German University in Cairo
Department of Computer Science
Dr. Nourhan Ehab

CSEN 703 Analysis and Design of Algorithms, Winter Term 2022
Practice Assignment 1

Exercise 1-1 From CLRS (©MIT Press 2001)

Illustrate the operation of Insertion Sort on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution:

At j = 2

31	41	59	26	41	58

At j = 3

31	41	59	26	41	58

At j = 4

31	41	59	26	41	58

At j = 5

26	31	41	59	41	58

At j = 6

26	31	41	41	59	58

Finally

26	31	41	41	58	59

Exercise 1-2

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 on A.

i. List the five inversions of the array $(2, 3, 8, 6, 1)$. Note that inversions are specified by indices rather than by the values in the array.

   Solution:

   Assuming one-based indexing the 5 inversions are: $(1, 5), (2, 5), (3, 5), (4, 5)$ and $(3, 4)$.

ii. Identify the array containing all the elements from the set $\{1, 2, ..., n\}$ which has the most inversions. How many does it have?

   Solution:

   An array that is reversely sorted has the most inversions. A reversely sorted array $(n, n-1, n-2, ..., 1)$ has $\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$ inversions.

iii. Is there a relationship between the operations performed in insertion sort given an input array, and the number of

   Solution:

   Yes, there is a direct relation between the operations of insertion sort and the number of inversions. The inner while loop is entered only when $A[i] > key$ where $key = A[j]$. Since $A[i] > A[j]$ and $i < j$, then (i, j) is one of the inversions in array A. The operations in the inner while loop inserts the key $A[j]$ in its correct position in the subarray $A[1..j]$. Hence, each execution of the inner while loop corresponds to the elimination of one inversion.

Exercise 1-3 From CLRS (©MIT Press 2001)

Consider the **searching problem**:
Input: A sequence of n numbers $A[a_1, a_2, ..., a_n]$ 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.

i. Write pseudocode for linear search which scans through the sequence looking for v.

Solution:

```
1: function LinearSearch(A,v)
2:     for i ← 1, A.length do
3:         if A[i] == v then
4:             return i
5:         end if
6:     end for
7:     return NIL
8: end function
```

ii. Analyze the best and worst-time complexity for linear search.

Solution:

In the best case, v is the first element in A. Therefore, only 1 iteration of the for loop is executed, and the if condition is checked only once before 1 is returned.

line	*cost*	*times*
2	c_1	1
3	c_2	1
4	c_3	1
5	0	0
6	0	0
7	c_4	0

$T(n)_{bestcase} = c_1(1) + c_2(1) + c_3(1)$

In the worst case, v is not in the list. In this case, we will have to traverse the whole list looking for v.

line	*cost*	*times*
2	c_1	$n+1$
3	c_2	n
4	c_3	0
5	0	0
6	0	0
7	c_4	1

$T(n)_{worstcase} = c_1(n + 1) + c_2(n) + c_4(1) = (c_1 + c_2)n + c_1 + c_4$

iii. Using a loop invariant, prove that your algorithm is correct.

Solution:

Loop Invariant: At the start of each iteration of the for loop we have $A[j] \neq v$ for all $j < i$.
Initialization: Before the first loop iteration we have $i = 1$ and the sub-array A[1..i-1] is empty. Accordingly, the loop invariant trivially holds.
Maintenance: The loop invariant is maintained after each iteration. Otherwise, if the invariant did not hold, this would mean that there is some $j < i$ such that $A[j] = v$. In this case, j would have been returned at the $j - th$ iteration terminating the loop and there would be no $i - th$ iteration. Hence, a contradiction.
Termination: There are two cases that can cause the loop to terminate: either v is found in the array and the loop terminates after i iterations where $i \leq A.length$, or v is never found in the array and NIL is returned. In the first case, the if condition check ensures that $A[i] = v$. Since we reached the i-th iteration, then for all $j < i$ $A[j] \neq v$, and the loop invariant is maintained. In the later case, since NIL is returned, then for all $j \leq A.length$ $A[j] \neq v$ and the loop invariant is maintained as well.

Exercise 1-4

Write an algorithm to find the index of the largest and smallest value in an array of integers.

Solution:

```
 1: function MaxMin(A)
 2:     min ← 1
 3:     max ← 1
 4:     for i=2, A.length do
 5:         if A[i]< A[min] then
 6:             min ← i
 7:         end if
 8:         if A[i]> A[max] then
 9:             max ← i
10:         end if
11:     end for
12: end function
```

i. What is the best-case and worst-case running times of your algorithm?

Solution:

In the best case, $min = max = 1$. Therefore, lines 6 and 9 will never be executed.

line	*cost*	*times*
2	c_1	1
3	c_2	1
4	c_3	n
5	c_4	$n-1$
6	c_5	0
8	c_6	$n-1$
9	c_7	0

$$\begin{aligned} T(n)_{bestcase} &= c_1 + c_2 + c_3(n) + c_4(n-1) + c_6(n-1) \\ &= (c_3 + c_4 + c_6)n + (c_1 + c_2 - c_4 - c_6) \end{aligned}$$

In the worst case, the array is ascendingly or descendingly sorted. In the former case, line 9 will be executed $n - 1$ times and:

$$T(n)_{worstcase} = c_1 + c_2 + c_3(n) + c_4(n-1) + c_6(n-1) + c_7(n-1)$$

In the latter case, line 6 will be executed $n - 1$ times and:

$$T(n)_{worstcase} = c_1 + c_2 + c_3(n) + c_4(n-1) + c_5(n-1) + c_6(n-1)$$

ii. Define the loop invariant for your algorithm and show that it holds.

Solution:

Loop Invariant: At any index i, min is the index of the minimum element in $A[1..i-1]$ and max is the index of the maximum element in $A[1..i-1]$.
Initialization: Before the first iteration, we have $i = 2$ and $min = max = 1$. Therefore, min and max are the indices of the minimum and maximum elements respectively in the subarray A[1].
Maintenance: At each iterative step, we record in min and max the current index of the largest and smallest element within the list $A[1..i]$.
Termination: The for loop terminates when $i = A.length + 1$ with the min and max holding the values of the minumum and maximum elements in $A[1..A.length]$.

Exercise 1-5

The following code snippet computes the sum of the first n numbers in the array a.

1: sum $\leftarrow 0$
2: **for** $i = 1; i \leq A.length; i++$ **do**
3:     sum $\leftarrow$ sum $+ A[i]$
4: **end for**

i. What is the best-case and worst-case running time of the above algorithm?

Solution:

In both the best and worst cases, the for loop body will be executed n times in order to calculate the sum of all the elements in the array.

line	*cost*	*times*
1	c_1	1
2	c_2	$n+1$
3	c_3	n

$T(n) = c_1 + c_2(n+1) + c_3(n) = (c_2 + c_3)n + (c_1 + c_2)$

ii. What is the loop invariant?

Solution:

Before each iteration, sum holds the value of the summation of all the elements in the subarray $a[1..i-1]$.

iii. Prove your invariant by induction.

Solution:

Initialization: Before the first iteration, $sum = 0$ which is the summation of all elements in the empty subarray $A[1..i-1]$.
Maintenance: At each iteration of the for loop, we add to sum the value of $A[i]$. Therefore, before the next iteration sum contains the summation of all the elements in $A[1..i-1]$.
Termination: After the loop terminates, $i = n+1$. Therefore, sum contains the summation of the elements in $A[1..A.length]$.

Exercise 1-6

Consider sorting n numbers in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then, finding the second smallest element of A, and exchanging it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudo code for this algorithm, which is known as **selection sort**.

Solution:

```
1: function SelectionSort(A)
2:     n ← A.length
3:     for i ← 1, n − 1 do
4:         minIndex ← i
5:         for j ← i + 1, n do
6:             if A[j] < A[minIndex] then
7:                 minIndex ← j
8:             end if
9:         end for
10:        Exchange A[i] ↔ A[minIndex]
11:    end for
12: end function
```

i. Give the best-case and worst-case running times of selection sort.

Solution:

line	*cost*	*times*
2	c_1	1
3	c_2	$(n-1)+1=n$
4	c_3	$n-1$
5	c_4	$\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n+1}(1))$
6	c_5	$\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n}(1))$
7	c_6	$\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n}(1))$
8	0	$\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n}(1))$
9	0	$\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n}(1))$
10	c_7	$n-1$
11	0	$n-1$
12	0	$n-1$

Expanding the summations that we have,

$$\begin{aligned}
\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n+1}(1)) &= \sum_{i=1}^{n-1}[(n+1)-(i+1)+1] \\
&= \sum_{i=1}^{n-1}(n-i+1) \\
&= \sum_{i=1}^{n-1} n+1-\sum_{i=1}^{n-1} i \\
&= (n+1)(n-1-1+1)-\left[\frac{(n-1)(n)}{2}\right] \\
&= n^2-1-\frac{n^2}{2}+\frac{n}{2} \\
&= \frac{n^2}{2}+\frac{n}{2}-1
\end{aligned}$$

$$
\begin{aligned}
\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n}(1)\right) &= \sum_{i=1}^{n-1}[n-(i+1)+1] \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= (n^2-n) - \left[\frac{(n-1)(n)}{2}\right] \\
&= n^2 - n - \frac{n^2}{2} + \frac{n}{2} \\
&= \frac{n^2}{2} - \frac{n}{2}
\end{aligned}
$$

In the best case, the cost of lines 2,3,4,5,6 and 10 are calculated giving us

$$
\begin{aligned}
T(n) &= c_1(1) + c_2(n) + c_3(n-1) + c_4(\tfrac{n^2}{2} + \tfrac{n}{2} - 1) + c_5(\tfrac{n^2}{2} - \tfrac{n}{2}) + c_7(n-1) \\
&= (c_1 - c_3 - c_4 - c_7)1 + (c_2 + c_3 + \tfrac{1}{2}c_4 - \tfrac{1}{2}c_5 + c_7)n + (\tfrac{1}{2}c_4 + \tfrac{1}{2}c_5)n^2
\end{aligned}
$$

In the worst case, the cost of lines 2,3,4,5,6,7 and 10 are calculated giving us :

$$
\begin{aligned}
T(n) &= c_1(1) + c_2(n) + c_3(n-1) + c_4(\tfrac{n^2}{2} + \tfrac{n}{2} - 1) + c_5(\tfrac{n^2}{2} - \tfrac{n}{2}) + c_6(\tfrac{n^2}{2} - \tfrac{n}{2}) \\
&\quad + c_7(n-1) \\
&= (c_1 - c_3 - c_4 - c_7)1 + (c_2 + c_3 + \tfrac{1}{2}c_4 - \tfrac{1}{2}c_5 - \tfrac{1}{2}c_6 + c_7)n \\
&\quad + (\tfrac{1}{2}c_4 + \tfrac{1}{2}c_5 + \tfrac{1}{2}c_6)n^2
\end{aligned}
$$

ii. Why does the algorithm need to run for only the first $n-1$ elements, rather than for all n elements?

Solution:

The algorithm maintains the loop invariant that at the start of each iteration of the outer for loop, the subarray $A[1..i]$ consists of the i smallest elements in the array $A[1..n]$, and this subarray is in sorted order. After the first $n-1$ iterations of the outer for loop, the subarray $A[1..n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

iii. Prove that selection sort is correct.

Solution:

Initialization: Initially, $i = 1$. Therefore, the sub-array $A[1]$ contains only 1 element and is trivially sorted.
Maintenance: As i grows, the sub-array $A[1 \cdots i]$ becomes sorted by scanning elements $A[i+1 \cdots n]$ for the i^{th} smallest element and exchanging it with $A[i]$. Therefore, the algorithm's loop invariant is maintained.
Termination: When $i = n$, the outer loop terminates. As a result of the maintenance step, the sub-array $A[1 \cdots n-1]$ is sorted. We also know that $A[n]$ is trivially sorted with respect to the former subarray. Hence, the array $A[1 \cdots n]$ is sorted and the algorithm works correctly.

Exercise 1-7

Consider the following pseudo code for the algorithm Gnome Sort:

```
1: function GnomeSort(Array A)
2:     i ← 1
3:     while i ≤ n do
4:         if (i == 1) or (A[i − 1] ≤ A[i]) then
5:             i + +
6:         else
7:             Exchange A[i] ↔ A[i − 1]
8:             i − −
9:         end if
10:    end while
11: end function
```

i. Provide an example for each of best and worst-case inputs.

Solution:

Best case sample input :

1	3	4	5	8	12

Worst case sample input:

12	8	5	4	3	1

ii. Analyze the best and worst-time complexity of gnome sort.

Solution:

In the best case, the array is already sorted.

line	*cost*	*times*
2	c_1	1
3	c_2	$n+1$
4	c_3	n
5	c_4	n
7	c_5	0
8	c_6	0

$T(n)_{bestcase} = c_1 + (n+1)c_2 + nc_3 + nc_4 = (c_2 + c_3 + c_4)n + (c_1 + c_2)$

In the worst case, the array is reversely sorted.

line	*cost*	*times*
2	c_1	1
3	c_2	$(\sum_{i=1}^{n} n)+1$
4	c_3	$\sum_{i=1}^{n} n$
5	c_4	$\sum_{i=1}^{n} i$
7	c_5	$\sum_{i=1}^{n} i-1$
8	c_6	$\sum_{i=1}^{n} i-1$

Expanding the summations that we have,

$$\sum_{i=1}^{n} n = n^2$$

$$\sum_{i=1}^{n} i = \frac{n^2 + n}{2}$$

$$\sum_{i=1}^{n} i - 1 = \frac{n^2 - n}{2}$$

Note that the challenging part was to calculate the number of times the while loop is executed in the worst case. It happens to be straightforward, however, once a systematic approach is considered. We observe that for each value of i such that $1 \leq i \leq n$, the value is decremented $i-1$ times for swapping (the else block), and incremented i times again to sort the subarray of size $i+1$ (the then block). The while loop is therefore executed $\sum_{i=1}^{n} n = \sum_{i=1}^{n} i + \sum_{i=1}^{n} i - 1$ times.

$$
\begin{aligned}
T(n)_{worstcase} &= c_1 + c_2(n^2+1) + c_3(n^2) + c_4(\frac{1}{2}n^2 + \frac{1}{2}n) + c_5(\frac{1}{2}n^2 - \frac{1}{2}n) + c_6(\frac{1}{2}n^2 - \frac{1}{2}n) \\
&= (c_2 + c_3 + \frac{1}{2}c_4 + \frac{1}{2}c_5 + \frac{1}{2}c_6)n^2 + (\frac{1}{2}c_4 - \frac{1}{2}c_5 - \frac{1}{2}c_6)n + (c_1 + c_2)
\end{aligned}
$$

iii. Prove that the algorithm is correct.

Solution:

Loop invariant: Before each iteration, the subarray $A[1..i]$ is sorted.
Initialization: Initially, we start with i=1. Therefore, the subarray $A[1]$ is trivially sorted.
Maintenance: In each loop iteration i, we have two cases:

1 If $A[i-1] \leq A[i]$ or $i = 1$, then i is incremented. It can be seen that the subarray $A[1..i]$ is sorted.
2 Otherwise, if $A[i-1] > A[i]$, a swap will be performed and $A[i-1..i]$ will be sorted. Afterwards, i will be decremented k times where $1 \leq k \leq i-1$ to continue the swapping process and reach the state where $A[1..i]$ is sorted.

Termination: The loop terminates when $i = n+1$. At this point, it is obvious that $A[1..n]$ became sorted in the maintenance step. Therefore, the algorithm is correct.