Introduction to Algorithms - Solutions
3rd Edition

Purbayan Chowdhury

October 12, 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.**

Algorithm 1: Non-increasingInsertionSort

Input : A ⟵ Unsorted Array
Output: A ⟵ Array Sorted in Non-increasing Order

```
1 for j ⟵ 1 to A.length − 1 do
2 |   key ⟵ A[j]
  |   /* Insert A[j] into the sorted sequence A[1..j-1]      */
3 |   i ⟵ j − 1
4 |   while i ≥ 0 and A[i] > key do
5 |   |   A[i + 1] ⟵ A[i]
6 |   |   i ⟵ i − 1
7 |   end
8 |   A[i + 1] ⟵ key
9 end
```

A.

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.**

Algorithm 2: Linear-Search

Input : A ⟵ Array
v ⟵ value to be searched
Output: i ⟵ index of the value if found, else NIL

```
1 i ⟵ NIL
2 for j ⟵ 0 to A.length − 1 do
3 |   if A[j] = v then
4 |   |   i ⟵ j
5 |   |   break
6 |   end
7 end
8 return i
```

A.

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.**

Algorithm 3: n-bitBinaryAddition

Input : A ⟵ First Array
B ⟵ Second Array
Output: C ⟵ Binary Addition Result

```
1  carry ⟵ 0
2  for i ⟵ n − 1 downto 0 do
3  |   C[i + 1] ⟵ (A[i] + B[i] + carry)(mod2)
4  |   if A[i] + B[i] + carry ≥ 2 then
5  |   |   carry ⟵ 1
6  |   end
7  |   else
8  |   |   carry ⟵ 0
9  |   end
10 end
11 C[0] ⟵ carry
```

A.

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.

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
A. |  |  /* 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 $A[0 \dots i-1]$ 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.

Algorithm 5: MergeSort

Input : A ⟵ Unsorted Array
p ⟵ start index
q ⟵ middle index
r ⟵ end index

Output: A ⟵ Array Sorted in Increasing Order

```
1  n1 ⟵ q − p + 1
2  n2 ⟵ r − q
3  let L[1, . . . , n1] and R[1, . . . , n2] be new arrays
4  for i ⟵ 0 to n1 − 1 do
5  |  L[i] ⟵ A[p + i]
6  end
7  for j ⟵ 0 to n2 − 1 do
8  |  R[j] ⟵ A[q + j + i]
9  end
10 i ⟵ 0
11 j ⟵ 0
12 k ⟵ p
13 while i ≠ n1 and j ≠ n2 do
14 |  if L[i] ≤ R[j] then
15 |  |  A[k] ⟵ L[i]
16 |  |  i ⟵ i + 1
17 |  end
18 |  else
19 |  |  A[k] ⟵ R[j]
20 |  |  j ⟵ j + 1
21 |  end
22 |  k ⟵ k + 1
23 end
24 if i = n1 then
25 |  for m ⟵ j to n2 − 1 do
26 |  |  A[k] ⟵ R[m]
27 |  |  k ⟵ k + 1
28 |  end
29 end
30 if j = j1 then
31 |  for m ⟵ j to n1 − 1 do
32 |  |  A[k] ⟵ L[m]
33 |  |  k ⟵ k + 1
34 |  end
35 end
```

A.

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 $\Theta(\lg n)$.

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

```
1  if a > b then
2  |  return NIL
A. 3  end
4  m ⟵ ⌊(a+b)/2⌋
5  if A[m] = v then
6  |  return m
7  end
8  if A[m] < v then
9  |  return RecursiveBinarySearch(a, m, v)
10 end
11 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 $\longleftarrow$ Array of n integers
x $\longleftarrow$ sum to be found
Output: If found return *true*, else *false*

```
1  i ⟵ 0
2  j ⟵ n
3  while i < j do
4  |  if S[i] + S[j] = x then
5  |  |  return true
6  |  end
7  |  if S[i] + S[j] < x then
8  |  |  i ⟵ i + 1
9  |  end
10 |  if S[i] + S[j] > x then
11 |  |  j ⟵ j − 1
12 |  end
13 end
14 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

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

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?**

```
1 y ⟵ 0
2 for i ⟵ 0 to n do
3     y_i ⟵ 1
4     for j ⟵ 1 to i do
5         y_i ⟵ y_i * x
6     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 $\Theta(n \lg n)$ worst-case time. (Hint: Modify merge sort)**

Algorithm 8: Inversions

Input : A ⟵ Unsorted array
p ⟵ start index
r ⟵ end index

Output: inv ⟵ No of inversions in the array

```
1 if p < r then
2 |   q ⟵ ⌊(p + r)/2⌋
3 |   left ⟵ Inversions(A, p, q)
4 |   right ⟵ Inversions(A, q + 1, r)
5 |   inv ⟵ CountInversions(A, p, q, r)+left + right
6 |   return inv
7 end
8 return 0
```

A.

Algorithm 9: CountInversions

Input : A ⟵ Unsorted array
p ⟵ start index
q ⟵ middle index
r ⟵ end index

Output: inv ⟵ No of inversions in the array

```
1  inv ⟵ 0
2  n1 ⟵ q − p + 1
3  n2 ⟵ r − q
4  let L[1, ..., n1] and R[1, ..., n2] be new arrays
5  for i ⟵ 0 to n1 − 1 do
6  |  L[i] ⟵ A[p + i]
7  end
8  for j ⟵ 0 to n2 − 1 do
9  |  R[j] ⟵ A[q + j + 1]
10 end
11 i ⟵ 0
12 j ⟵ 0
13 k ⟵ p
14 while i ≠ n1 and j ≠ n2 do
15 |  if L[i] ≤ R[j] then
16 |  |  A[k] ⟵ L[i]
17 |  |  i ⟵ i + 1
18 |  end
19 |  else
20 |  |  inv ⟵ inv + j      /* This keeps track of the number of
   |  |   inversions between the left and right arrays */
21 |  |
22 |  |  j ⟵ j + 1
23 |  end
24 |  k ⟵ k + 1;
25 end
26 if i = n1 then
27 |  for m ⟵ j to n2 − 1 do
28 |  |  A[k] ⟵ R[m]
29 |  |  k ⟵ k + 1
30 |  end
31 end
32 if j = n2 then
33 |  for m ⟵ i to n1 − 1 do
34 |  |  A[k] ⟵ L[m]
35 |  |  inv ⟵ inv + n2
   |  |   /* Tracks inversions once we have exhausted the right
   |  |   array.  At this point, every element of the right
   |  |   array contributes an inversion */
36 |  |  k ⟵ k + 1
37 |  end
38 end
39 return inv
```

Chapter 3

Growth of Functions

3.1 Asymptotic notation

Ex 3.1-1 Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that max($f(n).g(n)$) = $\Theta(f(n) + g(n))$.

A.

Ex 3.1-2 Show that for any real constants a and b, where $b > 0$, $(n + a)^b = \Theta(n^b)$

A. Let $c = 2^b$ and $n_0 \geq 2a$, then for all $n \geq n_0$, we have $(n + a)^b \leq (2n)^b = cn^b$, so $(n + a)^b = O(n^b)$. Let $n_0 \geq \frac{-a}{1-1/2^{1/b}}$ and $c = 1/2$, then $n \geq n_0 \geq \frac{-a}{1-1/2^{1/b}}$ if and only if $n - 2^{1/b} \geq -a$, also $n + a \geq (1/2)^{a/b}n$, also $(n + a)^b \geq cn^b$. Therefore $(n + a)^b = \Omega(n^b)$. By Theorem 3.1, $(n + a)^b = \Theta(n^b)$.

Ex 3.1-3 Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

A.

Ex 3.1-4 Is $2^{n+1} = O(n^2)$? Is $2^{2n} = O(n^2)$?

A. $2^{n+1} \geq 2.2^n$ for all $n \geq 0$, so $2^{n+1} = O(2^n)$, but, 2^{2n} is not $O(2^n)$. If there would exist n_0 and c such that $n \geq n_0$ implies $2^n.2^n = 2^{2n} \leq c2^n$, so $2^n \leq c$ for $n \geq n_0$, which is clearly impossible since c is a constant.

Ex 3.1-5 Prove Theorem 3.1

A. Suppose $f(n) \in \Theta(g(n))$, then $\exists c_1, c_2, n_0, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. We have $c_1 g(n) \leq f(n) (f(n) \in \Omega(g(n)))$ and $f(n) \leq c_2 g(n) (f(n) \in O(g(n)))$. Suppose that we had $\exists n_1, c_1, \forall n \geq n_1, c_1 g(n) \leq f(n)$ and

$\exists n_2, c_2, \forall n \geq n_2, f(n) \leq c_2 g(n)$. Putting these together, letting $n_0 = max(n_1, n_2)$, we have $\forall n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Ex 3.1-6 **Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.**

Ex 3.1-7 **Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.**

Ex 3.1-8 **We can extend our notation to the case of two parameters n and m that can go to infinity independently at different rates. For a given function $g(n, m)$, we denote by $O(g(n, m))$ the set of functions**

$$O(g(n,m)) = \{f(n,m) : \text{there exist positive constants such that } 0 \leq f(n,m) \leq cg(n,m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}.$$

Give corresponding definitions for $\Omega(g(n, m))$ and $\Theta(g(n, m))$.

3.2 Standard notations and common functions

Ex 3.2-1 **Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n) + g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n).g(n)$ is monotonically increasing.**

Ex 3.2-2 **Prove the equation (3.16).**

Ex 3.2-3 **Prove the equation (3.19). Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.**

Ex 3.2-4 **Is the function $\lceil \lg n \rceil !$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil !$ polynomially bounded?**

Ex 3.2-5 **Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?**

Ex 3.2-6 **Show that the golden ratio ϕ and its conjugate $\hat{\phi}$ both satisfy the equation $x^2 = x + 1$.**

Ex 3.2-7 **Prove by induction that the ith Fibonacci number satisfies the equality**
$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$
where ϕ is the golden ratio and $\hat{\phi}$ is its conjugate.

Ex 3.2-8 **Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n / \ln n)$.**

Problems

3-1 ***Asymptotic behavior of polynomials***

Let
$p(n) = \sum_{i=0}^{d} a_i n^i$,
where $a_d = 0$, **be a degree-**d **polynomial in** n, **and let** k **be a constant. Use the definitions of the asymptotic notations to prove the following properties.**

(a) If $k \geq d$, **then** $p(n) = O(n^k)$.

(b) If $k \leq d$, **then** $p(n) = \Omega(n^k)$.

(c) If $k = d$, **then** $p(n) = \Theta(n^k)$.

(d) If $k > d$, **then** $p(n) = o(n^k)$.

(e) If $k < d$, **then** $p(n) = \omega(n^k)$.

3-2 ***Relative asymptotic growths***

Indicate, for each pair of expressions (A, B) **in the table below, whether** A **is** O, o, Ω, ω, **or** Θ **of** B. **Assume that** $k \geq 1$, $\epsilon\ 0$, **and** $c > 1$ **are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.**

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ					
$n^k n$	c^n					
$\sqrt{n}$	$n^{\sin n}$					
2^n	$2^{n/2}$					
$n^{\lg c}$	$c^{\lg n}$					
$\lg(n!) n$	$\lg(n^n)$					

3-3 ***Ordering by asymptotic growth rates***

(a) Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_3 0$ **of the functions satisfying** $g_1 = \Omega(g_2), g_2 D \Omega(g_3), \ldots, g_2 9 = \Omega(g_3 0)$**s. Partition your list into equivalence classes such that functions** $f(n)$ **and** $g(n)$ **are in the same class if and only if** $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n.2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$(\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

(b) Give an example of a single nonnegative function $f(n)$ **such that for all functions** $g_i(n)$ **in part (a),** $f(n)$ **is neither** $O(g_i(n))$ **nor** $\Theta(g_i(n))$.

3-4 ***Asymptotic notation properties***

Let $f(n)$ **and** $g(n)$ **be asymptotically positive functions. Prove or disprove each of the following conjectures.**

(a) $f(n) = O(g(n))$ **implies** $g(n) = O(f(n))$

(b) $f(n) + g(n) = \Theta(min(f(n), g(n)))$

(c) $f(n) = O(g(n))$ **implies** $(\lg(f(n)))$, **where** $\lg(g(n)) \geq 1$ **and** $f(n) \geq 1$ **for all sufficiently large** n.

(d) $f(n) = O(g(n))$ **implies** $2^{f(n)} = O(2^{g(n)})$.

(e) $f(n) = O((f(n))^2)$.

(f) $f(n) = O(g(n))$ **implies** $g(n) = \Omega(f(n))$.

(g) $f(n) = \Theta((f(n/2))$

(h) $f(n) + o(f(n)) = \Theta((f(n))$

3-5 ***Asymptotic notation properties***

Some authors define Ω in a slightly different way than we do; let's use Ω^∞ (read "omega infinity") for this alternative definition. We say that $f(n) = \Omega^\infty(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers n.

(a) **Show that for any two functions $f(n)$ and $g(n)$ that are asymptotically nonnegative, either $f(n) = O(g(n))$ or $f(n) = \Omega^\infty(g(n))$ or both, whereas this is not true if we use Ω in place of Ω^∞.**

(b) **Describe the potential advantages and disadvantages of using Ω^∞ instead of Ω to characterize the running times of programs.**
Some authors also define O in a slightly different manner; let's use O' for the alternative definition. We say that $f(n) = O'(g(n))$ if and only if $|f(n)| = O(g(n))$.

(c) **What happens to each direction of the "if and only if" in Theorem 3.1 if we substitute O' for O but still use Ω?**
Some authors define $\tilde{O}$ (read "soft-oh") to mean O with logarithmic factors ignored:

$$\tilde{O}g(n) = \{f(n) : \text{there exist positive constants } c, k \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n)\lg^k(n) \text{ for all } n \geq n_0\}$$

(d) **Define $\tilde{\Omega}$ and $\tilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.**

3-6 ***Iterated functions***

We can apply the iteration operator* used in the lg* function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function f_c^* by

$$f_c^*(n) = \min\{i \geq 0 \ : \ f^{(i)} \leq c\},$$

which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the number of iterated applications of the function f required to reduce its argument down to c or less.
For each of the following functions $f(n)$ and constants c, give as tight a bound as possible on $f_c^*(n)$.

$f(n)$	c	$f_c^*(n)$
$n-1$	0	$\lceil n \rceil$
$\lg n$	1	$\lg^* n$
$n/2$	1	$\lceil \lg(n) \rceil$
$n/2$	2	$\lceil \lg(n) \rceil - 1$
$\sqrt{n}$	2	$\lg \lg n$
$\sqrt{n}$	1	$undefined$
$n^{1/3}$	2	$\lg_3 \lg_2(n)$
$n/\lg n$	2	$\Omega(\frac{\lg n}{\lg \lg n})$

Chapter 4

Divide-and-Conquer

4.1 The maximum-subarray problem

Ex 4.1-1 What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?

A. It will return the least negative one. As each of the cross sums are computed, the most positive one must have the shortest possible length. The algorithm doesn't consider zero sub-arrays, so it must have length 1.

Ex 4.1-2 Write pseudocode for the brute-force method of solving the maximum subarray problem. Your procedure should run in $\Theta(n^2)$ time.

Algorithm 10: BruteForceMaxSubarray

```
   Input  : A ⟵ Array
   Output: left, right ⟵ starting & ending indexes of subarray in the
           array
 1 left ⟵ 0
 2 right ⟵ 0
 3 max ⟵ A[0]
   /* Increment left end of subarray                              */
 4 for i ⟵ 0 to n − 1 do
 5 |  curSum ⟵ 0
A. |  /* Increment right end of subarray                          */
 6 |  for j ⟵ i to n − 1 do
 7 |  |  curSum ⟵ curSum + A[j]
 8 |  |  if curSum > max then
 9 |  |  |  max = curSum
10 |  |  |  left = i
11 |  |  |  right = j
12 |  |  end
13 |  end
14 end
15 return i
```

Ex 4.1-3 **Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0 . Does that change the crossover point?**

Ex 4.1-4 **Suppose we change the definition of the maximum subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is 0. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?**

A. Do a linear scan of the input array to see if it contains any positive entries. If it does, run the algorithm as usual otherwise return the empty subarray with sum 0 and terminate the algorithm.

Ex 4.1-5 **Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1 \ldots j]$, extend the answer to find a maximum subarray**

ending at index $j+1$ by using the following observation: a maximum subarray of $A[1 \ldots j+1]$ is either a maximum subarray of $A[1 \ldots j]$ or a subarray $A[i \ldots j+1]$, for some $1 \leq i \leq j+1$. Determine a maximum subarray of the form $A[i \ldots j+1]$ in constant time based on knowing a maximum subarray ending at index j.

4.2 Strassen's algorithm for matrix multiplication

Ex 4.2-1 Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}$$

Show your work.

A. $S_1 = 8 - 2 = 6$
$S_2 = 1 + 3 = 4$
$S_3 = 7 + 5 = 12$
$S_4 = 4 - 6 = -2$
$S_5 = 1 + 5 = 6$
$S_6 = 6 + 2 = 8$
$S_7 = 3 - 5 = -2$
$S_8 = 4 + 2 = 6$
$S_9 = 1 - 7 = -6$
$S_{10} = 6 + 8 = 14$

$P_1 = 6$, $P_2 = 8$, $P_3 = 72$, $P_4 = -10$, $P_5 = 48$, $P_6 = -12$, $P_7 = -84$

$C_{11} = 48 - 10 - 8 - 12 = 18$
$C_{12} = 6 + 8 = 14$
$C_{21} = 72 - 10 = 62$
$C_{22} = 48 + 6 - 72 + 84 = 96$
So, we get the final result:

$$\begin{pmatrix} 18 & 14 \\ 62 & 96 \end{pmatrix}$$

Ex 4.2-2 Write pseudocode for Strassen's algorithm.

Algorithm 11: Strassen

```
   /* Let A[i...j][k...m] denote the submatrix of A consisting
      of rows i through j and columns k through m.          */
 1 if A.length = 1 then
 2 |  return A[0].B[0]
 3 end
   /* Let C be a new n × n matrix                           */
 4 A11 ⟵ A[0...n/2 − 1][0...n/2 − 1]
 5 A12 ⟵ A[0...n/2 − 1][n/2...n − 1]
 6 A21 ⟵ A[n/2...n − 1][0...n/2 − 1]
 7 A22 ⟵ A[n/2...n − 1][n/2...n − 1]
 8 S1 = B12 − B22
 9 S2 = A11 + A12
10 S3 = A21 + A22
11 S4 = B21 − B11
12 S5 = A11 + A22
13 S6 = B11 + B22
14 S7 = A12 − A22
15 S8 = B21 + B22
16 S9 = A11 − A21
17 S10 = B11 + B12
18 P1 = Strassen(A11, S1)
19 P2 = Strassen(S2, B22)
20 P3 = Strassen(S3, B11)
21 P4 = Strassen(A22, S4)
22 P5 = Strassen(S5, S6)
23 P6 = Strassen(S7, S8)
24 P7 = Strassen(S9, S10)
25 C[0...n/2 − 1][0...n/2 − 1] = P5 + P4 − P2 + P6
26 C[0...n/2 − 1][n/2...n − 1] = P1 + P2
27 C[n/2...n − 1][0...n/2 − 1] = P3 + P4
28 C[n/2...n − 1][n/2...n − 1] = P5 + P1 − P3 − P7; return C
```

Ex 4.2-3 How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

A. We can pad out the input matrices to the next largest powers of 2 and run the given algorithm. This will at most double the value of n because each power of 2 is off from each other by a factor of 2. So, this will have runtime
$m^{\lg 7} \leq (2n)^{\lg 7} = 7n^{\lg 7} \in O(n^{\lg 7})$ and $m^{\lg 7} \geq n^{\lg 7} \in \Omega(n^{\lg 7})$
Putting these together, we get the runtime is $\Theta(n^{\lg 7})$.

Ex 4.2-4 What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multi-

plication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

A. Assume that $n = 3^m$ for some m, then using block matrix multiplication, we obtain the recursive running time $T(n) = kT(n/3) + O(1)$. Using the Master theorem, we need the largest integer k such that $\log_3 k < \lg 7$. This is given by $k = 21$.

Ex 4.2-5 V. Pan has discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare to Strassen's algorithm?

Ex 4.2-6 How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

Ex 4.2-7 Show how to multiply the complex numbers $a+bi$ and $c+di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac-bd$ and the imaginary component $ad+bc$ separately

A. We can see that the final result should be

$$(a + bi)(c + di) = ac - bd + (cb + ad)i$$

We will be multiplying

$$P1 = (a + b)c = ac + bc,\ P2 = b(c + d) = bc + bd,\ P3 = (a - b)d$$

We can get real part by taking $P1 - P2$ and imaginary part by taking $P2 + P3$.

4.3 The substitution method for solving recurrences

Ex 4.3-1 Show that the solution of $T(n) = T(n-1) + n$ is $O(n^2)$.

A. Assume by induction, $T(n) \leq cn^2$, where c is taken to be $\max(1, T(1))$, then

$$T(n) = T(n-1)+n \leq c(n-1)^2+n = cn^2+(1-2c)n+1 \leq cn^2+2-2c \leq cn^2$$

By inductive hypothesis from the first inequality, the second from the fact that $n \geq 1$ and $1 - 2c < 0$ and from the fact that $c \geq 1$.

Ex 4.3-2 Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$.

A. Assume $T(n) \leq 3 \log n - 1$, which implies $T(n) = O(\lg n)$.

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + 1 \\ &\leq 3 \lg(\lceil n/2 \rceil) - 1 + 1 \\ &\leq 3 \lg(3n/4) \\ &= 3 \lg n + 3 \lg(3/4) \\ &\leq 3 \lg n + \lg(1/2) \\ &= 3 \lg n - 1 \end{aligned}$$

Ex 4.3-3 We saw that the solution of $T(n) = 2T(\lfloor n/2 \rfloor) + n$ is $O(n \lg n)$. Show that the solution of this recurrence is also $\Omega(n \lg n)$. Conclude that the solution is $\Theta(n \lg n)$.

Ex 4.3-4 Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition $T(1) = 1$ for recurrence (4.19) without adjusting the boundary conditions for the inductive proof.

Ex 4.3-5 Show that $\Theta(n \lg n)$ is the solution to the "exact" recurrence (4.3) for merge sort.

Ex 4.3-6 Show that the solution to $T(n) = 2T(\lceil n/2 \rceil + 17) + n$ is $O(n \lg n)$

Ex 4.3-7 Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/3) + n$ is $T(n) = \Theta(n^{\log_3 4})$. Show that a substitution proof with the assumption $T(n) \leq cn^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

Ex 4.3-8 Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/3) + n^2$ is $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \leq cn^2$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

A.

Ex 4.3-9 Solve the recurrence $T(n) = 3T(n) + \log n$ by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

A. Consider n of the form $2k$ and then the recurrence becomes

$$T(2^k) = 3T(2^{k/2}) + k$$

We define $S(k) = T(2^k)$, so $S(k) = 3S(k/2) + k$
We use the inductive hypothesis $S(k) \leq (S(1) + 2)k^{\lg 3} - 2k$

$S(k) = 3S(k/2) + k \leq 3(S(1) + 2)(k/2)^{\lg 3} - 3k + k = (S(1) + 2)k^{\lg 3} - 2k$

as desired. Similarly, we also show that $S(k) \geq (S(1) + 2)k^{\lg 3} - 2k$

$$S(k) = 3S(k/2) + k \geq (S(1) + 2)k^{\lg 3} - 2k$$

So, we get $S(k) = (S(1) + 2)k^{\lg 3} - 2k$ and putting this into T, $T(2^k) = (T(2) + 2)k^{\lg 3} - 2k$. So, $T(n) = (T(2) + 2)(\lg n)^{\lg 3} - 2\lg(n)$.

4.4 The recursion-tree method for solving recurrences

Ex 4.4-1

4.5 The master method for solving recurrences

Ex 4.5-1 Use the master method to give tight asymptotic bounds for the following recurrences.

(a) $\mathbf{T(n) = 2T(n/4) + 1}$

A. $a = 2$, $b = 4$, $f(n) = 1 = O(n^0)$
$T(n) = \Theta(\sqrt{n})$

(b) $\mathbf{T(n) = 2T(n/4) + \sqrt{n}}$

A. $a = 2$, $b = 4$, $f(n) = n^{1/2} = O(n^{1/2})$

4.6 Proof of the master theorem

Problems

Chapter 5

Probabilistic Analysis and Randomized Algorithms

Chapter 6

Heapsort

Chapter 7

Quicksort

Chapter 8

Sorting in Linear Time

Chapter 9

Medians and Order Statistics

Chapter 10

Elementary Data Structures