists pairwise, and so on, until there's just one list. The pairwise merging requires $\Theta(n)$ work at each level, since we are still working on n elements, even if they are partitioned among sublists. The number of levels, starting with n/k lists (with k elements each) and finishing with 1 list (with n elements), is $\lceil \lg(n/k) \rceil$. Therefore, the total running time for the merging is $\Theta(n \lg(n/k))$.

c.

The modified algorithm has the same asymptotic running time as standard merge sort when $\Theta(nk + n \lg(n/k)) = \Theta(n \lg n)$. The largest asymptotic value of k as a function of n that satisfies this condition is $k = \Theta(\lg n)$.

To see why, first observe that k cannot be more than $\Theta(\lg n)$ (i.e., it can't have a higher-order term than $\lg n$), for otherwise the left-hand expression wouldn't be $\Theta(n \lg n)$ (because it would have a higher-order term than $\Theta(n \lg n)$). So all we need to do is verify that $k = \Theta(\lg n)$ works, which we can do by plugging $k = \Theta(\lg n)$ into $\Theta(nk + n \lg(n/k)) = \Theta(nk + n \lg n - n \lg k)$ to get

$\Theta(n \lg n + n \lg n - n \lg \lg n) = \Theta(2n \lg n - n \lg \lg n)$,

which, by taking just the high-order term and ignoring the constant coefficient, equals $\Theta(n \lg n)$.

d.

In practice, k should be the largest list length on which insertion sort is faster than merge sort.

Problem 2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

(See `+sort/bubblesort.m`)

a. Let A' denote the output of $\text{Bubblesort}(A)$. To prove that Bubblesort is correct, we need to prove that it terminates and that

$A'[1] \le A'[2] \le \cdots \le A'[n]$,

where $n = A.length$. In order to show that Bubblesort actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

b. State precisely a loop invariant for the $\textbf{for}$ loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the $\textbf{for}$ loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Answer:*

a.

We need to show that the elements of A' form a permutation of the elements of A.

b.

Loop invariant: At the start of each iteration of the $\textbf{for}$ loop of lines 2–4, $A[j] = \min\{A[k] : j \le k \le n\}$ and the subarray $A[j..n]$ is a permutation of the values that were in $A[j..n]$ at the time that the loop started.

- **Initialization:** Initially, $j = n$, and the subarray $A[j..n]$ consists of single element $A[n]$. The loop invariant trivially holds.
- **Maintenance:** Consider an iteration for a given value of j. By the loop invariant, $A[j]$ is the smallest value in $A[j..n]$. Lines 3–4 exchange $A[j]$ and $A[j-1]$ if $A[j]$ is less than $A[j-1]$, and so $A[j-1]$ will be the smallest value in $A[j-1..n]$ afterward. Since the only change to the subarray $A[j-1..n]$ is this possible exchange, and the subarray $A[j..n]$ is a permutation of the values that were in $A[j..n]$ at the time that the loop started, we see that $A[j-1..n]$ is a permutation of the values that were in $A[j-1..n]$ at the time that the loop started. Decrementing j for the next iteration maintains the invariant.
- **Termination:** The loop terminates when j reaches i. By the statement of the loop invariant, $A[i] = \min\{A[k] : i \le k \le n\}$ and $A[i..n]$ is a permutation of the values that were in $A[i..n]$ at the time that the loop started.

c.

Loop invariant: At the start of each iteration of the $\textbf{for}$ loop of lines 1–4, the subarray $A[1..i-1]$ consists of the $i-1$ smallest values originally in $A[1..n]$, in sorted order, and $A[i..n]$ consists of the $n-i+1$ remaining values originally in $A[1..n]$.

- **Initialization:** Before the first iteration of the loop, $i = 1$. The subarray $A[1..i-1]$ is empty, and so the loop invariant vacuously holds.
- **Maintenance:** Consider an iteration for a given value of i. By the loop invariant, $A[1..i-1]$ consists of the i smallest values in $A[1..n]$, in sorted order. Part (b) showed that after executing the **for** loop of lines 2–4, $A[i]$ is the smallest value in $A[i..n]$, and so $A[1..i]$ is now the i smallest values originally in $A[1..n]$, in sorted order. Moreover, since the **for** loop of lines 2–4 permutes $A[i..n]$, the subarray $A[i+1..n]$ consists of the $n-i$ remaining values originally in $A[1..n]$.
- **Termination:** The **for** loop of lines 1–4 terminates when $i = n$, so that $i - 1 = n - 1$. By the statement of the loop invariant, $A[1..i-1]$ is the subarray $A[1..n-1]$, and it consists of the $n-1$ smallest values originally in $A[1..n]$, in sorted order. The remaining element must be the largest value in $A[1..n]$, and it is in $A[n]$. Therefore, the entire array $A[1..n]$ is sorted.

d.

The running time depends on the number of iterations of the **for** loop of lines 2–4. For a given value of i, this loop makes $n - i$ iterations, and i takes on the values $1, 2, \ldots, n-1$. The total number of iterations, therefore, is

$$\begin{aligned}
\sum_{i=1}^{n-1} &= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= n(n-2) - \frac{n(n-1)}{2} \\
&= \frac{n(n-1)}{2} \\
&= \frac{n^2}{2} - \frac{n}{2}.
\end{aligned}$$

Thus, the running time of bubblesort is $\Theta(n^2)$ in all cases. The worst-case running time is the same as that of insertion sort.

Problem 2-3 Correctness of Horner's rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned}
P(x) &= \sum_{k=0}^{n} a_k x^k \\
&= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)).
\end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

(See `+arthm/polynomial.m`).

a. In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

c. Consider the following loop invariant:

- At the start of each iteration of the **for** loop of lines 2–3,
- $y = \displaystyle\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

Answer:

a.

$\Theta(n)$

b.

(See `+arthm/polynomial_bf.m`.)

$\Theta(n^2)$

Problem 2-4 Inversions

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint*: Modify merge sort.)

Answer:*

a.

The inversions are $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$. (Remember that inversions are specified by indices rather than by the values in the array.)

b.

The array with elements from $\{1, 2, \ldots, n\}$ with the most inversions is $\langle n, n-1, n-2, \ldots, 2, 1 \rangle$. For all $1 \le i < j \le n$, there is an inversion (i, j). The number of such inversions is $\binom{n}{2} = n(n-1)/2$.

c.

Suppose that the array A starts out with an inversion (k, j). Then $k < j$ and $A[k] > A[j]$. At the time that the outer **for** loop of lines 1–8 sets $key = A[j]$, the value that started in $A[k]$ is still somewhere to the left of $A[j]$. That is, it's in $A[i]$, where $1 \le i < j$, and so the inversion has become (i, j). Some iteration of the **while** loop of lines 5–7 moves $A[i]$ one position to the right. Line 8 will eventually drop key to the left of this element, thus eliminating the inversion. Because line 5 moves only elements that are greater than key, it moves only elements that correspond to inversions. In other words, each iteration of the **while** loop of lines 5–7 corresponds to the elimination of one inversion.

d.

We follow the hint and modify merge sort to count the number of inversions in $\Theta(n \lg n)$ time.

To start, let us define a ***merge-inversion*** as a situation within the execution of merge sort in which the Merge procedure, after copying $A[p..q]$ to L and $A[q+1..r]$ to R, has values x in L and y in R such that $x > y$. Consider an inversion (i, j), and let $x = A[i]$ and $y = A[j]$, so that $i < j$ and $x > y$. We claim that if we were to run merge sort, there would be exactly one merge-inversion involving x and y. To see why, observe that the only way in which array elements change their positions is within the Merge procedure. Moreover, since Merge keeps elements within L in the same relative order to each other, and correspondingly for R, the only way in which two elements can change their ordering relative to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one merge-inversion involving x and y. To see that there is exactly one such merge-inversion, observe that after any call of Merge that involves both x and y, they are in the same sorted subarray and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim.

We have shown that every inversion implies one merge-inversion. In fact, the correspondence between inversions and merge-inversions is one-to-one. Suppose we have a merge-inversion involving values x and y, where x originally was $A[i]$ and y was originally $A[j]$. Since we have a merge-inversion, $x > y$. And since x is in L and y is in R, x must be within a subarray preceding the subarray containing y. Therefore x started out in a position i preceding y's original position j, and so (i, j) is an inversion.

Having shown a one-to-one correspondence between inversions and merge-inversions, it suffices for us to count merge-inversions.

Consider a merge-inversion involving y in R. Let z be the smallest value in L that is greater than y. At some point during the merging process, z and y will be the "exposed" values in L and R, i.e., we will have $z = L[i]$ and $y = R[j]$ in line 13 of Merge. At that time, there will be merge-inversions involving y and $L[i], L[i+1], L[i+2], \ldots, L[n_1]$, and these $n_1 - i + 1$ merge-inversions will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the Merge procedure and add the value of $n_1 - i + 1$ at that time to our total count of merge-inversions.

The following pseudocode, modeled on merge sort, works as we have just described. It also sorts the array A.

(See `+srt/inversion.m`.)

The initial call is $\text{CountInversions(A, 1, n)}$.

In MergeInversions, whenever $R[j]$ is exposed and a value greater than $R[j]$ becomes exposed in the L array, we increase $\mathit{inversions}$ by the number of remaining elements in L. Then because $R[j+1]$ becomes exposed, $R[j]$ can never be exposed again. We don't have to worry about merge-inversions involving the sentinel ∞ in R, since no value in L will be greater than ∞.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last **for** loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: $\Theta(n \lg n)$.