讨论与反馈

2014-3-20

6.3. Analyze the worst-case time complexity of each of the four algorithms given in Exercise 5.15 for computing m^n. Count multiplications only.

5.15. Each of the following algorithms, *Pwr1*, *Pwr2*, *Pwr3*, and *Pwr4*, computes the value of m^n and produces the result in the variable PW. It is assumed that m is a positive integer and that n is a natural number (i.e., either 0 or a positive integer). Prove the total correctness of all four algorithms.

i. Algorithm *Pwr1*:

$PW \leftarrow 1$;
do the following n times:
  $PW \leftarrow PW \times m$.

ii. Algorithm *Pwr2*:

call **compute-power-of** m **and** n.

The subroutine **compute-power** is defined as:

subroutine **compute-power-of** B **and** E:
  if $E = 0$ then $PW \leftarrow 1$;
  otherwise (i.e., if E is positive) do the following:
    call **compute-power-of** B **and** $E - 1$;
    $PW \leftarrow B \times PW$;
  return.

iii. Algorithm *Pwr3*:

$PW \leftarrow 1$;
$B \leftarrow m$;
$E \leftarrow n$;
while $E \neq 0$ do the following:
  if E is an even number then do the following:
    $B \leftarrow B \times B$;
    $E \leftarrow E/2$;
  otherwise (i.e., if E is an odd number) do the following:
    $PW \leftarrow PW \times B$;
    $E \leftarrow E - 1$.

iv. Algorithm *Pwr4*:

call **times-power-of** 1, m, **and** n.

The subroutine **times-power** is defined by:

subroutine **times-power-of** Q, B, **and** E:
  if $E = 0$ then $PW \leftarrow Q$;
  otherwise, if E is an even number then
    call **times-power-of** Q, $B \times B$, **and** $E/2$;
  otherwise (i.e., if E is an odd number)
    call **times-power-of** $Q \times B$, B, **and** $E - 1$;
  return.

From O(n) to O(logn).
在数论算法中我们还会碰到到。

6.15. Recall the problem of detecting palindromes, described in Exercise 5.10. In the following, consider the two correct solutions to this problem: algorithm *Pal1*, presented in Exercise 5.10, and algorithm *Pal4*, which you were asked to construct in Exercise 5.14. Assume that strings are composed of the two symbols "a" and "b" only, and that the only operations we count are comparisons (that is, applications of the **eq** predicate).

(a) Analyze the worst-case time complexity of algorithms *Pal1* and *Pal4*, providing upper bounds for both.

(b) Suggest a good lower bound for the problem of detecting palindromes.

(c) Assume a uniform distribution of the strings input by the algorithms. In other words, for each N, all strings of length N over the alphabet {a, b} can occur as inputs with equal probability. Perform an average-case time analysis of algorithms *Pal1* and *Pal4*.

6.16. A correct solution to Exercise 6.15 shows that the average-case complexity of *Pal4* is *better* than the lower bound on the palindrome detection problem.

(a) Explain why this fact is not self-contradictory.

(b) What does this fact mean with regards to the performance of algorithm *Pal4* on a large set of strings ("most" strings)? What does it mean with regards to the performance on a small set of strings?

(c) How would your answer to (b) change if the worst-case complexity of *Pal4* was significantly *larger* than the lower bound on the problem?

(b) Palindromes of length n cannot be detected with fewer than $n/2$ symbol comparisons in the worst case, otherwise we would be able to exhibit nonpalindrome strings that are tagged as palindromes by the algorithm in question. Hence, we have a lower bound of $n/2$ on the problem.

(c) Algorithm *Pal1* requires $O(n)$ comparisons on *every* string, hence this is also its average-case complexity.

The average-case analysis of *Pal4* is harder. Fix an integer n, and for simplicity assume that n is even, i.e., $n = 2k$ for some integer $k > 0$. Let $s = a_1 a_2 \ldots a_k b_k \ldots b_2 b_1$ be a typical string of length n. Call s an *i-nonpalindrome*, for some $i \in \{1, 2, \ldots, k\}$, if $a_j = b_j$ for all $j < i$, but $a_i \neq b_i$. Obviously, every string of length n, is either a palindrome or an i-nonpalindrome for precisely one i. Now, recalling that we are talking about a two-symbol alphabet, we note that for every $i \in \{1, 2, \ldots, k\}$, there are precisely 2^{n-i} i-nonpalindromes, while *Pal4* executes i operations when given such a string. Also, there are 2^k palindromes of length n, and *Pal4* executes k operations on each. Hence, since we assume a uniform distribution of strings, the number of operations that algorithm *Pal4* executes on the average for strings of length n, is

$$\frac{(k \times 2^k) + \Sigma_{i=1}^{k}(i \times 2^{n-i})}{2^n} = \frac{(k \times 2^k) + \Sigma_{i=1}^{k}\Sigma_{j=k}^{n-i} 2^j}{2^n}$$

$$= \frac{(k \times 2^k) + \Sigma_{i=1}^{k}(2^{n-i+1} - 2^k)}{2^n} = \frac{2^{n+1} - 2^{k+1}}{2^n} < 2$$

In a similar fashion, one can show that if n is odd, the average number of operations is less than 4. Thus, quite surprisingly, the average-case complexity of *Pal4* is $O(1)$, that is, a constant.

TC 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.

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?

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.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

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

$$A'[1] \le A'[2] \le \cdots \le A'[n] , \tag{2.3}$$

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 **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 **for** loop in lines 1–4 that will allow you to prove in-

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

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:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

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,
>
> $$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$
>
> 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 $a_0, a_1, \ldots, a_n$.

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.)

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
a.	$\lg^k n$	n^c					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$
\begin{aligned}
f(n) &= O(g(n)) \quad \text{is like} \quad a \leq b\,, \\
f(n) &= \Omega(g(n)) \quad \text{is like} \quad a \geq b\,, \\
f(n) &= \Theta(g(n)) \quad \text{is like} \quad a = b\,, \\
f(n) &= o(g(n)) \quad \text{is like} \quad a < b\,, \\
f(n) &= \omega(g(n)) \quad \text{is like} \quad a > b\,.
\end{aligned}
$$

3-3 *Ordering by asymptotic growth rates*

a. Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. 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}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	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 $\Omega(g_i(n))$.

$2^{2^{n+1}}$	
2^{2^n}	
$(n+1)!$	
$n!$	see justification 7
e^n	see justification 1
$n \cdot 2^n$	
2^n	
$(3/2)^n$	
$(\lg n)^{\lg n} = n^{\lg \lg n}$	see identity 1
$(\lg n)!$	see justifications 2, 8
n^3	
$n^2 = 4^{\lg n}$	see identity 2
$n \lg n$ and $\lg(n!)$	see justification 6
$n = 2^{\lg n}$	see identity 3
$(\sqrt{2})^{\lg n} (= \sqrt{n})$	see identity 6, justification 3
$2^{\sqrt{2 \lg n}}$	see identity 5, justification 4
$\lg^2 n$	
$\ln n$	
$\sqrt{\lg n}$	
$\ln \ln n$	see justification 5
$2^{\lg^* n}$	
$\lg^* n$ and $\lg^*(\lg n)$	see identity 7
$\lg(\lg^*) n$	
$n^{1/\lg n} (= 2)$ and 1	see identity 4

Much of the ranking is based on the following properties:

- Exponential functions grow faster than polynomial functions, which grow faster than polylogarithmic functions.
- The base of a logarithm doesn't matter asymptotically, but the base of an exponential and the degree of a polynomial do matter.

We have the following *identities*:

1. $(\lg n)^{\lg n} = n^{\lg \lg n}$ because $a^{\log_b c} = c^{\log_b a}$.
2. $4^{\lg n} = n^2$ because $a^{\log_b c} = c^{\log_b a}$.
3. $2^{\lg n} = n$.
4. $2 = n^{1/\lg n}$ by raising identity 3 to the power $1/\lg n$.
5. $2^{\sqrt{2 \lg n}} = n^{\sqrt{2/\lg n}}$ by raising identity 4 to the power $\sqrt{2 \lg n}$.
6. $\left(\sqrt{2}\right)^{\lg n} = \sqrt{n}$ because $\left(\sqrt{2}\right)^{\lg n} = 2^{(1/2)\lg n} = 2^{\lg \sqrt{n}} = \sqrt{n}$.
7. $\lg^*(\lg n) = (\lg^* n) - 1$.

The following *justifications* explain some of the rankings:

1. $e^n = 2^n (e/2)^n = \omega(n 2^n)$, since $(e/2)^n = \omega(n)$.
2. $(\lg n)! = \omega(n^3)$ by taking logs: $\lg(\lg n)! = \Theta(\lg n \lg \lg n)$ by Stirling's approximation, $\lg(n^3) = 3 \lg n$. $\lg \lg n = \omega(3)$.

3. $(\sqrt{2})^{\lg n} = \omega\left(2^{\sqrt{2\lg n}}\right)$ by taking logs: $\lg(\sqrt{2})^{\lg n} = (1/2)\lg n$, $\lg 2^{\sqrt{2\lg n}} = \sqrt{2\lg n}$. $(1/2)\lg n = \omega(\sqrt{2\lg n})$.
4. $2^{\sqrt{2\lg n}} = \omega(\lg^2 n)$ by taking logs: $\lg 2^{\sqrt{2\lg n}} = \sqrt{2\lg n}$, $\lg\lg^2 n = 2\lg\lg n$. $\sqrt{2\lg n} = \omega(2\lg\lg n)$.
5. $\ln\ln n = \omega(2^{\lg^* n})$ by taking logs: $\lg 2^{\lg^* n} = \lg^* n$. $\lg\ln\ln n = \omega(\lg^* n)$.
6. $\lg(n!) = \Theta(n\lg n)$ (equation (3.18)).
7. $n! = \Theta(n^{n+1/2}e^{-n})$ by dropping constants and low-order terms in equation (3.17).
8. $(\lg n)! = \Theta((\lg n)^{\lg n+1/2}e^{-\lg n})$ by substituting $\lg n$ for n in the previous justification. $(\lg n)! = \Theta((\lg n)^{\lg n+1/2}n^{-\lg e})$ because $a^{\log_b c} = c^{\log_b a}$.

b. The following $f(n)$ is nonnegative, and for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

$$f(n) = \begin{cases} 2^{2^{n+2}} & \text{if } n \text{ is even}, \\ 0 & \text{if } n \text{ is odd}. \end{cases}$$

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

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

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

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

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

6.11. Analyze the worst-case time and space complexities of the breadth-first algorithm for checking whether a given tree is balanced you were asked to design in Exercise 4.7. Compare them with the complexities of a straightforward depth-first algorithm for the same problem, which uses the algorithm for tree isomorphism that you were asked to design in Exercise 4.6 as a subroutine applied to the offspring of every node containing a binary operation.

4.7. Design an algorithm that checks whether an expression is balanced, given its tree representation. (Hint: perform breadth-first traversal of the tree.)

An arithmetic expression formed by non-negative integers and the standard unary operation "−" and the binary operations "+", "−", "×", and "/", can be represented by a binary tree as follows:

- An integer I is represented by a leaf containing I.
- The expression $-E$, where E is an expression, is represented by a tree whose root contains "−" and its single offspring is the root of a subtree representing the expression E.
- The expression $E * F$, where E and F are expressions and "$*$" is a binary operation, is represented by a tree whose root contains "$*$", its first offspring is the root of a subtree representing the expression E and its second offspring is the root of a subtree representing F.

Note that the symbol "−" stands for both unary and binary operations, and the nodes of the tree containing this symbol may have outdegree either 1 or 2.

An expression E is said to be *balanced*, if every binary operation in it is applied to two isomorphic expressions. For example, the expressions -5, $(1+2)*(3+5)$ and $((-3)/(-4))/((-1)/(-100))$ are balanced, while $12+(3+2)$ and $(-3)*(-3)$ are not.

We say that two arithmetic expressions E and F are *isomorphic*, if E can be obtained from F by replacing some non-negative integers by others. For example, the expressions $(2+3)\times 6-(-4)$ and $(7+0)\times 6-(-9)$ are isomorphic, but none of them is isomorphic to any of $(-2+3)\times 6-(-4)$ and $(7+0)+6-(-9)$.