Problem Set 1 Solutions

Exercise 1-1. Do Exercise 2.3-5 on page 37 in CLRS.

Solution:

Procedure BINARY-SEARCH takes a sorted array A, a value v, and a range $[\mathit{low} \,..\, \mathit{high}]$ of the array, in which the value v should be searched for. The procedure compares v to the midpoint of the range and decides to eliminate half the range from further consideration. Both iterative and recursive versions are given. These versions should be initially called with the range $[1 \,..\, \mathit{length}[A]]$.

```
ITERATIVE-BINARY-SEARCH$(A, v, low, high)$
1  while $low \le high$
2       do  $mid \leftarrow \lfloor (low + high)/2 \rfloor$
3  if $v = A[mid]$
4     then return $mid$
5  if $v > A[mid]$
6     then $low \leftarrow mid + 1$
7     else  $high \leftarrow mid - 1$
8  return NIL
```

```
RECURSIVE-BINARY-SEARCH$(A, v, low, high)$
1  if $low > high$
2     then return NIL
3  $mid \leftarrow \lfloor (low + high)/2 \rfloor$
4  if $v = A[mid]$
5     then return $mid$
6  if $v > A[mid]$
7     then return RECURSIVE-BINARY-SEARCH$(A, v, mid + 1, high)$
8     else  return RECURSIVE-BINARY-SEARCH$(A, v, low, mid - 1)$
```

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $\mathit{low} > \mathit{high}$) and terminate successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

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

Solution:

By the definition of Θ-notation (CLRS p. 42) we must show that there exist positive constants c_1, c_2, and n_0 such that for $n > n_0$,

$$0 \le c_1(f(n) + g(n)) \le \max(f(n), g(n)) \le c_2(f(n) + g(n)) .$$

Without loss of generality, let $\max(f(n), g(n)) = f(n)$. Clearly, $f(n) + g(n) \le 2f(n)$. Also, since $g(n) \ge 0$, $f(n) + g(n) \ge f(n)$. Thus, selecting $c_1 = 1/2$ and $c_2 = 1$ and $n_0 = 1$ satisfies the definition.

Exercise 1-4. Do Exercise 4.1-6 on page 67 in CLRS.

Solution:

Let $n = 2^m$ or, equivalently, $m = \log_2 n$. The recurrence becomes

$$T(2^m) = 2T(2^{m/2}) + 1 .$$

We will need one more substitution: Let $S(m) = T(2^m)$. The recurrence then becomes:

$$S(m) = 2S(m/2) + 1 \; .$$

By the master method, $S(m) = \Theta(m)$. Equivalently, in terms of T we have $T(2^m) = \Theta(m)$. Going back to n ($= 2^m$), we get

$$T(n) = \Theta(\log_2 n)$$

Exercise 1-5. Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, …, $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	e^n	$\lg^*(n^n)$
3^n	n^3	$\lg^2 n$	$\lg(n!)$	$n^{2+\sin n}$	$n^{1/\lg n}$
1	$\lg^*(\lg n)$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	$\ln \ln n$
$3^{\lg n}$	$(\lg n)^{\lg n}$	2^n	$n \lg n$	$\sum_{k=1}^{n} \frac{1}{k}$	$\prod_{k=2}^{n} \left(1 - \frac{1}{k}\right)$

Solution:

The ranking is based on the following facts:

- exponential functions grow faster than polynomial functions, which grow faster than logarithmic functions;
- the base of a logarithm does not matter asymptotically, but the base of an exponential and the degree of a polynomial do matter.

In addition several identities are helpful:

1. $(\lg n)^{\lg n} = n^{\lg \lg n}$
2. $2^{\lg n} = n$
3. $2 = n^{1/\lg n}$ (raise the previous one to the power $1/\lg n$)
4. $2^{\sqrt{2 \lg n}} = n^{\sqrt{2/\lg n}}$ (raise the previous one to the power $\sqrt{2 \lg n}$)
5. $(\sqrt{2})^{\lg n} = \sqrt{n}$
6. $\lim_{n \to \infty} (1 + x/n)^n = e^n$

Finally Stirling's approximation bounds are useful in ranking expression with factorials:

- $\lg(n!) = \Theta(n \lg n)$
- $n! = \Theta(n^{n+1/2}e^{-n})$

So here's the ranking (listed from left to right by row)

$2^{2^{n+1}}$	2^{2^n}	$n!$	e^n	$n \cdot 2^n$	2^n
$(\frac{3}{2})^n$	$n^{\lg\lg n}$	$(\lg n)^{\lg n}$	n^3	n^2	$\sum_{k=2}^{n}(k/\lg k)$
$\lg(n!)$	$n \lg n$	$2^{\lg n}$	$(\sqrt{2})^{\lg n}$	$\sum_{k=1}^{\lg n}(4/3)^k$	$\lg^2 n$
$\ln n$	$\sum_{k=1}^{n} \frac{k^2+3k+1}{3k^3+5k^2+k+2}$	$\ln\ln n$	$\lg^*(n^n)$	$\lg^*(\lg n) \lg(\lg^*)n$	
$\sum_{k=1}^{n} e^k/k!$	$n^{1/\lg n}$	1	$\sum_{k=1}^{n} \frac{k^2+3k+1}{3k^4+5k^2+k+2}$	$\prod_{k=2}^{n}(1-1/k)$	

The oscillating function $n^{2+\sin n}$ does not fit in the ranking because although $n^{2+\sin n} = \Omega(n)$ and $n^3 = \Omega(n^{2+\sin n})$, it is not Ω-related to n^2.

The equivalence classes determined by the Θ relationship are:

1. $\{1, \sum_{k=1}^{n} e^k/k!, n^{1/\lg n}, \sum_{k=1}^{n} \frac{k^2+3k+1}{3k^4+5k^2+k+2},\}$
2. $\{\lg^*(n^n), \lg^*(\lg n)\}$
3. $\{n, 2^{\lg n}\}$
4. $\{n \lg n, \lg(n!)\}$
5. $\{(\lg n)^{\lg n}, n^{\lg\lg n}\}$
6. $\{\sum_{k=1}^{n} \frac{k^2+3k+1}{3k^4+5k^2+k+2}, \ln n\}$

Problem 1-1. Asymptotic notation for multivariate functions

The generalization of asymptotic notation from one variable to multiple variables is surprisingly tricky. One proper generalization of O-notation for two variables is the following:

Definition 1

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

Consider the following alternative definition:

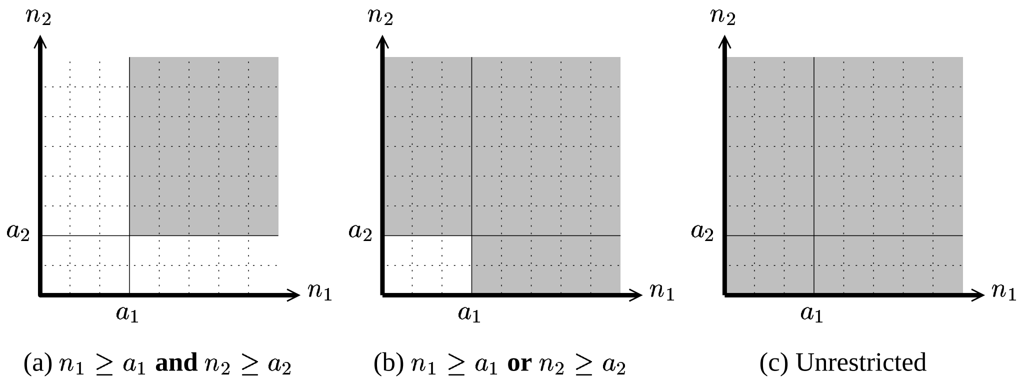
Definition 2

$$O'(g(m,n)) = \{f(m,n) : \text{there exist positive constants } m_0,\ n_0, \text{ and } c \text{ such that } 0 \le f(m,n) \le cg(m,n) \text{ for all } m \ge m_0 \text{ and } n \ge n_0\}\ .$$

(a) Explain why Definition 2 is a "bogus" definition. That is, what anomalies does the definition of O' permit that are counterintuitive? You may find it helpful to illustrate your answer with a diagram of relevant regions of the $m \times n$ plane.

Solution:

The distinction between these two interpretations can best be illustrated with a diagram of the space parameterized by m and n; see Figure 1. The definition of O' requires that the inequality hold in the shaded rectangle in Figure 1(a), defined by $m \geq m_0$ and $n \geq n_0$, leaving the strips $m < m_0$ and $n < n_0$ uncovered. In contrast, the definition of O requires in addition that the inequality hold for sufficiently large values in those strips, i.e., for the shaded region in Figure 1(b). Ideally, we would also hope for the inequality to hold for all values of m and n, as in Figure 1(c); we call this the **unrestricted interpretation**.



(a) $n_1 \geq a_1$ **and** $n_2 \geq a_2$ (b) $n_1 \geq a_1$ **or** $n_2 \geq a_2$ (c) Unrestricted

Figure 1: Three candidate regions in which a statement about a two-variable function should hold.

The definition of O' is bogus because it allows $f(m, n)$ to be outside $c \cdot g(m, n)$ for infinitely many pairs of values (m, n). Recall that for univariate functions, an equivalent interpretation of what $f(n) = O(g(n))$ means is the following: there exists a constant c such that $f(n) \leq c \cdot g(n)$ for all but finitely many[1] values of n. We should thus expect the definition of O notation in multivariate functions to allow for only finitely many points (tuples) to be outside the stated range.

Remarkably, famous computer scientists have used Definition 2 without being aware of its deficiencies. Nevertheless, their theorems and analyses carry over to Definition 1, because the functions they analyzed satisfy two key properties.

The first property is "monotonicity":

[1]or equivalently, "for infinitely many values of n"

Definition 3 A two-variable function $f(m,n)$ is ***monotonically increasing*** if

$$f(m,n) \leq f(m+1,n)$$

and

$$f(m,n) \leq f(m,n+1)$$

for all nonnegative m and n.

(b) Explain this definition in plain English.

Solution:

A function $f(m,n)$ is *monotonically increasing* if whenever either (or both) of the function's arguments increase, the function's value either increases or remains constant, but never decreases.

The second property is more complicated:

Definition 4 A two-variable function $g(m,n)$ is ***multiplicatively separable*** if there exist a constant $L \geq 0$ and two one-variable functions, $a(m)$ and $b(n)$, such that whenever $m \geq 0$, $n \geq 0$, and $g(m,n) \geq L$, we have

$$g(m,n) \leq a(m) \cdot g(m-1,n)$$

and

$$g(m,n) \leq b(n) \cdot g(m,n-1)\ .$$

Intuitively, increasing one argument of a multiplicatively separable function g increases the value of g by at most a multiplicative factor which can be bounded in terms of that argument itself, independent of the other argument.

(c) For each of the following functions $g(m,n)$, argue that g is multiplicatively separable.

i. $g(m,n) = m+n$
ii. $g(m,n) = m^2 n$
iii. $g(m,n) = 2^m + 2^n$
v. $g(m,n) = 2^{m+n}$
vi. $g(m,n) = 2^{2^m+2^n}$

Solution:

We need to show that there exist functions $a(m)$ and $b(n)$ with the desired properties.

i. $a(m) = 2$.
$b(n) = 2$.

ii. $a(m) = 4$.
$b(n) = 2$.
However, this is true only if we assume that $m \geq 1$ and $n \geq 1$,

iii. $a(m) = 2$.
$b(n) = 2$.

v. $a(m) = 2$.
$b(n) = 2$.

vi. $a(m) = 2^{2^{m-1}}$.
$b(n) = 2^{2^{n-1}}$.

(d) Prove that the following two functions are not multiplicatively separable:

i. $g(m, n) = 2^{m \cdot n}$

ii. $g(m, n) = m^n$

Solution:

i. Proof by contradiction: Suppose the function was multiplicatively separable. Then we would have:

$$2^{m \cdot n} \leq a(m) \cdot 2^{(m-1)n} \Rightarrow a(m) \geq 2^n$$

And so $a(m)$ would not be bounded in terms of the argument itself, independent of the other argument
Indeed, in this case, $f(m, n) = O(g(m, n))$ would not necessarily hold under the "or" interpretation if it holds under the "and" interpretation. For example, consider the function

$$f(m, n) = \begin{cases} 2^{m \cdot n} & \text{for } m \geq 2, n \geq 2 \\ 2^{m \cdot 2} & \text{for } n = 1 \\ 2^{2 \cdot n} & \text{for } m = 1 \end{cases}$$

Let $m_0 = n_0 = 2$. Then $f(m, n) = O(g(m, n))$ holds whenever $m \geq m_0$ and $n \geq n_0$ but is not necessarily true under the "or" definition.

ii. Similarly,

$$m^n \leq a(m)(m-1)^n \Rightarrow a(m) \geq \left(\frac{m}{m-1}\right)^n$$

Again $a(m)$ would not be bounded in terms of the argument itself, independent of the other argument[2].
Similarly, if $f(m, n) = O(g(m, n))$ is true under the "and" interpretation, it is not necessarily true under the "or" interpretation. For example, consider the function

[2]notice that $\left(\frac{m}{m-1}\right)^n$ grows without bounds as n grows

$$f(m,n) = \begin{cases} m^n & \text{for } m \geq 2 \\ 2^n & \text{for } m = 1 \end{cases}$$

with $m_0 = n_0 = 2$. Then $f(m,n) = O(g(m,n))$ holds whenever $m \geq m_0$ and $n \geq n_0$ but is not necessarily true under the "or" definition.

Suppose that f is monotonically increasing and g is multiplicatively separable. Suppose further that $f(m,n) = O'(g(m,n))$, that is, there exist positive constants m_0, n_0, and c such that $0 \leq f(m,n) \leq cg(m,n)$ for all $m \geq m_0$ and $n \geq n_0$.

(e) Prove that there exists a constant $r \geq 0$ such that

$$f(m,n) \leq r \cdot g\left(\max(m,m_0), \max(n,n_0)\right)$$

for all nonnegative m and n.

Solution:

By monotonicity of f,

$$f(m,n) \leq f\Big(\max(m,m_0), n\Big) \leq f\Big(\max(m,m_0), \max(n,n_0)\Big).$$

Because $f(m,n) = O(g(m,n))$ holds under the 'and' interpretation,

$$f\Big(\max(m,m_0), \max(n,n_0)\Big) \leq c \cdot g\Big(\max(m,m_0), \max(n,n_0)\Big).$$

Putting these two inequalities together,

$$f(m,n) \leq c \cdot g\Big(\max(m,m_0), \max(n,n_0)\Big),$$

so $r = c$ suffices.

(f) Prove that there exists a constant $s \geq 0$ such that

$$g\left(\max(m,m_0), \max(n,n_0)\right) \leq s \cdot g(m,n)$$

for all nonnegative m and n.

Solution:

Suppose for example that $m < m_0$ and $n \geq n_0$. (The other cases are similar.)
By repeated application of multiplicative separability,

$$\begin{aligned}
g(m_0,n) &\leq a(m_0) \cdot g(m_0 - 1, n) \\
&\leq a(m_0) \cdot a(m_0 - 1) \cdot g(m_0 - 2, n) \\
&\leq \cdots \\
&\leq a(m_0) \cdot a(m_0 - 1) \cdots a(m+1) \cdot g(m,n) \\
&\leq \max\{1, a(m_0)\} \cdot \max\{1, a(m_0 - 1)\} \cdots \max\{1, a(0)\} \cdot g(m,n).
\end{aligned}$$

(The $\max$'s are necessary to deal with the possibility that a takes on values less than 1.) Thus,

$$s = \max\{1, a(m_0)\} \cdot \max\{1, a(m_0 - 1)\} \cdots \max\{1, a(0)\}$$

suffices in this case.

(g) Conclude that $f(m, n) = O(g(m, n))$.

Solution:

Let $t = r \cdot s$. By parts (a) and (b), for all m and n,

$$f(m, n) \leq r \cdot s \cdot g(m, n) = t \cdot g(m, n).$$

Therefore, $f(m, n) = O(g(m, n))$ holds under the unrestricted interpretation and thus in particular the 'or' definition.

(h) (*Extra credit.*) Give a proper generalization of Ω to two variables. Justify your definition.

Solution:

Awaiting ideas from students...

Problem 1-2. Tree Traversal

The following pseudocode is a standard recursive tree-traversal algorithm for counting the number of nodes in a tree R. The initial call is COUNT-NODES$(root[R])$.

```
COUNT-NODES(x)
1  if x = NIL
2     then return 0
3     else  return 1 + COUNT-NODES(left[x])
                     + COUNT-NODES(right[x])
```

Define *size*(x) to be the number of nodes in the subtree rooted at node $x \in R$, and let $T(x)$ denote the worst-case running time of COUNT-NODES(x).

(a) Give a recurrence for $T(x)$ in terms of *left*(x) and *right*(x).

Solution:

$$T(x) = T(\textit{left}(x)) + T(\textit{right}(x)) + \Theta(1)$$

(b) Use the substitution method to prove that $T(x) = O(\textit{size}(x))$.

Solution:

For convenience, let $|y|$ denote $size(y)$, that is, the size of the tree whose root is node y.

In order to prove that $T(x) = O(|x|)$, we need to show that there exists a constant c such that $T(x) \leq c \cdot |x|$.

Proof. Let k be an upper bound on the $\Theta(1)$ term [3]. Assume that there exists some constant $c \geq k$ such that:

for all trees y with $|y| < |x|$, $T(y) \leq c \cdot |y|$.

That is, we assume that the statement holds for all trees whose size is less than $|x|$.

We want to prove that $T(x) \leq c \cdot |x|$.

From part (a), we have: $T(x) = T(\textit{left}(x)) + T(\textit{right}(x)) + \Theta(1)$. Since $\textit{right}(x)$ and $\textit{left}(x)$ are smaller than x, we have

$$\begin{aligned} T(x) &= T(\textit{left}(x) + T(\textit{right}(x)) + \Theta(1) \\ &\leq c \cdot |\textit{left}(x)| + c \cdot |\textit{right}(x)| + k \\ &= c \cdot [|\textit{left}(x)| + |\textit{right}(x)|] + k \\ &= c \cdot [|x| - 1] + k \\ &= c \cdot |x| - (c - k) \\ &\leq c \cdot |x| \end{aligned}$$

□

A common compiler optimization of this code, called ***tail recursion***, is to replace one of the recursive calls with a loop, resulting in the following pseudocode:

```
COUNT-NODES-TAIL($x$)
1 $s \leftarrow 0$
2 while $x \neq$ NIL
3       do $s \leftarrow s + 1 +$ COUNT-NODES-TAIL($\textit{left}[x]$)
4          $x \leftarrow \textit{right}[x]$
5 return $s$
```

Let $\textit{right}^{i}[x]$ denote the i th right descendant of x, that is,

$$\textit{right}^{i}[x] = \begin{cases} x & \text{if } i = 0 \,, \\ \textit{right}[\textit{right}^{i-1}[x]] & \text{if } i > 0 \,. \end{cases}$$

[3]notice that the $\Theta(1)$ term does not depend on the size of the tree

Consider the loop invariant

$$s = k + \sum_{i=0}^{k-1} \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{\,i}[x]]), \tag{1}$$

where $k \geq 0$ is the number of times the **while** loop (lines 2–4) in COUNT-NODES-TAIL has been executed.

(c) Prove that if Equation (1) holds for k, then it holds for $k+1$.

Solution:

Let s' be the new value of s after one more execution of the loop. Since after k executions we have

$$s = k + \sum_{i=0}^{k-1} \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{\,i}[x]])$$

at the end of the $(k+1)$-st execution we will have

$$\begin{aligned} s' &= s + 1 + \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{k}[x]]) \\ &= \underbrace{k + \sum_{i=0}^{k-1} \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{\,i}[x]])}_{s} + 1 + \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{k}[x]]) \\ &= k + 1 + \sum_{i=0}^{k} \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{\,i}[x]]) \end{aligned}$$

So it holds for $k+1$

Let $K(x)$ be the smallest positive integer for which $\mathit{right}^{\,K(x)}[x] = \textsc{nil}$.

(d) Prove that COUNT-NODES-TAIL returns

$$K(x) + \sum_{i=0}^{K(x)-1} \textsc{Count-Nodes-Tail}(\mathit{left}[\mathit{right}^{\,i}[x]]) \,.$$

Solution:

Since $K(x)$ is the smallest positive integer for which $\mathit{right}^{\,K(x)}[x] = \textsc{nil}$, the algorithm will exit the loop after exactly $K(x)$ iterations, and return the current value of s. So if Equation (1) holds, then indeed COUNT-NODES-TAIL will return

$$K(x) + \sum_{i=0}^{K(x)-1} \text{COUNT-NODES-TAIL}(\textit{left}[\textit{right}^{\,i}[x]]) \; .$$

However, we need to prove that Equation (1) holds.

Proof.

Base Case: Consider a tree of size 1. The loop will execute only once and at the end of the loop $s = 1$, as predicted by Equation(1). So the base case holds.

Inductive step: The inductive step was taken care of in part (c).

$\square$

(e) Prove by induction that COUNT-NODES-TAIL(x) correctly computes *size*(x).

Solution:

Base case: tree of size 1. As shown above, the algorithm returns 1, which is the right answer.

Inductive step: assume that the algorithm returns the right result for all trees up to $size(x) - 1$. We want to prove that it will return the right result for any tree of $size(x)$ as well. Consider a tree x.
As shown in part (d), the algorithm returns

$$K(x) + \sum_{i=0}^{K(x)-1} \text{COUNT-NODES-TAIL}(\textit{left}[\textit{right}^{\,i}[x]]) \; .$$

Since all left subtrees have size at most $size(x) - 1$, we know that

$$\text{COUNT-NODES-TAIL}(\textit{left}[\textit{right}^{\,i}[x]])$$

correctly counts the number of leaves of all the left subtrees.
Since the tree x consists of $K(x)$ right-most nodes plus all their left subtrees, the algorithm returns the right result for trees of $size(x)$ as well.

Problem 1-3. Polynomial multiplication

If we have two linear polynomials $ax+b$ and $cx+d$, we can multiply them using the four coefficient multiplications

$$\begin{aligned} m_1 &= a \cdot c \, , \\ m_2 &= a \cdot d \, , \\ m_3 &= b \cdot c \, , \\ m_4 &= b \cdot d \end{aligned}$$

to form the polynomial

$$m_1 x^2 + (m_2 + m_3)x + m_4 \; .$$

(a) Give a divide-and-conquer algorithm for multiplying two polynomials of degree-bound n 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 $ax+b$ and $cx+d$ 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

$$(ax+b)(cx+d) = m_2x^2 + (m_1 - m_2 - m_3)x + m_3 \ .$$

(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.