Data Structures and Algorithms
Lecture notes: Introduction, definitions, terminology, Brassard Chap. 2

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

29 septembre 2021

Outline

- Definitions of terms like *algorithm*, *input size*, *formal parameters*, *output*, *running time*, *time complexity*, *basic operations*, *rate of growth*, etc.
- Distinction between terms like *problem* and *problem's instance*, or *algorithm* and *program*.
- Introduction of the basic procedure and steps to analyze algorithms.
- Algorithms on YouTube, see
  https://www.youtube.com/watch?v=6hfOvs8pY1k,
  https://www.youtube.com/watch?v=e_WfC8HwVB8,
  https://www.youtube.com/watch?v=CvSOaYi89B4

What is an algorithm ?

An algorithm is a solution to a problem in a form of a sequence of steps precise and non-ambiguous enough that it can be coded into a programming language for execution on a computer.

Algorithms can be described using pseudocode, which is similar to procedural languages, but free of programming languages syntactic details.

The following is an example of pseudocode describing a sorting algorithm :

Selection sort(A[1..n])
for $i = 1$ **to** $n - 1$ **do**
    $minj = i$; $minx = A[i]$;
    for $j = i + 1$ **to** n **do**
        if $A[j] < minx$ **then**
            $minj = j$; $minx = A[j]$;
    A[$minj$] $= A[i]$; $A[i] = minx$;

Problem, Problem's instance & algorithms

A problem is a set of problem instances that share some common properties

The problem of sorting positive integers in nondecreasing order is defined in terms of its instances : "Given an array of positive integers find a nondecreasing sequence (permutation) of these numbers"

In the above example, each array of positive integers is an instance of this problem

An algorithm solves a problem, i.e. it works for all its instances, not just for a subset of them

Problems, Algorithms and Programs

We design algorithms and programs for a problem, not for a problem instance.

There are *often* several algorithms for a same problem (ex. sorting an array of integers : insertion, selection, merge sort, quicksort, bubblesort)

There is always many different programs for a same algorithm.

By the way, a computer program (that work) is always an algorithm but of course an algorithm is not a computer program.

Two different algorithms for a same problem

Here are two algorithms for solving the problem of sorting in increasing order an array of positive integers

```
Selection sort(A[1..n])
for i = 1 to n − 1 do
    minj = i ; minx = A[i] ;
    for j = i + 1 to n do
        if A[j] < minx then
            minj = j ; minx = A[j] ;
    A[minj] = A[i] ; A[i] = minx ;
```

```
InsertionSort(A[1..n])
for (i = 2; i <= n; i + +)
    v=A[i] ; j=i-1 ;
    while(j > 0 & A[j] > v)
            A[j+1]=A[j] ; j− ;
    A[j+1] = v ;
```

Many algorithms for a same problem

Given that there is often several algorithms for a same problem, one might want to compare the algorithms with each others

If you discover a new algorithm, you might want to compare it with existing algorithms

Then the question becomes :

- How should we compare two algorithms with each others ?
- What should we use as a measure of how "good" an algorithm is ?

There are different answers to these questions, we will study some of them

Comparing running (execution) times

The running time to solve instances is a relevant attribute for comparing algorithms with each other

Pretty much every one prefer algorithms that run fast, the faster the better

But we need to be careful when measuring the running time of an algorithm because factors not related to the algorithm can affect the running time.

Experimental Running Time

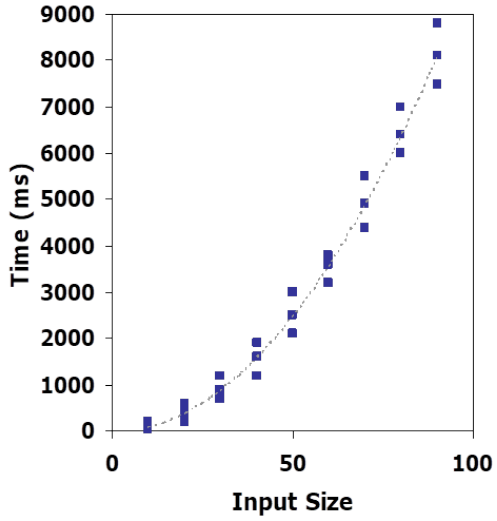
Write a program implementing the algorithm

Run the program with inputs of varying size and composition

Use a method like clock( ) to get an accurate measure of the actual running time

```
clock_t startTime = clock();
doSomeOperation();
clock_t endTime = clock();
clock_t clockTicksTaken = endTime - startTime;
double timeInSeconds = clockTicksTaken / (double) CLOCKS_PER_SEC;
```

Plot the results



Limitations of experimental running time measurements

Experimental evaluation of running time is very useful but

- It is necessary to code the algorithm, which may take some time
- Results may not be indicative of the running time on other inputs not included in the experiment
  - The particular instance of data the algorithm is operating on (e.g., amount of data [*input size*], type of data).
- In order to compare two algorithms, the same hardware and software environments must be used
  - Characteristics of the computer (e.g. processor speed, amount of memory, file-system type, number and type of elementary operations, size of the registers, etc.)
  - The way the algorithm is coded (the program ! ! !)

Running time is not “wall-clock” time

To avoid the issues describe in the previous slide we don't measure running time experimentally (wall-clock measurements)

Rather we are going to count the number of time a particular instructions is executed in the pseudo-code

Algorithms' running time

Uses a pseudo-code description of the algorithm instead of program code

Characterizes running time as the number of times an instruction is executed based of the input size (denoted as n)

Takes into account all possible inputs

Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

- Changing the hardware/software environment affects the running time by a constant factor, but does not alter the number of time a particular instruction is executed

Identifying the input size

The inputs of an algorithm are represented by parameters of the algorithm

Find the input(s) most relevant to measure the running time

Next determine the size of the input (*how much data*). This could be

- size of a file
- size of an array or matrix
- number of nodes in a tree or graph
- degree of a polynomial

We usually use n to denote the input size

Sometime more than one input is needed to analyze the running time of an algorithm

- for graphs one may need to know the number of nodes and number of edges in order to analyze the running time of the algorithm

Identifying the input size

The "selection sort" algorithm below has a single input parameter, the array A.

The size of this input is $n\times$ whatever the number of bits selected to represent an integer in the code

However we abstract the size of the input by n because

1. we don't care about how the algo is implemented
2. the number of times an operation is executed in the pseudo-code does not depend on how an integer is represented but rather on the number n of integers in the array

Selection sort(A[1..n])
for $i = 1$ **to** $n - 1$ **do**
  $minj = i$; $minx = A[i]$;
  for $j = i + 1$ **to** n **do**
    if $A[j] < minx$ **then**
      $minj = j$; $minx = A[j]$;
  A$[minj] = A[i]$; $A[i] = minx$;

Identifying the instruction to be counted

We count the instruction that is executed the most often

This instruction is selected among the most basic or elementary instructions of the pseudo-code, such as assignments, elementary arithmetic operations, or loop control operations

But the reasoning here is a bit complicated :

1. The different basic operations execute different number of machine instructions among themselves

2. The same basic operation may also require a different number of machine instructions when execute on a different computers

Abstracting the effect of different computer architectures

The different numbers of machine instructions (related to computer architectures) for a same pseudo-code instruction is abstracted by replacing this number by a variable

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

Here the expression describing the running time $T(n)$ is the same across computer architectures :

$T(n) =$

$c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$

Aggregating the terms in $T(n)$

Assuming $t_j = j$, the summation

$\sum_{j=2}^{n} t_j = 2 + 3 + 4 + \cdots + n = (\frac{n(n+1)}{2} - 1)$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) \\
&\quad + c_8(n-1) \\
&= c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) \\
&\quad + c_7(\frac{n(n-1)}{2}) + c_8(n-1) \\
&= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2})n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \\
&= an^2 + bn + c
\end{aligned}
$$

Rational for selecting one basic instruction

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

The basic instruction executed the most often in this code is 5, the control of the **while** loop which is executed $\frac{n^2+n}{2} - 1$ times for $\frac{c_5 n^2 + c_5 n}{2} - c_5$ machine operations

What is lost in terms of machine instructions when only considering instruction 5 ?

The relation between c_5 and a is linear, it is $c_5 y = a$, i.e. a is a multiplicative constant of c_5. Same for b and c.

Basic operations, a first example

The problem : Given an array $A[1..n]$ of n integers, sort the elements of the array in increasing order.

Selection sort(A[1..n])
for $i = 1$ **to** $n - 1$ **do**
    $minj = i$; $minx = A[i]$;
    for $j = i + 1$ **to** n **do**
        if $A[j] < minx$ **then**
            $minj = j$; $minx = A[j]$;
    A$[minj] = A[i]$; $A[i] = minx$;

What are the elementary operations ?

Which EO will be a good one to measure the running time of this algorithm ?

How many elementary operations are required by Selection sort ?

Selection sort (continue)

Selection sort(A[1..n])
for $i = 1$ **to** $n - 1$ **do**
    $minj = i$; $minx = A[i]$;
    for $j = i + 1$ **to** n **do**
        if $A[j] < minx$ **then**
            $minj = j$; $minx = A[j]$;
    A$[minj] = A[i]$; $A[i] = minx$;

Counting the number of basic operations : When $i = 1$, $n - 1$ basic operations are executed, when $i = 2$, ...

outer loop index	1	2	3	$\cdots$	n-2	n-1
# of basic ops	n-1	n-2	n-3	$\cdots$	2	1

so, the number of basic operations is
$1 + 2 + \cdots + n - 3 + n - 2 + n - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$

Basic operations, a second example

The problem : search for a value x in an array $A[1..n]$ of n integers. If x is found returns the entry i in A where x has been found, else returns 0.

Here is the pseudocode of an algorithm to solve this problem :

```
SequentialSearch (L[1..n], x)
Foundx = false; i = 1;
while (i ≤ n & Foundx == false) do
      if L[i] = x then
        Foundx = true;
      else
        i = i + 1;
if (Foundx == false) then i = 0;
return i;
```

- What is the size of the input ?
- What are the elementary operations ?
- Which EO will be a good one to measure the running time of this algorithm ?
- How many elementary operations, as a function of the array size n ?

Running time & worst case instances

For many algorithms, we cannot tell their running time just by counting the number of times an elementary operation is executed

Example, SequentialSearch on previous slide, for the following two instances with same size : $val = 4, A = [1, 2, 3, 4]$ and $val = 4, A = [4, 3, 2, 1]$, the running time is different :

- SequentialSearch([1,2,3,4]) is 4
- SequentialSearch([4,3,2,1]) is 1

though the input size of each instance is the same

To solve this issue we calculate the running time for the worst case instance of a given size

Worst case analysis

In a worst case analysis, we need to find the "worst instance", i.e. the instance(s) which have the largest running time for input size n

Beside the worst case analysis, there are also two other types of analysis :

- There is also best-case analysis where one seeks to find the instances with the lowest running time
- and average-case analysis, average-case will be described in later in another lecture

Worst case running time analysis

```
SequentialSearch (L[1..n], x)
Foundx = false; i = 1;
while (i ≤ n & Foundx == false) do
      if L[i] = x then
         Foundx = true;
      else
         i = i + 1;
if not Foundx then i = 0;
return i;
```

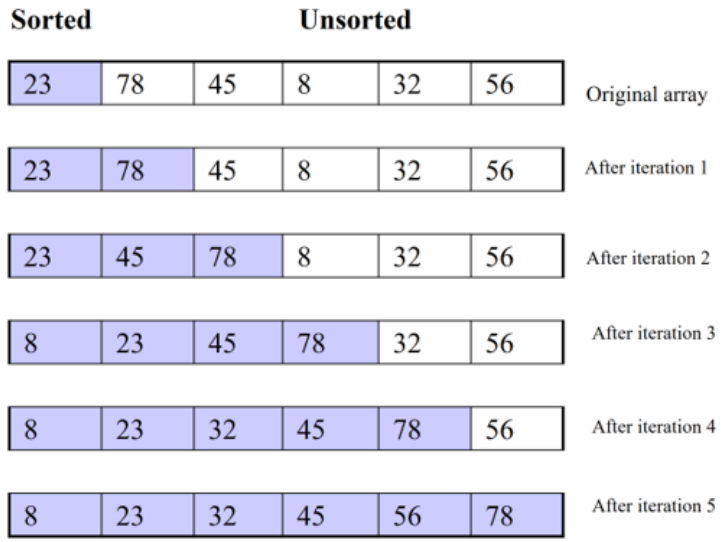
- What is (are) the worst-case instance(s) of a given size n
- How many elementary operations, as a function of the array size n, are required in worst case ?
- What is the best-case instance ?

Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0 \ \& \ A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 2$ one basic operation is executed

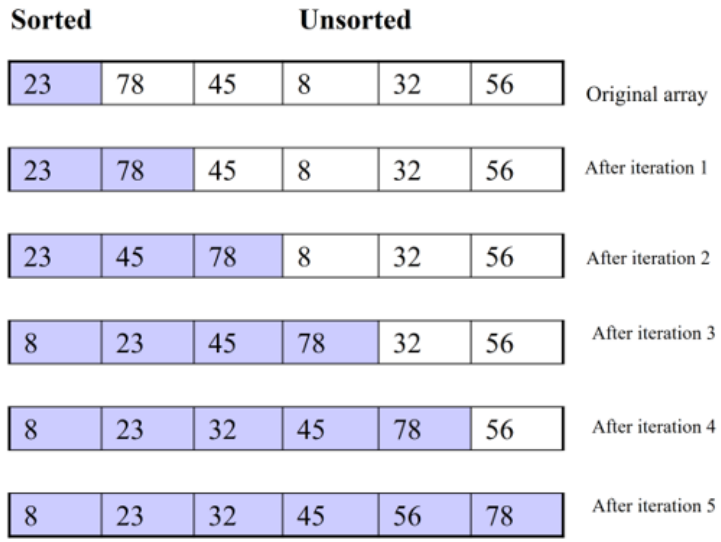


Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0 \ \& \ A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 3$, two basic operations are executed,therefore 1 + 2

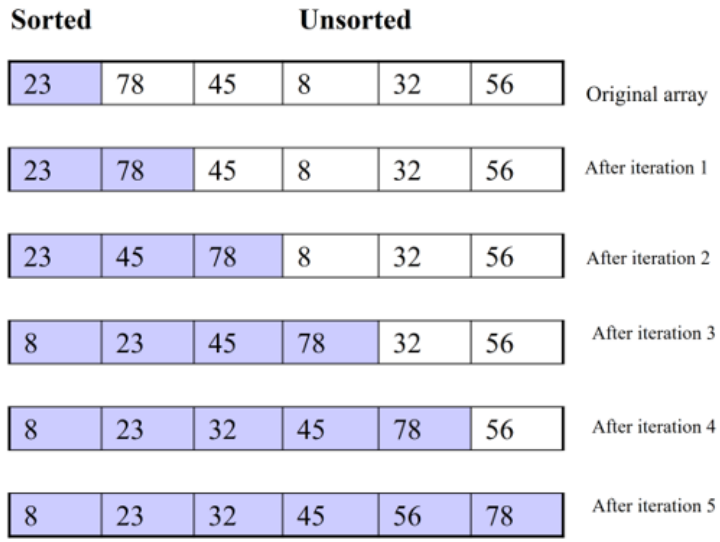


Insertion sort, worst case analysis

```
InsertionSort(A[1..n])
for (i = 2; i <= n; i++)
    v=A[i] ; j=i-1 ;
    while(j > 0 & A[j] > v)
            A[j+1]=A[j] ; j– ;
    A[j+1] = v ;
```

Running time

for $i = 4$, three basic operations are executed,therefore 1 + 2 +3

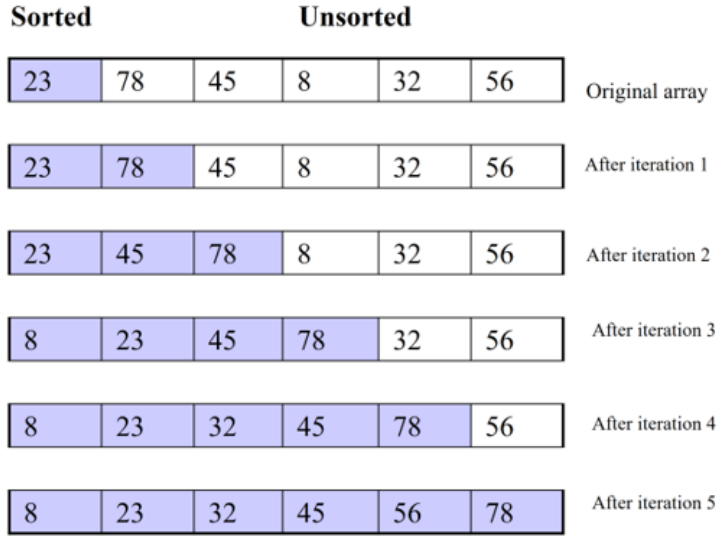


Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0 \ \& \ A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 5$, two basic operations are executed,therefore $1 + 2 + 3 + 2$

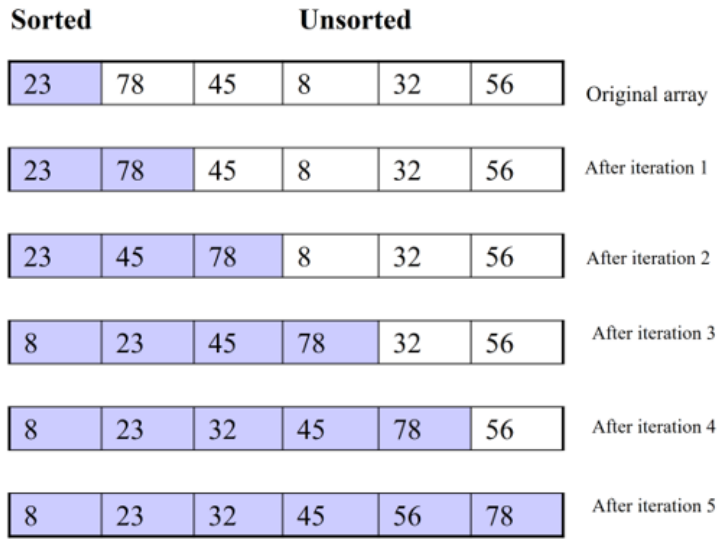


Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0 \ \& \ A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 6$, one basic operation is executed,therefore $1 + 2 + 3 + 2 + 1 = 9$ basic operations



Insertion sort, worst case analysis

The input array is already sorted (best case)

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78

Running time

In this case the running time is $n - 1$

Insertion sort, worst case analysis

The input array is sorted in decreasing order (worst case)

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

In this case the running time is $2 + 3 + 4 + 5 + 6 = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$

78	56	45	32	23	8
56	78	45	32	23	8
45	56	78	32	23	8
32	45	56	78	23	8
23	32	45	56	78	8
8	23	32	45	56	78

Selection sort, worst case analysis

Selection sort(A[1..n])
for $i = 1$ **to** $n - 1$ **do**
    $minj = i$; $minx = A[i]$;
    for $j = i + 1$ **to** n **do**
        if $A[j] < minx$ **then**
            $minj = j$; $minx = A[j]$;
    A$[minj] = A[i]$; $A[i] = minx$;

What is the worst case ?

The execution of the two loops is independent of how the input array is structured, therefore the running time for selection sort is the same for all inputs of a given size n

Example : Search for Maximum

Problem : Search the maximum element in an array of n integers.
Algorithm :

```
max(A[1..n])
max_value = int.MIN_VAL ;
for ( i = 0; i < n; i++)
    max_value = MAXIMUM(max_value, A[i]) ;
return max_value ;
```

Selecting (=) inside the loop as EO, the worst-case running time in terms of n are :

- for an array of length 1, 1 comparison
- for an array of length 2, 2 comparisons

  ⋮

- for an array of length n, n comparisons

Here worst-case and best-case are the same.

What's next

Actually, when comparing algorithms with each other, we are not interested in the exact number of basic operations executed, but rather we focus on the number of operations as the input size increase.

We are interested in the growth rate of the running time of algorithms, we will describe the techniques which classify algorithms according to their grow rate.

Exercises

1. As the example on slides 27 to 30, show the steps of insertion sort on the following array $A = [31, 41, 59, 26, 41, 58]$. For each iteration of the outer **for** loop, gives the number of iterations executed by the inner **while** loop.
2. How many elementary operations insertion sort executes on each of the two following input arrays : $[4, 3, 2, 1], [1, 2, 3, 4], [3, 1, 4, 2]$?
3. How would you modify the procedure insertion-sort such that it sorts in decreasing order rather than in increasing order as it is on slide 27 ?

Exercises (cont)

4. The following algorithm test whether an array of size n is sorted in increasing order. If so return true. Algorithm :

   TestForAscendingOrder($L[1..n]$)
   $i = 1$; ascending = true ;
   while ($i \leq n - 1$ & ascending) **do**
   ascending = $(L[i] \leq L[i + 1])$;
   $i = i + 1$;
   return ascending ;

   - Choose elementary operation(s) and decide what the size of the input is for the algorithm.
   - Give a worst-case input and compute how many elementary operations are performed for it.
   - Give a "best-case" input and compute how many elementary operations are performed for it.