CS 4310 HOMEWORK SET 1

PAUL MILLER

Section 2.1 Exercises

Exercise 2.1-1. Using Figure 2.2 as a model, illustrate the operation of `INSERTION-SORT` on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution: Assume A is indexed from 1. Begin with $A = [31, 41, 59, 26, 41, 58]$. Since $A[2]$ is already in the right spot relative to $A[1]$, we do nothing when $j = 2$. Similarly, since $A[3]$ is already in the right spot relative to $A[1]...A[2]$, we do nothing again when $j = 3$.

For $j = 4$, we find that $A[4] == 26$ is out of place, so we move it to its proper location in front of 31 and shift the other elements over to the right, giving $A = [26, 31, 41, 59, 41, 58]$.

For $j = 5$, we find that $A[5] == 41$ is out of place, so we move it to its proper place in front of the previous instance of 41 and shift elements over to accommodate it, giving $A = [26, 31, 41, 41, 59, 58]$.

Finally, for $j = 6$, we find $A[6] == 58$ is out of place, so we put it in its place and shift elements over to accommodate it, giving us finally $A = [26, 31, 41, 41, 58, 59]$, the sorted sequence. □

Exercise 2.1-3. Consider the searching problem:

- Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.
- 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 fulfills the three necessary properties.

Solution: The following Python code implements linear search:

Date: October 1, 2009.

```
1 def linear_search (A, v):
2     for i in range (len (A)):
3         if A[i] == v:
4             return i
5     return NIL
```

(Note that `range (len (A))` returns a list of integers from `0` to `len (A)-1`, inclusive. Also, Python arrays are indexed from 0, so an array of length n has indices $0 \ldots n-1$.)

To show correctness, we shall use the following loop invariant: $v \notin \{A[0], A[1], \ldots, A[i-1]\}$.

- Initialization: At the beginning of the first iteration, we have $i = 0$, so the statement is empty.
- Maintenance: Suppose the invariant is not maintained after the k-th iteration. We would then have some $i \leq k-1$ such that $A[i] = v$. However, in that case, the code would have returned the value i already, so there would not actually be a k-th iteration. This contradiction shows our initial assumption was incorrect, so the invariant is, in fact, maintained by the loop.
- Termination: Since the loop is a `for` loop over a finite sequence `0 ...len (A)-1`, the loop will always terminate. If the algorithm finds v in the array A, we have $A[i] = v$, so the loop invariant is true (and the algorithm returns the index `i`). Otherwise, the loop terminates after `len (A)` iterations, in which case the invariant states that $v \notin \{A[0], \ldots, A[\text{len}(A)-1]\}$, which is the whole array A, so the invariant is again true, and the algorithm returns `NIL`.

Therefore, the function `linear_search` is correct by the loop invariant. □

Section 2.2 Exercises

Exercise 2.2-1. Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

Solution: Note: This is a badly worded problem. As written, I can write $n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3/1000 - 100n^2 - 100n + 3)$. A

better rephrasing would be to find all values of c for which $n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^c)$. Since

$$\lim_{n\to\infty} \frac{n^3/1000 - 100n^2 - 100n + 3}{n^c} = \begin{cases} 1 & \text{if } c = 3, \\ 0 & \text{if } c > 3, \text{ and} \\ \infty & \text{if } c < 3, \end{cases}$$

we have that $n^3/1000-100n^2-100n+3 = \Theta(n^c)$ if and only if $c = 3$. □

Exercise 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.

Solution: The following pseudocode implements selection sort:

```
1  def index_of_min (A):
2      smallest = 0
3      i = 0
4      while i < len (A):
5          if A[i] < A[smallest]:
6              smallest = i
7          i += 1
8      return smallest
9
10 def selection_sort (A):
11     i = 0
12     while i < len (A) - 1:
13         # A[i:] indicates A[i], A[i+1], ...
14         j = index_of_min (A[i:]) + i
15
16         # Swaps A[i] and A[j]
17         A[i], A[j] = A[j], A[i]
18         i += 1
19     return A
```

This algorithm maintains the loop invariant that after i iterations, the subarray consisting of the first i elements of the array A is sorted.

Indeed, after 1 iteration, the first 1 elements of the array are sorted, since we have swapped the previous value of `A[0]` with `min (A)` (if necessary). The invariant is maintained because on the i-th iteration, we swap `A[i-1]` with `min (A[i-1:])` (if necessary).

It only needs to run for the first $n-1$ elements of A because after the first $n-1$ elements of A are sorted, the last element is guaranteed to be in the correct position. (If it were not, this would imply that the first $n-1$ elements were not in sorted order, contradicting the loop invariant.) This then shows that the array is sorted upon termination of the loop.

Since we must examine every element of an unsorted array to find the index of the minimum value, the function `index_of_min` must always examine all n elements of an n-element array. Thus, selection sort always uses

$$n + n - 1 + n - 2 + \cdots + 1 = \sum_{j=1}^{n} j = n(n+1)/2$$

comparisons to sort the array. Thus, selection sort uses $\Theta(n^2)$ comparisons in both the best and worst cases. □

Section 2.3 Exercises

Exercise 2.3-1. Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

Solution: At the bottom level, we have eight subarrays of length 1:

$$[3], [41], [52], [26], [38], [57], [9], [49].$$

These are then merged to form four sorted subarrays of length 2:

$$[3, 41], [26, 52], [38, 57], [9, 49].$$

(Note: when merging, we always merge consecutive pairs of smaller subarrays according to Figure 2-4. In reality, it does not matter which pairs of sorted subarrays we merge – this is simply the order in which the algorithm does things as presented in the text.)

These then get merged to form two subarrays of length 4:

$$[3, 26, 41, 52], [9, 38, 49, 57].$$

Finally, these are merged to form a single subarray of length 8:

$$[3, 9, 26, 38, 41, 49, 52, 57],$$

and we are done. □

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

Solution: Since $T(2) = 2 \lg 2 = 2 = 2^1$ by substitution, the base case holds. For the induction hypothesis, assume that $T(2^k) = 2^k \lg 2^k$. By the definition of our recurrence, for $n = 2^{k+1}$, we then have that

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

By the induction hypothesis, $T(2^k) = 2^k \lg 2^k$, so

$$\begin{aligned} T(2^{k+1}) &= 2(2^k \lg 2^k) + 2^{k+1} \\ &= 2^{k+1}(\lg 2^k + 1) \\ &= 2^{k+1} \lg 2^{k+1}, \end{aligned}$$

which is what we were after, so the result follows by induction. □

Exercise 2.3-5. Referring back to the searching problem (see Exercise 2.1-3), observe that 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. Binary search is an algorithm that 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 $\Theta(\lg n)$.

Solution: The following pseudocode implements a recursive binary search, returning the index of the element v if it is present, or the value `None` if it is not found:

```
1 def binary_search(A, v, low, high):
2     if low > high:
3         return None
4     mid = (low + high) / 2
5     if A[mid] < v:
```

```
6           return binary_search (A, v, mid + 1, high)
7       elif A[mid] > v:
8           return binary_search (A, v, low, mid - 1)
9       else:
10          return mid
```

Since the algorithm eliminates at least $1/2$ of the possible indices under consideration at each step, after at most $\lceil \lg n \rceil$ steps, we will have either found the value v or determined that it is not in the array. Thus, the algorithm terminates after at most $\Theta(\lg n)$ steps. □

Exercise Problem 2-3. 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 $y \leftarrow 0$
2 $i \leftarrow n$
3 **while** $i \geq 0$
4 **do** $y \leftarrow a_i + x \cdot y$
5 $i \leftarrow i - 1$

1. What is the asymptotic running time of this code fragment for Horner's rule?

2. 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?

3. Prove that the following is a loop invariant for the while loop in lines 3-5.

At the start of each iteration of the while loop of lines 3-5,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Interpret a summation with no terms as equaling 0. Your proof should follow the structure of the loop invariant proof presented in this chapter and should show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

4. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

Solution: 1. This algorithm does precisely two additions and one multiplication per iteration of the `while` loop. Since the `while` loop terminates after n iterations, it follows that Horner's rule computes $P(x)$ using $2n$ additions and n multiplications, hence $\Theta(n)$ time.

2. The following pseudocode implements the naïve polynomial evaluation algorithm. The sequence a is assumed to contain the coefficients of $P(x)$.

```
def naive_poly (a, x):
    k = len (a)
    if k == 1:
        return a[0]
    j = 1
    for i in range (k-1):
        j *= x
    return a[k-1] * j + naive_poly (a[0:k-1], x)
```

For a polynomial of degree n, this algorithm always does precisely j multiplications to compute the $a_j x^j$ term, and precisely $n-1$ additions in total. Thus, the algorithm does $1+2+\cdots+n-1+n = n(n+1)/2$ multiplications and $n-1$ additions, implying this algorithm uses $\Theta(n^2)$ operations.

3. At the start of the first iteration, we have $i = n$ and $y = 0$, so

$$\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = \sum_{k=0}^{-1} a_{k+n+1} x^k = 0$$

because the sum is empty, so the loop invariant holds.

Suppose $i = j$ with $0 \le j < k$ at the beginning of an iteration. By our loop invariant, we have

$$y = \sum_{k=0}^{n-(j+1)} a_{k+j+1} x^k$$

at the beginning of the iteration. The algorithm then sets y to

$$a_j + x \cdot \sum_{k=0}^{n-(j+1)} a_{k+j+1} x^k = a_j x^0 + \sum_{k=0}^{n-(j+1)} a_{k+j+1} x^{k+1}.$$

Next, the algorithm decrements i, so we now have $j = i+1$. Thus, the previous expression becomes

$$\begin{aligned} a_{i+1}x^0 + \sum_{k=0}^{n-(i+1+1)} a_{k+(i+1)+1}x^{k+1} &= a_{i+1}x^0 + \sum_{k=0}^{n-(i+2)} a_{k+i+2}x^{k+1} \\ &= a_{i+1}x^0 + \sum_{m=1}^{n-(i+1)} a_{m+i+1}x^m \\ &= \sum_{m=0}^{n-(i+1)} a_{m+i+1}x^m, \end{aligned}$$

so the invariant is maintained.

Finally, at termination, we have $i = -1$, so the value of y is

$$\sum_{k=0}^{n-(-1+1)} a_{k+-1+1}x^m = \sum_{k=0}^{n} a_k x^k,$$

4. Since the final value of y is $\sum_{k=0}^{n} a_k x^k$, we conclude that this algorithm correctly evaluates the polynomial $P(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$. □

Big O problems[1]

For each pair of functions $f(n), g(n)$, determine which of the following apply:

(a) $f(n) = \Theta(g(n))$,
(b) $f(n) = O(g(n))$ but $f(n) \neq \Theta(g(n))$, or
(c) $f(n) = \Omega(g(n))$ but $f(n) \neq \Theta g(n)$.

Exercise 1. $f(n) = 100n + \log n,\ g(n) = n + (\log n)^2$.

Solution: Since

$$\begin{aligned} \lim_{n\to\infty} \frac{100n + \log n}{n + (\log n)^2} &= \lim_{n\to\infty} \frac{100 + 1/n}{1 + (2\log n)/n} \text{ (by L'Hôpital's rule)} \\ &= 100, \end{aligned}$$

by the theorem given in class (see, e.g., the text by Brainerd & Landweber), we have $f(n) = \Theta(g(n))$. □

[1]Not part of Homework Set 1; from the text by Udi Manber.

Exercise 2. $f(n) = \log(n),\ g(n) = \log(n^2)$.

Solution: Since

$$g(n) = 2\log(n) = 2f(n),$$

we have $f(n) = \Theta(g(n))$. □

Exercise 3. $f(n) = n^2/(\log n),\ g(n) = n(\log n)^2$.

Solution: Since

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^2/(\log n)}{n(\log n)^2} &= \lim_{n\to\infty} \frac{n}{(\log n)^3} \\
&= \lim_{n\to\infty} \frac{1}{3(\log n)^2/n} \text{ (by L'Hôpital's rule)} \\
&= \lim_{n\to\infty} \frac{n}{3(\log n)^2} \\
&= \lim_{n\to\infty} \frac{1}{6(\log n)/n} \\
&= \lim_{n\to\infty} \frac{n}{6\log n} \\
&= \lim_{n\to\infty} \frac{1}{6/n} \text{ (by L'Hôpital's rule yet again)} \\
&= \lim_{n\to\infty} \frac{n}{6} = \infty,
\end{aligned}$$

the theorem from class tells us that $f(n) = \Omega(g(n))$ but $f(n) \neq \Theta(g(n))$. □

Exercise 4. $f(n) = (\log n)^{\log n},\ g(n) = n/(\log n)$.

Solution: Let $n = e^u$ for some u. We then have $f(n) = u^u$ and $g(n) = (\log u)/u$. Then,

$$\frac{f(n)}{g(n)} = \frac{u^u}{(\log u)/u} = \frac{u^{u-1}}{\log u}.$$

Since u^{u-1} increases faster than u, and

$$\begin{aligned}\lim_{u\to\infty}\frac{u}{\log u} &= \lim_{u\to\infty}\frac{1}{1/u}\\ &= \lim_{u\to\infty} u\\ &= \infty,\end{aligned}$$

we know $\lim_{u\to\infty} f(n)/g(n) = \lim_{n\to\infty} f(n)/g(n) = \infty$. Hence, our theorem tells us that $f(n) = \Omega(g(n))$ but $f(n) \neq \Theta(g(n))$. □

Exercise 5. $f(n) = n^{1/2},\ g(n) = (\log n)^5$.

Solution: Substituting $n = e^{2u}$, we get that

$$\frac{f(n)}{g(n)} = \frac{e^u}{32u^5}.$$

Applying L'Hôpital's rule five times sequentially, we finally get

$$\lim_{u\to\infty}\frac{e^u}{32u^5} = \lim_{u\to\infty}\frac{e^u}{32\cdot 5!} = \infty = \lim_{n\to\infty}\frac{f(n)}{g(n)},$$

so our theorem tells us that $f(n) = \Omega(g(n))$ but $f(n) \neq \Theta(g(n))$. □

Exercise 6. $f(n) = n2^n,\ g(n) = 3^n$.

Solution: Since $f(n)/g(n) = n(2/3)^n$, substituting $n = \log_{2/3} u$, we get

$$\frac{f(n)}{g(n)} = u\log_{2/3} u.$$

We thus conclude that

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{u\to\infty} u\log_{2/3} u = \infty,$$

therefore, we have $f(n) = \Omega(g(n))$ but $f(n) \neq \Theta(g(n))$. □