

Introduction to Algorithms - Solutions
3rd Edition

Purbayan Chowdhury

October 10, 2020

Contents

1	The Role of Algorithms in Computing	5
2	Getting Started	7
2.1	Insertion Sort	7
2.2	Analyzing algorithms	8
2.3	Designing algorithms	9
3	Growth of Functions	19
3.1	Asymptotic notation	19
3.2	Standard notations and common functions	20
4	Divide-and-Conquer	23
4.1	The maximum-subarray problem	23
4.2	Strassen's algorithm for matrix multiplication	23
4.3	The substitution method for solving recurrences	23
4.4	The recursion-tree method for solving recurrences	23
4.5	The master method for solving recurrences	23
4.6	Proof of the master theorem	23
5	Probabilistic Analysis and Randomized Algorithms	25
5.1	The hiring problem	25
5.2	Indicator random variables	25
5.3	Randomized algorithms	25
5.4	Probabilistic analysis and further uses of indicator random variables	25
6	Heapsort	27
6.1	Heaps	27
6.2	Maintaining the heap property	27
6.3	Building a heap	27
6.4	The heapsort algorithm	27
6.5	Priority queues	27

7	Quicksort	29
7.1	Description of quicksort	29
7.2	Performance of quicksort	29
7.3	A randomized version of quicksort	29
7.4	Analysis of quicksort	29
8	Sorting in Linear Time	31
8.1	Lower bounds for sorting	31
8.2	Counting sort	31
8.3	Radix sort	31
8.4	Bucket sort	31
9	Medians and Order Statistics	33
9.1	Minimum and maximum	33
9.2	Selection in expected linear time	33
9.3	Selection in worst-case linear time	33
10	Elementary Data Structures	35
10.1	Stacks and queues	35
10.2	Linked lists	35
10.3	Implementing pointers and objects	35
10.4	Representing rooted trees	35

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 : $A \leftarrow$ Unsorted Array
Output: $A \leftarrow$ Array Sorted in Non-increasing Order

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

Ex 2.1-3

Ex 2.1-4 Consider the searching problem: **Input:** A sequence of n numbers $A = (a_1, a_2, \dots, a_n)$ **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 : $A \leftarrow$ Array
 $v \leftarrow$ value to be searched
Output: $i \leftarrow$ index of the value if found, else NIL

```

1  $i \leftarrow NIL$ 
2 for  $j \leftarrow 0$  to  $A.length - 1$  do
A. 3   if  $A[j] = v$  then
4        $i \leftarrow j$ 
5       break
6   end
7 end
8 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.

Algorithm 3: n -bitBinaryAddition

Input : $A \leftarrow$ First Array
 $B \leftarrow$ Second Array
Output: $C \leftarrow$ Binary Addition Result

```

1  $carry \leftarrow 0$ 
2 for  $i \leftarrow n - 1$  downto  $0$  do
A. 3    $C[i + 1] \leftarrow (A[i] + B[i] + carry)(\text{mod}2)$ 
4   if  $A[i] + B[i] + carry \geq 2$  then
5        $carry \leftarrow 1$ 
6   end
7   else
8        $carry \leftarrow 0$ 
9   end
10 end
11  $C[0] \leftarrow carry$ 
```

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.

Algorithm 4: SelectionSort

Input : $A \leftarrow$ Unsorted Array
Output: $A \leftarrow$ Array Sorted in Increasing Order

```

1  for  $i \leftarrow 0$  to  $n - 1$  do
2       $min \leftarrow i$ 
3      for  $j \leftarrow i + 1$  to  $n$  do
4          /* Find the index of the  $i$ th smallest element */
5          if  $A[j] < A[min]$  then
6               $min \leftarrow j$ 
7          end
8      end
9      Swap  $A[min]$  and  $A[i]$ 
10 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 n th 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 : $A \leftarrow$ Unsorted Array
 $p \leftarrow$ start index
 $q \leftarrow$ middle index
 $r \leftarrow$ end index

Output: $A \leftarrow$ Array Sorted in Increasing Order

```

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

```

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

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

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

Ex 2.3-4 We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \dots n]$, we recursively sort $A[1 \dots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \dots n-1]$. 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 $A[1 \dots n-1]$. 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 ← ⌊ $\frac{a+b}{2}$ ⌋
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 $A[1 \dots j-1]$. 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 : $S \leftarrow$ Array of n integers

$x \leftarrow$ sum to be found

Output: If found return *true*, else *false*

```

1  $i \leftarrow 0$ 
2  $j \leftarrow n$ 
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      $i \leftarrow i + 1$ 
9   end
10  if  $S[i] + S[j] > x$  then
11     $j \leftarrow j - 1$ 
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 $c_1n - n(c_2) = 0$ 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

$$A'[1] \leq A'[2] \leq \dots \leq A'[n], \quad (2.1)$$

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 $A[i \dots n]$ 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 $A[i \dots n]$ 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 $A[i \dots n]$, so when we swap $A[k]$ and $A[k-1]$ we know that the smallest element of $A[i \dots n]$ 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 $A[i \dots n]$ 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 $A[1 \dots i-1]$ 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 $A[i \dots n]$ is in position i . Since the $i-1$ smallest elements of A are already in $A[1 \dots i-1]$, $A[i]$ must be the i th smallest element of A . Therefore $A[1 \dots i]$ contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, $A[1 \dots n]$ 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 i th 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

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

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

```

1  $y \leftarrow 0$ 
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  $y \leftarrow 0$ 
2 for  $i \leftarrow 0$  to  $n$  do
3    $y_i \leftarrow 1$ 
4   for  $j \leftarrow 1$  to  $i$  do
5      $y_i \leftarrow y_i * x$ 
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, \dots, 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 $A[1 \dots n]$ 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 $1, 2, \dots, n$ has the most inversions? How many does it have?**
- A. The n -element array with the most inversions is $(n, n-1, \dots, 2, 1)$. It has $n-1 + n-2 + \dots + 2 + 1 = n(n-1)/2$ 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 $\sum_{i=1}^n I(i)$ 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 $\sum_{i=1}^n I(i)$ 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 : $A \leftarrow$ Unsorted array $p \leftarrow$ start index $r \leftarrow$ end index**Output:** $inv \leftarrow$ No of inversions in the array

A. **1 if** $p < r$ **then**

2 $q \leftarrow \lfloor (p + r) / 2 \rfloor$

3 $left \leftarrow \text{Inversions}(A, p, q)$

4 $right \leftarrow \text{Inversions}(A, q + 1, r)$

5 $inv \leftarrow \text{CountInversions}(A, p, q, r) + left + right$

6 **return** inv

7 **end**

8 **return** 0

Algorithm 9: CountInversions

Input : $A \leftarrow$ Unsorted array
 $p \leftarrow$ start index
 $q \leftarrow$ middle index
 $r \leftarrow$ end index

Output: $inv \leftarrow$ No of inversions in the array

```

1  $inv \leftarrow 0$ 
2  $n1 \leftarrow q - p + 1$ 
3  $n2 \leftarrow r - q$ 
4 let  $L[1, \dots, n1]$  and  $R[1, \dots, n2]$  be new arrays
5 for  $i \leftarrow 0$  to  $n1 - 1$  do
6    $L[i] \leftarrow A[p + i]$ 
7 end
8 for  $j \leftarrow 0$  to  $n2 - 1$  do
9    $R[j] \leftarrow A[q + j + 1]$ 
10 end
11  $i \leftarrow 0$ 
12  $j \leftarrow 0$ 
13  $k \leftarrow p$ 
14 while  $i \neq n1$  and  $j \neq n2$  do
15   if  $L[i] \leq R[j]$  then
16      $A[k] \leftarrow L[i]$ 
17      $i \leftarrow i + 1$ 
18   end
19   else
20      $inv \leftarrow inv + j$  /* This keeps track of the number of
21                          inversions between the left and right arrays */
22      $j \leftarrow j + 1$ 
23   end
24    $k \leftarrow k + 1$ 
25 end
26 if  $i = n1$  then
27   for  $m \leftarrow j$  to  $n2 - 1$  do
28      $A[k] \leftarrow R[m]$ 
29      $k \leftarrow k + 1$ 
30   end
31 end
32 if  $j = n2$  then
33   for  $m \leftarrow i$  to  $n1 - 1$  do
34      $A[k] \leftarrow L[m]$ 
35      $inv \leftarrow inv + n2$ 
36     /* Tracks inversions once we have exhausted the right
37        array. At this point, every element of the right
38        array contributes an inversion */
39      $k \leftarrow k + 1$ 
40   end
41 end
42 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. $2^{n+1} \geq 2 \cdot 2^n$ 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 $2^n \cdot 2^n = 2^{2n} \leq c2^n$, 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

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

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 $f(n) \cdot g(n)$ 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 i th 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 $a_d \neq 0$, 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	$2^{n/2}$					
$n^{\lg c}$	$c^{\lg n}$					
$\lg(n!)n$	$\lg(n^n)$					

3-2 Ordering by asymptotic growth rates

- (a) Rank the following functions by order of growth; that is, find an arrangement g_1, g_2, \dots, g_9 of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \dots, g_8 = \Omega(g_9)$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

Chapter 4

Divide-and-Conquer

- 4.1 The maximum-subarray problem
- 4.2 Strassen's algorithm for matrix multiplication
- 4.3 The substitution method for solving recurrences
- 4.4 The recursion-tree method for solving recurrences
- 4.5 The master method for solving recurrences
- 4.6 Proof of the master theorem

Problems

Chapter 5

Probabilistic Analysis and Randomized Algorithms

5.1 The hiring problem

5.2 Indicator random variables

5.3 Randomized algorithms

5.4 Probabilistic analysis and further uses of indicator random variables

Problems

Chapter 6

Heapsort

6.1 Heaps

6.2 Maintaining the heap property

6.3 Building a heap

6.4 The heapsort algorithm

6.5 Priority queues

Problems

Chapter 7

Quicksort

7.1 Description of quicksort

7.2 Performance of quicksort

7.3 A randomized version of quicksort

7.4 Analysis of quicksort

Problems

Chapter 8

Sorting in Linear Time

8.1 Lower bounds for sorting

8.2 Counting sort

8.3 Radix sort

8.4 Bucket sort

Problems

Chapter 9

Medians and Order Statistics

9.1 Minimum and maximum

9.2 Selection in expected linear time

9.3 Selection in worst-case linear time

Problems

Chapter 10

Elementary Data Structures

10.1 Stacks and queues

10.2 Linked lists

10.3 Implementing pointers and objects

10.4 Representing rooted trees

Problems