Introduction to Algorithms September 24, 2004
Massachusetts Institute of Technology 6.046J/18.410J
Professors Piotr Indyk and Charles E. Leiserson Handout 7

Problem Set 1 Solutions

Exercise 1-1. Do Exercise 2.3-7 on page 37 in CLRS.

Solution:

The following algorithm solves the problem:

1. Sort the elements in S using mergesort.
2. Remove the last element from S. Let y be the value of the removed element.
3. If S is nonempty, look for $z = x - y$ in S using binary search.
4. If S contains such an element z, then STOP, since we have found y and z such that $x = y + z$. Otherwise, repeat Step 2.
5. If S is empty, then no two elements in S sum to x.

Notice that when we consider an element y_i of S during ith iteration, we don't need to look at the elements that have already been considered in previous iterations. Suppose there exists $y_j \in S$, such that $x = y_i + y_j$. If $j < i$, i.e. if y_j has been reached prior to y_i, then we would have found y_i when we were searching for $x - y_j$ during jth iteration and the algorithm would have terminated then.

Step 1 takes $\Theta(n \lg n)$ time. Step 2 takes $O(1)$ time. Step 3 requires at most $\lg n$ time. Steps 2–4 are repeated at most n times. Thus, the total running time of this algorithm is $\Theta(n \lg n)$. We can do a more precise analysis if we notice that Step 3 actually requires $\Theta(\lg(n - i))$ time at ith iteration. However, if we evaluate $\sum_{i=1}^{n-1} \lg(n - i)$, we get $\lg(n-1)!$, which is $\Theta(n \lg n)$. So the total running time is still $\Theta(n \lg n)$.

Exercise 1-2. Do Exercise 3.1-3 on page 50 in CLRS.

Exercise 1-3. Do Exercise 3.2-6 on page 57 in CLRS.

Exercise 1-4. Do Problem 3-2 on page 58 of CLRS.

Problem 1-1. Properties of Asymptotic Notation

Prove or disprove each of the following properties related to asymptotic notation. In each of the following assume that f, g, and h are asymptotically nonnegative functions.

(a) $f(n) = O(g(n))$ and $g(n) = O(f(n))$ implies that $f(n) = \Theta(g(n))$.

Solution:
This Statement is True.

Since $f(n) = O(g(n))$, then there exists an n_0 and a c such that for all $n \geq n_0$, $f(n) \leq cg(n)$. Similarly, since $g(n) = O(f(n))$, there exists an n_0' and a c' such that for all $n \geq n_0'$, $g(n) \leq c'f(n)$. Therefore, for all $n \geq \max(n_0, n_0')$, $\frac{1}{c'}g(n) \leq f(n) \leq cg(n)$. Hence, $f(n) = \Theta(g(n))$.

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

Solution:
This Statement is True.

For all $n \geq 1$, $f(n) \leq \max(f(n), g(n))$ and $g(n) \leq \max(f(n), g(n))$. Therefore:

$$f(n) + g(n) \leq \max(f(n), g(n)) + \max(f(n), g(n)) \leq 2\max(f(n), g(n))$$

and so $f(n) + g(n) = O(\max(f(n), g(n)))$. Additionally, for each n, either $f(n) \geq \max(f(n), g(n))$ or else $g(n) \geq \max(f(n), g(n))$. Therefore, for all $n \geq 1$, $f(n) + g(n) \geq \max(f(n), g(n))$ and so $f(n) + g(n) = \Omega(\max(f(n), g(n)))$. Thus, $f(n) + g(n) = \Theta(\max(f(n), g(n)))$.

(c) Transitivity: $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies that $f(n) = O(h(n))$.

Solution:
This Statement is True.

Since $f(n) = O(g(n))$, then there exists an n_0 and a c such that for all $n \geq n_0$, $f(n) \leq cg(n)$. Similarly, since $g(n) = O(h(n))$, there exists an n_0' and a c' such that for all $n \geq n_0'$, $g(n) \leq c'h(n)$. Therefore, for all $n \geq \max(n_0, n_0')$, $f(n) \leq cc'h(n)$. Hence, $f(n) = O(h(n))$.

(d) $f(n) = O(g(n))$ implies that $h(f(n)) = O(h(g(n))$.

Solution:
This Statement is False.

We disprove this statement by giving a counter-example. Let $f(n) = n$ and $g(n) = 3n$ and $h(n) = 2^n$. Then $h(f(n)) = 2^n$ and $h(g(n)) = 8^n$. Since 2^n is not $O(8^n)$, this choice of f, g and h is a counter-example which disproves the theorem.

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

Solution:
This Statement is True.

Let $h(n) = o(f(n))$. We prove that $f(n) + o(f(n)) = \Theta(f(n))$. Since for all $n \geq 1$, $f(n) + h(n) \geq f(n)$, then $f(n) + h(n) = \Omega(f(n))$.
Since $h(n) = o(f(n))$, then there exists an n_0 such that for all $n > n_0$, $h(n) \leq f(n)$. Therefore, for all $n > n_0$, $f(n) + h(n) \leq 2f(n)$ and so $f(n) + h(n) = O(f(n))$. Thus, $f(n) + h(n) = \Theta(f(n))$.

(f) $f(n) \neq o(g(n))$ and $g(n) \neq o(f(n))$ implies $f(n) = \Theta(g(n))$.

Solution:
This Statement is False.

We disprove this statement by giving a counter-example. Consider $f(n) = 1 + cos(\pi * n)$ and $g(n) = 1 - cos(\pi * n)$.
For all even values of n, $f(n) = 2$ and $g(n) = 0$, and there does not exist a c_1 for which $f(n) \leq c_1 g(n)$. Thus, $f(n)$ is not $o(g(n))$, because if there does not exist a c_1 for which $f(n) \leq c_1 g(n)$, then it cannot be the case that for *any* $c_1 > 0$ and sufficiently large n, $f(n) < c_1 g(n)$.
For all odd values of n, $f(n) = 0$ and $g(n) = 2$, and there does not exist a c for which $g(n) \leq c f(n)$. By the above reasoning, it follows that $g(n)$ is not $o(f(n))$. Also, there cannot exist $c_2 > 0$ for which $c_2 g(n) \leq f(n)$, because we could set $c = 1/c_2$ if such a c_2 existed.
We have shown that there do not exist constants $c_1 > 0$ and $c_2 > 0$ such that $c_2 g(n) \leq f(n) \leq c_1 g(n)$. Thus, $f(n)$ is not $\Theta(g(n))$.

Problem 1-2. Computing Fibonacci Numbers

The Fibonacci numbers are defined on page 56 of CLRS as

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for } n \geq 2 . \end{aligned}$$

In Exercise 1-3, of this problem set, you showed that the nth Fibonacci number is

$$F_n = \frac{\phi^n - \widehat{\phi}^n}{\sqrt{5}} ,$$

where ϕ is the golden ratio and $\widehat{\phi}$ is its conjugate.

A fellow 6.046 student comes to you with the following simple recursive algorithm for computing the nth Fibonacci number.

```
FIB(n)
 1  if n = 0
 2     then return 0
 3  elseif n = 1
 4     then return 1
 5  return FIB(n − 1) + FIB(n − 2)
```

This algorithm is correct, since it directly implements the definition of the Fibonacci numbers. Let's analyze its running time. Let $T(n)$ be the worst-case running time of $\text{FIB}(n)$.[1]

(a) Give a recurrence for $T(n)$, and use the substitution method to show that $T(n) = O(F_n)$.

Solution: The recurrence is: $T(n) = T(n-1) + T(n-2) + 1$.
We use the substitution method, inducting on n. Our Induction Hypothesis is: $T(n) \leq cF_n - b$.
To prove the inductive step:

$$\begin{aligned} T(n) &\leq cF_{n-1} + cF_{n-2} - b - b + 1 \\ &\leq cF_n - 2b + 1 \end{aligned}$$

Therefore, $T(n) \leq cF_n - b + 1$ provided that $b \geq 1$. We choose $b = 2$ and $c = 10$. For the base case consider $n \in \{0, 1\}$ and note the running time is no more than $10 - 2 = 8$.

(b) Similarly, show that $T(n) = \Omega(F_n)$, and hence, that $T(n) = \Theta(F_n)$.

Solution: Again the recurrence is: $T(n) = T(n-1) + T(n-2) + 1$.
We use the substitution method, inducting on n. Our Induction Hypothesis is: $T(n) \geq F_n$.
To prove the inductive step:

$$\begin{aligned} T(n) &\geq F_{n-1} + F_{n-2} + 1 \\ &\geq F_n + 1 \end{aligned}$$

Therefore, $T(n) \leq F_n$. For the base case consider $n \in \{0, 1\}$ and note the running time is no less than 1.

[1] In this problem, please assume that all operations take unit time. In reality, the time it takes to add two numbers depends on the number of bits in the numbers being added (more precisely, on the number of memory words). However, for the purpose of this problem, the approximation of unit time addition will suffice.

Professor Grigori Potemkin has recently published an improved algorithm for computing the nth Fibonacci number which uses a cleverly constructed loop to get rid of one of the recursive calls. Professor Potemkin has staked his reputation on this new algorithm, and his tenure committee has asked you to review his algorithm.

```
FIB'(n)
1  if n = 0
2     then return 0
3  elseif n = 1
4     then return 1
5   sum ← 1
6   for k ← 1 to n − 2
7       do  sum ← sum + FIB'(k)
8  return sum
```

Since it is not at all clear that this algorithm actually computes the nth Fibonacci number, let's prove that the algorithm is correct. We'll prove this by induction over n, using a loop invariant in the inductive step of the proof.

(c) State the induction hypothesis and the base case of your correctness proof.

Solution: To prove the algorithm is correct, we are inducting on n. Our induction hypothesis is that for all $n < m$, $Fib'(n)$ returns F_n, the nth Fibonacci number.

Our base case is $m = 2$. We observe that the first four lines of Potemkin guarantee that $Fib'(n)$ returns the correct value when $n < 2$.

(d) State a loop invariant for the loop in lines 6-7. Prove, using induction over k, that your "invariant" is indeed invariant.

Solution: Our loop invariant is that after the $k = i$ iteration of the loop,

$$sum = F_{i+2} \ .$$

We prove this induction using induction over k. We assume that after the $k = (i-1)$ iteration of the loop, $sum = F_{i+1}$. Our base case is $i = 1$. We observe that after the first pass through the loop, $sum = 2$ which is the 3rd Fibonacci number.

To complete the induction step we observe that if $sum = F_{i+1}$ after the $k = (i-1)$ and if the call to $Fib'(i)$ on Line 7 correctly returns F_i (by the induction hypothesis of our correctness proof in the previous part of the problem) then after the $k = i$ iteration of the loop $sum = F_{i+2}$. This follows immediately form the fact that $F_i + F_{i+1} = F_{i+2}$.

(e) Use your loop invariant to complete the inductive step of your correctness proof.

Solution: To complete the inductive step of our correctness proof, we must show that if $Fib'(n)$ returns F_n for all $n < m$ then $Fib'(m)$ returns m. From the previous part we know that if $Fib'(n)$ returns F_n for all $n < m$, then at the end of the $k = i$ iteration of the loop $sum = F_{i+2}$. We can thus conclude that after the $k = m - 2$ iteration of the loop, $sum = F_m$ which completes our correctness proof.

(f) What is the asymptotic running time, $T'(n)$, of FIB$'(n)$? Would you recommend tenure for Professor Potemkin?

Solution: We will argue that $T'(n) = \Omega(F_n)$ and thus that Potemkin's algorithm, Fib' does not improve upon the assymptotic performance of the simple recurrsive algorithm, Fib. Therefore we would not recommend tenure for Professor Potemkin.

One way to see that $T'(n) = \Omega(F_n)$ is to observe that the only constant in the program is the 1 (in lines 5 and 4). That is, in order for the program to return F_n lines 5 and 4 must be executed a total of F_n times.

Another way to see that $T'(n) = \Omega(F_n)$ is to use the substitution method with the hypothesis $T'(n) \geq F_n$ and the recurrence $T'(n) = cn + \sum_{k=1}^{n-2} T'(k)$.

Problem 1-3. Polynomial multiplication

One can represent a polynomial, in a symbolic variable x, with degree-bound n as an array $P[0\,..\,n]$ of coefficients. Consider two linear polynomials, $A(x) = a_1 x + a_0$ and $B(x) = b_1 x + b_0$, where a_1, a_0, b_1, and b_0 are numerical coefficients, which can be represented by the arrays $[a_0, a_1]$ and $[b_0, b_1]$, respectively. We can multiply A and B using the four coefficient multiplications

$$\begin{aligned} m_1 &= a_1 \cdot b_1 \,, \\ m_2 &= a_1 \cdot b_0 \,, \\ m_3 &= a_0 \cdot b_1 \,, \\ m_4 &= a_0 \cdot b_0 \,, \end{aligned}$$

as well as one numerical addition, to form the polynomial

$$C(x) = m_1 x^2 + (m_2 + m_3)x + m_4 \,,$$

which can be represented by the array

$$[c_0, c_1, c_2] = [m_4, m_3 + m_2, m_1] \,.$$

(a) Give a divide-and-conquer algorithm for multiplying two polynomials of degree-bound n, represented as coefficient arrays, based on this formula.

Solution:

We can use this idea to recursively multiply polynomials of degree $n-1$, where n is a power of 2, as follows:

Let $p(x)$ and $q(x)$ be polynomials of degree $n-1$, and divide each into the upper $n/2$ and lower $n/2$ terms:

$$\begin{aligned} p(x) &= a(x)x^{n/2} + b(x) \ , \\ q(x) &= c(x)x^{n/2} + d(x) \ , \end{aligned}$$

where $a(x)$, $b(x)$, $c(x)$, and $d(x)$ are polynomials of degree $n/2 - 1$. The polynomial product is then

$$\begin{aligned} p(x)q(x) &= (a(x)x^{n/2} + b(x))(c(x)x^{n/2} + d(x)) \\ &= a(x)c(x)x^n + (a(x)d(x) + b(x)c(x))x^{n/2} + b(x)d(x) \ . \end{aligned}$$

The four polynomial products $a(x)c(x)$, $a(x)d(x)$, $b(x)c(x)$, and $b(x)d(x)$ are computed recursively.

(b) Give and solve a recurrence for the worst-case running time of your algorithm.

Solution:

Since we can perform the dividing and combining of polynomials in time $\Theta(n)$, recursive polynomial multiplication gives us a running time of

$$\begin{aligned} T(n) &= 4T(n/2) + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$

(c) Show how to multiply two linear polynomials $A(x) = a_1 x + a_0$ and $B(x) = b_1 x + b_0$ using only *three* coefficient multiplications.

Solution:

We can use the following 3 multiplications:

$$\begin{aligned} m_1 &= (a+b)(c+d) = ac + ad + bc + bd \ , \\ m_2 &= ac \ , \\ m_3 &= bd \ , \end{aligned}$$

so the polynomial product is

$$(ax+b)(cx+d) = m_2 x^2 + (m_1 - m_2 - m_3)x + m_3 \ .$$

(d) Give a divide-and-conquer algorithm for multiplying two polynomials of degree-bound n based on your formula from part (c).

Solution:
The algorithm is the same as in part (a), except for the fact that we need only compute three products of polynomials of degree $n/2$ to get the polynomial product.

(e) Give and solve a recurrence for the worst-case running time of your algorithm.

Solution:
Similar to part (b):

$$\begin{aligned} T(n) &= 3T(n/2) + \Theta(n) \\ &= \Theta(n^{\lg 3}) \\ &\approx \Theta(n^{1.585}) \end{aligned}$$

Alternative solution Instead of breaking a polynomial $p(x)$ into two smaller polynomials $a(x)$ and $b(x)$ such that $p(x) = a(x) + x^{n/2}b(x)$, as we did above, we could do the following:
Collect all the *even* powers of $p(x)$ and substitute $y = x^2$ to create the polynomial $a(y)$. Then collect all the *odd* powers of $p(x)$, factor out x and substitute $y = x^2$ to create the second polynomial $b(y)$. Then we can see that

$$p(x) = a(y) + x \cdot b(y)$$

Both $a(y)$ and $b(y)$ are polynomials of (roughly) half the original size and degree, and we can proceed with our multiplications in a way analogous to what was done above. Notice that, at each level k, we need to compute $y_k = y_{k-1}^2$ (where $y_0 = x$), which takes time $\Theta(1)$ per level and does not affect the asymptotic running time.