Problem Set 1 Solutions

Harm Derksen

Problem 1. (See Exercises 2-1-3, 2-2-3 in the book)

Consider the searching problem:
Input: A sequence of $n$ numbers $A = (a_1, a_2, \ldots, a_n)$ and a value $v$.
Output: An index $i$ such that $v = A[i]$ or the special value NIL if $v$ does not appear in $A$.

(a) Write pseudocode for $\text{linear-search}(A, v)$ which scans through the sequence, looking for $v$.

(b) Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

(c) Analyse your algorithm. How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in $\Theta$ notation? Justify your answers.

Solution:

(a) After the algorithm below terminates, $i$ will be the position of $v$ in the array $A$ if $v$ appears in the array $A$ and otherwise $i$ will be equal to NIL.

$\text{linear-search}(A, v)$
1. $i \leftarrow 1$
2. while $i \leq \text{length}[A]$ and $v \neq A[i]$
3. do $i \leftarrow i + 1$
4. if $i > \text{length}[A]$
5. then $i \leftarrow \text{NIL}$

(b) The loop in lines 2–3 has the following invariant:

At the beginning of each iteration of the loop, we have $A[j] \neq v$ for all integers $j$ with $1 \leq j \leq i - 1$.

Initialization: At the beginning of the first iteration, we have $i = 1$ and the loop invariant is the empty statement because there does not exist an integer $j$ with $1 \leq j \leq 0$. In particular, the loop invariant is true in the first iteration.

Maintenance: Suppose that in a certain iteration we have that $A[j] \neq v$
for $1 \leq j \leq i - 1$. If $A[i] \neq v$ or $i > \text{length}[A]$ then the loop breaks and there will be no more iterations. Otherwise we get that $A[j] \neq v$ for $1 \leq j \leq i$. In the loop we increase $i$ by 1. This means that we still have that $A[j] \neq v$ for $1 \leq j \leq i - 1$ in the next iteration.

**termination:** Clearly if the loop breaks where $i$ has a value $\leq \text{length}[A]$, then $v = A[i]$. The algorithm is correct in this situation. If the loop breaks where $i$ has a value $> \text{length}[A]$ then the algorithm will terminate where $i$ has the value equal to NIL. In this case we have to prove that there does not exist any $j$ with $1 \leq j \leq \text{length}[A]$ such that $A[j] = v$. To prove this, we use the loop invariant. At the beginning of the loop, just before the loop breaks, we have $i = \text{length}[A] + 1$. The loop invariant tells us that $A[j] \neq v$ for $j = 1, \ldots, \text{length}[A]$ which is what we wanted to show.

(c) Let $j$ be the smallest integer such that $A[j] = v$ if it exists. Otherwise we put $j = n + 1$ where $n = \text{length}[A]$. Here is an analysis what each step in the algorithm costs, timewise:

1. $c_1$ (executed only once)
2. $c_2$ (executed $j$ times)
3. $c_3$ (executed $j - 1$ times)
4. $c_4$ (executed once)
5. $c_5$ (executed at most once)

The total time is $c_1 + c_2j + c_3(j - 1) + c_4$ and $c_1 + c_2n + c_3(n - 1) + c_4 + c_5$ if $j = n + 1$.

The worst case scenario is when $v$ does not appear in the array at all. In this case, the running time is $\Theta(n)$.

The best case is when $v = A[1]$. In that case, the loop breaks of immediately. The running time is $\Theta(1)$.

The average value of $j$ is $((n + 1) + 1)/2 = n/2 + 1$. The running time will be still $\Theta(n)$.

**Problem 2.** (see Exercise 2-2-2 in book)

(a) Write pseudocode for this algorithm which is known as selection sort.
(b) 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?
(c) Give the best-case and worst-case running times of selection sort in $\Theta$-notation.
Solution:

(a) selection-sort($A$)
1. for $i \leftarrow 1$ to length[$A$] − 1
2. do $j \leftarrow i$
3. for $k \leftarrow i + 1$ to length[$A$]
5. then $j \leftarrow k$
6. dummy $\leftarrow A[j]$
8. $A[i] \leftarrow$ dummy

Explanation: In lines 3–5, the algorithm searches for the smallest element in $A[i], A[i+1], \ldots, A[n]$ (where $n = \text{length}[A]$). The variable $j$ will have the value of the index for which $A[j]$ is smallest after lines 3–5 are executed. (If one likes to be very precise, one can also write down a loop invariant for the inner loop to prove these claims.) In lines 6–8, the values of $A[i]$ and $A[j]$ are interchanges using a dummy variable $dummy$.

(b) An invariant for the outer loop in lines 1–8 is the following statement:

At the beginning of each iteration of the loop, we have $A[j] \leq A[k]$ for all $j$ and $k$ with $1 \leq j < i$ and $j < k \leq n$ where $n = \text{length}[A]$ and the entries in $A[1 \ldots n]$ are equal to the original entries in $A[1 \ldots n]$.

Initialization: At the first iteration, when $i = 1$, the loop invariant is an empty statement because there is no integer $j$ with $1 \leq j < i$. The loop invariant is therefore true at the beginning of first iteration of the loop.

Maintenance: Suppose that the loop invariant is true at the beginning of the iteration of the loop. This means that $A[j] \leq A[k]$ for all integers $j$ and $k$ with $1 \leq j < i$ and $j < k \leq n$. In the loop, lines 2–8, $A[i]$ is exchanged with $A[j]$ where $j$ is an index with $i \leq j \leq n$ for which $A[j]$ is minimal. In particular, after executing lines 2–8, we will have that $A[j] \leq A[k]$ for all $1 \leq j \leq i$ and $j < k \leq n$. At the end of the loop, $i$ is being increased by 1. This shows that $A[j] \leq A[k]$ for all $j$ with $1 \leq j < i$ and $j < k \leq n$ at the beginning of the next iteration of the loop.

Termination: The loop terminates when $i$ is equal to $n$. The loop invariant tells us that $A[j] \leq A[k]$ for all integers $j, k$ with $1 \leq j < n$ and $j < k \leq n$. This means that $A[1 \ldots n]$ are sorted. Also, the loop invariant tells us that the entries $A[1 \ldots n]$ are equal to the original entries in $A[1 \ldots n]$ (but in different order). This shows the correctness of the algorithm.

(c) Here is an analysis:
1. $c_1$ (executed $n$ times)
2. $c_2$ (executed $n - 1$ times)
3. \( c_3 \) (executed \( n + (n - 1) + \cdots + 2 = \frac{1}{2}n^2 + \frac{1}{2}n - 1 \) times)
4. \( c_4 \) (executed \( (n - 1) + (n - 2) + \cdots + 1 = \frac{n}{2}n^2 - \frac{1}{2}n \) times)
5. \( c_5 \) (executed at most \( \frac{1}{2}n^2 - \frac{1}{2}n \) times)
6. \( c_6 \) (executed \( n - 1 \) times)
7. \( c_7 \) (executed \( n - 1 \) times)
8. \( c_8 \) (executed \( n - 1 \) times)

The running times in any (best, worst or average) case is \( \Theta(n^2) \).

**Problem 3.** *With additional assumptions on the array \( A \) one can sort faster.* Suppose that \( A \) is an array of length \( n \) such that the entries of \( A[1..n] \) all lie in the set \( \{1, 2, \ldots, 2n\} \). Design an algorithm with running time \( \Theta(n) \) that sorts the array \( A \). Prove and analyse your algorithm.

**Solution.** We simply count how often each integer in \( \{1, 2, \ldots, 2n\} \). We create an array \( C[1\ldots2n] \) and initialize \( C[1] = C[2] = \cdots = C[2n] = 0 \). We go through the array \( A \) and increase \( C[j] \) if \( A[i] = j \). Then we can easily rearrange \( A \): 1 appears \( C[1] \) times, 2 appears \( C[2] \) times, etc. Here is the pseudocode:

```plaintext
sort(A)
1. \( n = \text{length}[A] \)
2. create an array \( C[1\ldots2n] \)
3. for \( i \leftarrow 1 \) to \( 2n \)
   4. \( C[i] \leftarrow 0 \)
5. for \( j \leftarrow 1 \) to \( n \)
6. do \( C[A[j]] \leftarrow C[A[j]] + 1 \)
7. \( j \leftarrow 1 \)
8. for \( i \leftarrow 1 \) to \( 2n \)
9. do while \( C[i] > 0 \)
10. do \( A[j] \leftarrow i \)
11. \( C[i] \leftarrow C[i] - 1 \)
12. \( j \leftarrow j + 1 \)
```

After lines 1–6 are executed, \( C[i] \) will be equal to the number of times \( i \) appears in the array \( A \) for \( i = 1, 2, \ldots, 2n \). In lines 7–12, \( i \) is copied exactly \( C[i] \) times into the array for \( i = 1, 2, \ldots, 2n \).

The running times of lines 1–7 is \( \Theta(n) \). Line 8 is executed \( 2n + 1 = \Theta(n) \) times. The while statement in line 9 is executed \( C[i] + 1 \) times and lines 10–12 are executed \( C[i] \) times for \( i = 1, 2, \ldots, 2n \). Note that \( \sum_{i=1}^{2n} C[i] = n \) and \( \sum_{i=1}^{2n} (C[i] + 1) = 3n \). This shows that the running time of lines 9–12 is also \( \Theta(n) \). The running time of this sorting algorithm is therefore \( \Theta(n) \).