Introduction to Algorithms - Solutions

3rd Edition

Purbayan Chowdhury

October 4, 2020

Contents

Chapter 1

The Role of Algorithms in Computing

Chapter 2

Getting Started

2.1 Insertion Sort

Ex 2.1-1

Ex 2.1-2 Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

A.

Algorithm 1: Non-increasingInsertionSort

```
Input  : A ⟵ Unsorted Array
Output: A ⟵ Array Sorted in Non-increasing Order
for j ⟵ 1 to A.length − 1 do
    key ⟵ A[j]
    /* Insert A[j] into the sorted sequence A[1..j-1]      */
    i ⟵ j − 1
    while i ≥ 0 and A[i] > key do
        A[i + 1] ⟵ A[i]
        i ⟵ i − 1
    end
    A[i + 1] ⟵ key
end
```

Ex 2.1-3

Ex 2.1-4 Consider the searching problem: Input: A sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$ 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 fulfils the three necessary properties.

A.

Algorithm 2: Linear-Search

```
Input  : A ⟵ Array
         v ⟵ value to be searched
Output: i ⟵ index of the value if found, else NIL
i ⟵ NIL
for j ⟵ 0 to A.length − 1 do
    if A[j] = v then
        i ⟵ j
        break
    end
end
return i
```

Ex. 2.1-5 **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 the two integers.**

A.

Algorithm 3: n-bitBinaryAddition

```
Input  : A ⟵ First Array
         B ⟵ Second Array
Output: C ⟵ Binary Addition Result
carry ⟵ 0
for i ⟵ n − 1 downto 0 do
    C[i + 1] ⟵ (A[i] + B[i] + carry)(mod2)
    if A[i] + B[i] + carry ≥ 2 then
        carry ⟵ 1
    end
    else
        carry ⟵ 0
    end
end
C[0] ⟵ carry
```

2.2 Analyzing algorithms

Ex 2.2-1

Ex 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 running times of selection sort in Θ-notation.

A.

Algorithm 4: SelectionSort

```
  Input  : A ⟵ Unsorted Array
  Output: A ⟵ Array Sorted in Increasing Order
1 for i ⟵ 0 to n − 1 do
2 |  min ⟵ i
3 |  for j ⟵ i + 1 to n do
  |  |  /* Find the index of the ith smallest element       */
4 |  |  if A[j] < A[min] then
5 |  |  |  min ⟵ j
6 |  |  end
7 |  end
8 |  Swap A[min] and A[i]
9 end
```

The loop invariant of selection sort is as follows:
At each iteration of the for loop of lines 1 through 9, the subarray $A[0\ldots i-1]$ contains the $i-1$ smallest elements of A in increasing order. After $n-1$ iterations of the loop, the $n-1$ smallest elements of A are in the first $n-1$ positions of A in increasing order so the nth element is necessarily the largest amount.
The best-case and worst-case running times of selection sort are $\Theta(n^2)$, this is because regardless of how the elements are initially arranged, on the i-th iteration of the for loop in line 1, always inspects each of the remaining $n-i$ elements to find the smallest one remaining.
This yields a running
$\sum_{i=1}^{n-1} n-i = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2-n}{2} = \frac{n^2-n}{2} = \Theta(n^2)$

Ex 2.2-3

Ex 2.2-4

2.3 Designing algorithms

Ex 2.3.1 Illustrate the operation of merge sort on the array A=(3, 41, 52, 26, 38, 57, 9, 49)

Ex 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.

A.

Algorithm 5: MergeSort

```
Input  : A ⟵ Unsorted Array
         p ⟵ start index
         q ⟵ middle index
         r ⟵ end index
Output: A ⟵ Array Sorted in Increasing Order
n1 ⟵ q − p + 1
n2 ⟵ r − q
let L[1, ..., n1] and R[1, ..., n2] be new arrays
for i ⟵ 0 to n1 − 1 do
|   L[i] ⟵ A[p + i]
end
for j ⟵ 0 to n2 − 1 do
|   R[j] ⟵ A[q + j + i]
end
i ⟵ 0
j ⟵ 0
k ⟵ p
while i ≠ n1 and j ≠ n2 do
|   if L[i] ≤ R[j] then
|   |   A[k] ⟵ L[i]
|   |   i ⟵ i + 1
|   end
|   else
|   |   A[k] ⟵ R[j]
|   |   j ⟵ j + 1
|   end
|   k ⟵ k + 1
end
if i = n1 then
|   for m ⟵ j to n2 − 1 do
|   |   A[k] ⟵ R[m]
|   |   k ⟵ k + 1
|   end
end
if j = j1 then
|   for m ⟵ j to n1 − 1 do
|   |   A[k] ⟵ L[m]
|   |   k ⟵ k + 1
|   end
end
```

Ex 2.3-3 Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

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

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

Ex 2.3-4 We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \ldots n]$, we recursively sort $A[1 \ldots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \ldots n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

A. Let $T(n)$ be running time for insertion sort on an array of size n.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ T(n-1) + I(n) & \text{otherwise} \end{cases}$$

where $I(n)$ denotes the amount of time taken to insert $A[n]$ into the sorted array $A[1 \ldots n-1]$. Since we have to shift as many as $n-1$ elements once we find the correct place to insert $A[n]$, we have $I(n) = \Theta(n)$.

Ex 2.3-5 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 $\Theta(\lg n)$.

A.

Algorithm 6: RecursiveBinarySearch

Input : A ⟵ Sorted Array
a ⟵ start index
b ⟵ end index
v ⟵ value to be searched

Output: i ⟵ index of the value if found, else NIL

if $a > b$ **then**
  return NIL
end
$m \longleftarrow \lfloor \frac{a+b}{2} \rfloor$
if $A[m] = v$ **then**
  return m
end
if $A[m] < v$ **then**
  return RecursiveBinarySearch(a, m, v)
end
return RecursiveBinarySearch(m+1,b,v)

After the initial of RecursiveBinarySearch(A,0,n,v), each call results a constant number of operations and a call to a problem instance where $b-a$ is a factor of $\frac{1}{2}$. So the recurrence relation satisfies $T(n) = T(n/2) + c$. So, $T(n) \in \Theta(\lg(n))$.

Ex 2.3-6 Observe that the while loop of lines 5-7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1 \ldots j-1]$. Can we use a binary search instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Ex 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.

A. Use Merge Sort to sort the array S in time $\Theta(n \lg n)$.

Algorithm 7: FindSum

```
Input  : S ⟵ Array of n integers
         x ⟵ sum to be found
Output: If found return true, else false
i ⟵ 0
j ⟵ n
while i < j do
    if S[i] + S[j] = x then
        return true
    end
    if S[i] + S[j] < x then
        i ⟵ i + 1
    end
    if S[i] + S[j] > x then
        j ⟵ j − 1
    end
end
return false
```

Problems

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.**

A. Time for insertion sort to sort a single list of length k is $\Theta(k^2)$, so n/k of them will take $\Theta(\frac{n}{k}k^2) = \Theta(nk)$.

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

A. Provided coarseness k, we can start usual merging procedure starting at the level in which array has a size at most k. So the depth of merge recursion tree is $\lg(n) - \lg(k) = lg(n/k)$. Each level of merging is cn, so the total merging takes $\Theta(n \lg(n/k))$.

(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?**

A. Considering k as a function of n, $k(n) \in O(\log(n))$, gives the same asymptotics and for any constant choice of k, the asymptotics are the same.

(d) **How should we choose k in practice?**

A. We optimize the expression to get $c_1 n - n(c_2) = 0$ where c_1 and c_2 are coefficients of nk and $n \lg(n/k)$. A constant choice of k is optimal, in particular.

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.

(a) **Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that**

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n], \tag{2.1}$$

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).

A. We need to prove that A 0 contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

(b) **State precisely a loop invariant for the 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.**

A. At the start of each iteration, the position of the smallest element of $A[i \ldots n]$ is at most j, it's true prior to first iteration where the position of any element is at most $A.length$. To see that each iteration maintains the loop invariant, suppose that $j = k$ and the position of the smallest element of $A[i \ldots n]$ is at most k, then we compare $A[k]$ to $A[k-1]$. If $A[k] < A[k-1]$ then $A[k-1]$ is not the smallest element of $A[i \ldots n]$, so when we swap $A[k]$ and $A[k-1]$ we know that the smallest element of $A[i \ldots n]$ must occur in the first $k-1$ positions of the subarray, the maintaining the invariant. On the other hand, if $A[k] \geq A[k-1]$ then the smallest element can't be $A[k]$. Since we do nothing, we conclude that the smallest element has position at most $k-1$. Upon termination, the smallest element of $A[i \ldots n]$ is in position i.

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

A. At the start of each iteration the subarray $A[1..i-1]$ contains the $i-1$ smallest elements of A in sorted order. Prior to the first iteration $i = 1$, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop invariant, fixing i and suppose that $A[1 \ldots i-1]$ contains the $i-1$ smallest elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest element of $A[i \ldots n]$ is in position i. Since the $i-1$ smallest elements of A are already in $A[1 \ldots i-1]$, $A[i]$ must be the ith smallest element of A. Therefore $A[1 \ldots i]$ contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, $A[1 \ldots n]$ contains the n elements of A in sorted order as desired.

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

A. The ith iteration of the for loop of lines 1 through 4 will cause $n-i$ iterations of the for loop of lines 2 through 4, each with constant time execution so the worst-case running time is $\Theta(n^2)$. This is the same as insertion sort; however bubble sort also has best-case running time $\Theta(n^2)$ whereas insertion sort has best-case running time $\Theta(n)$.

2-3 ***Correctness of Horner's rule***
The following code fragment implements Horner's rule for evaluating a polynomial

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

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

```
1 y ⟵ 0
2 for i = n downto 0 do
3     y_i = a_i + x.y
```

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

A. Assuming the arithmetic function is executed in constant time, then since the loop is being executed n times, it has runtime $\Theta(n)$.

(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?**

```
y ⟵ 0
for i ⟵ 0 to n do
    y_i ⟵ 1
    for j ⟵ 1 to i do
        y_i ⟵ y_i * x
    y = y + a_i.y_i
```

A. The code has runtime $\Theta(n^2)$ as it has two nested for loops each running in linear time. It's slower than Horner's rule.

(c) **Consider the following loop invariant:**
At the start of each iteration of the for loop of lines 2–3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

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

A. Initially $i = n$, so the upper bound of the summation is -1, so the sum evaluates to 0, which is the value of y. Assume that it is true for an i, then

$$\begin{aligned} y &= a_i + x \sum_{k=0}^{n(i+1)} a_{k+i+1}x^k \\ &= a_i + x \sum_{k=1}^{n-i} a_{k+i}x^{k-1} \\ &= \sum_{k=0}^{n-i} a_{k+i}x^k \end{aligned}$$

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

A. As stated in the previous problem, we evaluated the algorithm $\sum_{k=0}^{n} a_k x^k$ and the value of the polynomial evaluated at x.

2-4 ***Inversions***
Let $A[1 \ldots n]$ **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** $(2, 3, 8, 6, 1)$

A. The five inversions are (2, 1), (3, 1), (8, 6), (8, 1), and (6, 1).

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

A. The n-element array with the most inversions is $(n, n-1, \ldots, 2, 1)$. It has $n - 1 + n - 2 + ... + 2 + 1 = n(n-1)/2$ inversions.

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

A. The running time is a constant times the no of inversions. Let $I(i)$ denote the number of $j < i$ such that $A[j] > A[i]$, and $\sum_{n}^{i=1} I(i)$ equals the number of inversions in A. Considering the while loop in the insertion sort algorithm, the loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $I(j)$ times. We reach the while loop once for each iteration in the for loop, so the no of constant time steps of insertion sort is $\sum_{n}^{i=1} I(i)$ times of the inversion number of A.

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

Algorithm 8: Inversions

Input : A $\longleftarrow$ Unsorted array
p $\longleftarrow$ start index
r $\longleftarrow$ end index
Output: inv $\longleftarrow$ No of inversions in the array

if $p < r$ **then**
A. $\quad q \longleftarrow \lfloor (p+r)/2 \rfloor$
$\quad left \longleftarrow$ Inversions(A, p, q)
$\quad right \longleftarrow$ Inversions$(A, q+1, r)$
$\quad inv \longleftarrow$ CountInversions$(A, p, q, r)$$+ left + right$
$\quad$ **return** inv
end
return 0

Algorithm 9: CountInversions

```
Input  : A ⟵ Unsorted array
         p ⟵ start index
         q ⟵ middle index
         r ⟵ end index
Output: inv ⟵ No of inversions in the array
inv ⟵ 0
n1 ⟵ q − p + 1
n2 ⟵ r − q
let L[1, ..., n1] and R[1, ..., n2] be new arrays
for i ⟵ 0 to n1 − 1 do
    L[i] ⟵ A[p + i]
end
for j ⟵ 0 to n2 − 1 do
    R[j] ⟵ A[q + j + 1]
end
i ⟵ 0
j ⟵ 0
k ⟵ p
while i ≠ n1 and j ≠ n2 do
    if L[i] ≤ R[j] then
        A[k] ⟵ L[i]
        i ⟵ i + 1
    end
    else
        inv ⟵ inv + j      /* This keeps track of the number of
         inversions between the left and right arrays */

        j ⟵ j + 1
    end
    k ⟵ k + 1;
end
if i = n1 then
    for m ⟵ j to n2 − 1 do
        A[k] ⟵ R[m]
        k ⟵ k + 1
    end
end
if j = n2 then
    for m ⟵ i to n1 − 1 do
        A[k] ⟵ L[m]
        inv ⟵ inv + n2
         /* Tracks inversions once we have exhausted the right
         array.  At this point, every element of the right
         array contributes an inversion */
        k ⟵ k + 1
    end
end
return inv
```

Chapter 3

Growth of Functions