1. Sometimes an algorithm doesn’t always need the entire input to produce the correct output. Give an example of such an algorithm.

Solution: For instance an algorithm for finding an integer with a certain property (for example: equal to 2, larger than 2, even, . . . ) in a sequence of integers. Or an algorithm that checks whether a graph contains a cycle. Note that both are existential questions.

2. Assume given an alphabetically ordered list consisting of 25000 names. We sequentially search for a certain name $N$. How many steps are needed in the best-case, worst-case, average-case?

Solution: Worst-case: 25000 steps because $N$ was the last one. Best-case: 1 step because $N$ happened to be the first one. Average-case: 12500 steps. (The latter is probably not completely correct because names are not homogeneously distributed over the alphabet.)


Solution: The sorted part of the sequence is written in red. Please note: this only gives an indication of what happens during the application of the algorithms. It is a step-by-step rendering but does not explain the way the elements are moved in the array. See also the book.

Insertion sort:

$[22, 15, 36, 20, 3, 9, 29]$  
$[15, 22, 36, 20, 3, 9, 29]$  
$[[15, 22, 36, 20, 3, 9, 29]$  
$[15, 20, 22, 36, 9, 29]$  
$[3, 15, 20, 22, 36, 9, 29]$  
$[3, 9, 15, 20, 22, 36, 29]$  
$[3, 9, 15, 20, 22, 29, 36]$

4. Give a worst-case input array of length 5 for insertion sort.

Solution: A worst-case input-sequence for insertion sort: a decreasing sequence, for instance $[5, 4, 3, 2, 1]$. A best-case input-sequence for insertion sort is a sorted sequence.

5. Adapt the pseudo-code for insertion sort to sort in non-increasing (instead of non-decreasing) order, and show correctness of your algorithm.

Solution:
Pseudo-code for reverse-order insertion sort:

Algorithm insertionSortReverseOrder(A, n):
    for \( j := 2 \) to \( n \) do
        key := A[j]
        i := j - 1
        while \( i \geq 1 \) and \( A[i] \leq key \) do
            A[i + 1] := A[i]
            i := i - 1
        A[i + 1] := key

Now we show the correctness of insertionSortReverseOrder using an invariant. Recall that an invariant is a property that remains true under execution of a part of the algorithm. We use the following property \( I \):

After \( k \) iterations of the for-loop, the sub-array \( A[1 \ldots k + 1] \) is a decreasingly sorted permutation of the sub-array \( A[1 \ldots k + 1] \) of the input-array.

We show that the property is an invariant using induction.

Base case: \( k = 0 \). Without executing the for-loop, indeed the sub-array \( A[1] \) is a sorted permutation of that part of the input-array.

Induction step: Assume that the property holds for \( k \). Then before executing the \( k + 1 \)st iteration of the for-loop, \( A[1 \ldots k] \) is a sorted permutation of that part of the input-array. Executing the \( k + 1 \)st iteration of the loop we take the element at position \( k + 1 \) as key. The elements before are shifted one place to the right if they are smaller than the key. In this shifting they remain ordered decreasingly. Then the key is inserted at the first position (coming from the right) where the element left of it is larger. The part right of the insertion point remained ordered. Also the part left of the insertion point remained ordered. In addition, the key is inserted correctly because left of it the element is larger and right of it the element is smaller. Hence \( A[1 \ldots k + 1] \) is sorted (in decreasing order). It is a permutation of that part of the input array because all replacements concern positions \( 1, \ldots, k + 1 \).

We conclude that \( I \) is in an invariant.

If \( I \) holds after exiting the loop, then \( k = n - 1 \). Hence \( I \) implies that \( A[1 \ldots n] \) is a sorted permutation of the input.

6. Give pseudo-code for the algorithm of Euclid for computing the greatest common divisor.

Solution:
Algorithm Euclid\((a, b)\):
\[\text{Input: naturals } a \text{ and } b\]
\[\text{Output: greatest common divisor of } a \text{ and } b\]
\[
\begin{align*}
\text{if } a &= 0 \\
    \quad \text{return } b \\
\text{if } b &= 0 \\
    \quad \text{return } a \\
\text{if } a < b \\
    \quad \text{Euclid}(a, b \mod a) \\
\text{else} \\
    \quad \text{Euclid}(a \mod b, b)
\end{align*}
\]

7. Give the time complexity of the following loops in terms of \( \Theta \).

Algorithm Loop1\((n)\):
\[
\begin{align*}
s &:= 0 \\
\text{for } i &:= 1 \text{ to } n \text{ do} \\
    s &:= s + i
\end{align*}
\]

Algorithm Loop2\((n)\):
\[
\begin{align*}
p &:= 1 \\
\text{for } i &:= 1 \text{ to } 2n \text{ do} \\
    p &:= p \cdot i
\end{align*}
\]

Solution:
The algorithm Loop1 is in \( \Theta(n) \). We do once the assignment \( s := 0 \). We do \( n + 1 \) times the test for the for-loop. We perform \( n \) times the calculation and assignment in the for-loop. Assuming that those steps take constant time, we find the following function for the running time:
\[T(n) = (c_1 + c_3) \cdot n + (c_1 + c_2)\]. This is for the worst-case running time (and also for the best-case). Hence Loop1 has worst-case running time in \( \Theta(n) \).

The algorithm Loop2 is in \( \Theta(n) \). (The first intuition is possibly \( \Theta(2n) \); it is then better to use \( \Theta(n) \).) We do once the assignment \( p := 1 \). We do \( 2n + 1 \) times the test for the for-loop. We do \( 2n \) times the calculation and
assignment in the for-loop. Assuming that those steps take constant time, we find the following function for the running time: \( T(n) = (2 \cdot c_2 + 2 \cdot c_3) \cdot n + (c_1 + c_3) \). Hence Loop2 has worst-case running time in \( \Theta(n) \).

8. Consider the following algorithm, with \( A[1 \ldots n] \) an array with \( n \) integer values:

\[
\text{Algorithm} \ maxSubarray(A, n):
\]

\[
\text{max} := 0
\]

\[
\text{for} \ left := 1 \ \text{to} \ n \ \text{do}
\]

\[
\text{sum} := 0
\]

\[
\text{for} \ right := left \ \text{to} \ n \ \text{do}
\]

\[
\text{sum} := \text{sum} + A[right]
\]

\[
\text{if} \ \text{sum} > \text{max} \ \text{then}
\]

\[
\text{max} := \text{sum}
\]

\[
\text{done}
\]

\[
\text{done}
\]

\[
\text{return} \ max
\]

(a) What does this algorithm compute?
Solution: The algorithm computes a maximum summation of the values of a sub-array of the input-array.

(b) What is the worst-case time-complexity in terms of \( \Theta \)?
Solution: The worst case is if the test \( \text{sum} > \text{max} \) succeeds every time.

The test for the outer for-loop is executed \( n+1 \) times. The statements in the body of the outer for-loop are executed \( n \) times.

For a fixed \( left \), the test for the inner for-loop is executed \( n - left + 2 \) times, and the statements in the body of the inner for-loop are executed \( n - left + 1 \) times.

For the lines 2, 3, 4, 11 of the algorithm, we find the following parts of the function for computing the running time:

\[
\begin{align*}
&c_1 \\
&(n + 1) \cdot c_2 \\
&n \cdot c_3 \\
&c_8
\end{align*}
\]

For a fixed \( left \), we find the following parts for lines 5, 6, 7, 8, where
we take the worst-case scenario for lines 7, 8:

\[ (n - left + 2) \cdot c_4 \]
\[ (n - left + 1) \cdot c_5 \]
\[ (n - left + 1) \cdot c_6 \]
\[ (n - left + 1) \cdot c_7 \]

We note that

\[ \sum_{left=1}^{n} (n - left + 1) = \sum_{j=1}^{n} j = \frac{n \cdot (n + 1)}{2} \]

and

\[ \sum_{left=1}^{n} (n - left + 2) = \sum_{j=2}^{n+1} j = \frac{(n + 1)(n + 2)}{2} - 1 \]

Hence we find

\[ T(n) = \]
\[ \left( \frac{1}{2} c_4 + \frac{1}{2} c_5 + \frac{1}{2} c_6 + \frac{1}{2} c_7 \right) \cdot n^2 + \]
\[ (c_2 + c_3 + \frac{3}{2} c_4 + \frac{1}{2} c_5 + \frac{1}{2} c_6 + \frac{1}{2} c_7) \cdot n + \]
\[ c_1 + c_2 + c_8 \]

Hence the worst-case (and also the best-case) time complexity of the algorithm is in \( \Theta(n^2) \).

9. Consider the following definition of the power function, for \( n \geq 0 \):

\[ p(x, n) = \begin{cases} 
1 & \text{if } n = 0 \\
x \cdot p(x, n - 1) & \text{if } n > 0 
\end{cases} \]

Give a pseudocode description of an algorithm \( \text{Power}(x, n) \) to compute the power function according to this definition. As an example, calculate \( \text{Power}(x, 5) \) with a sequence of equations. Argue (informally) that the number of recursive calls is in \( \Theta(n) \).

Solution:

Pseudo-code:

\textbf{Algorithm} \text{Power}(x, n):
\begin{itemize}
  \item \textbf{Input}: positive integers \( x \) and \( n \)
  \item \textbf{Output}: the value \( x^n \)
  \begin{itemize}
    \item if \( n = 0 \) then
      \begin{itemize}
        \item return 1
      \end{itemize}
    \item if \( n > 0 \) then
      \begin{itemize}
        \item \( y := \text{Power}(x, n - 1) \)
        \item return \( x \cdot y \)
      \end{itemize}
  \end{itemize}
\end{itemize}
Example for $n = 5$ and some $x$:

\[
\begin{align*}
\text{Power}(x, 5) &= \\
x \cdot \text{Power}(x, 4) &= \\
x \cdot (x \cdot \text{Power}(x, 3)) &= \\
x \cdot (x \cdot (x \cdot \text{Power}(x, 2))) &= \\
x \cdot (x \cdot (x \cdot (x \cdot \text{Power}(x, 1)))) &= \\
x \cdot (x \cdot (x \cdot (x \cdot (x \cdot 1))))
\end{align*}
\]

Informally: The time complexity is linear in $n$, because there are $n$ recursive calls of $\text{Power}$.

(Later we will see a recurrence equation for describing the time complexity of this algorithm.)

10. Consider the following definition of the power function, for $n \geq 0$:

\[
q(x, n) = \begin{cases} 
1 & \text{if } n = 0 \\
x \cdot q(x, \frac{n-1}{2})^2 & \text{if } n > 0 \text{ oneven} \\
q(x, \frac{n}{2})^2 & \text{if } n > 0 \text{ even}
\end{cases}
\]

Give a pseudocode description of an algorithm $Qower(x, n)$ to compute the power function according to this definition. As an example, calculate $Qower(x, 8)$ with a sequence of equations. Argue (informally) that the number of recursive calls is in $\Theta(\log n)$.

Solution:

Pseudo-code:

\begin{algorithm}
\textbf{Algorithm} $Qower(x, n)$:
\begin{algorithmic}
\STATE \textbf{Input}: positive integers $x$ and $n$
\STATE \textbf{Output}: the value $x^n$
\IF{$n = 0$}
\STATE \textbf{return} 1
\ENDIF
\IF{$n$ odd}
\STATE $y := Qower(x, \frac{n-1}{2})$
\STATE \textbf{return} $x \cdot y \cdot y$
\ELSE
\STATE $y := Qower(x, \frac{n}{2})$
\STATE \textbf{return} $y \cdot y$
\ENDIF
\end{algorithmic}
\end{algorithm}
Example for \( n = 8 \) and some \( x \); we write \( Q_x(n) \) for \( \text{Qower}(x, n) \).

\[
\begin{align*}
Q_x(8) & = Q_x(4) \\
Q_x(4) & = (Q_x(2) \cdot Q_x(2)) \\
& = (Q_x(1) \cdot Q_x(1)) \cdot (Q_x(1) \cdot Q_x(1)) \\
& = ((A \cdot A) \cdot (A \cdot A)) \cdot ((A \cdot A) \cdot (A \cdot A)) \\
& = ((x \cdot x) \cdot (x \cdot x)) \cdot ((x \cdot x) \cdot (x \cdot x))
\end{align*}
\]

with abbreviation \( A = x \cdot Q_x(0) \cdot Q_x(0) \) because \( x \cdot Q_x(0) \cdot Q_x(0) = x \cdot 1 \cdot 1 = x \).

Question: is the test of \( n \) being odd elementary?

Informally: The time complexity is determined by the recursive call on the input of half the size, and is in \( O(\log n) \).

(Also here a function for the time complexity can be given via a recurrence equation.)

11. We compare (sorting) algorithms \( I \) with running time function \( I(n) = 8n^2 \) and \( M \) with running time function \( M(n) = 64n \log n \). (The logarithm is for base 2.) For what \( n \) is \( I \) faster than \( M \)?

Solution: Intuitively, eventually \( M \) will be faster, but for relatively small \( n \) the constant 64 spoils the efficiency of \( M \).

For \( n = 2^5 = 32 \) we have \( I(n) = 2^3 \cdot 2^{10} = 2^{13} = 4 \cdot 2^{11} \) and \( M(n) = 2^6 \cdot 2^5 \cdot 5 = 5 \cdot 2^{11} \) so \( I(n) < M(n) \).

For \( n = 2^6 = 62 \) we have \( I(n) = 2^3 \cdot 2^{12} = 2^{15} = 8 \cdot 2^{12} \) and \( M(n) = 2^6 \cdot 2^6 \cdot 6 = 6 \cdot 2^{12} \) so \( I(n) > M(n) \).

So for \( n \leq 2^5 \) the quadratic algorithm \( I \) is faster than \( M \). Note that \( 2^5 \) is not an exact bound.

12. We compare algorithms \( P \) with running time \( P(n) = 2^5 \cdot n^2 \) and \( E \) with running time \( E(n) = 2^n \). For what \( n \) is \( E \) faster than \( P \)?

Solution: Intuitively, eventually \( E \) will be faster but for relatively small \( n \) the constant \( 2^5 \) spoils the efficiency of \( P \).

For \( n = 12 \) we have \( P(n) = 144 \cdot 2^5 \) and \( E(n) = 2^{12} = 2^7 \cdot 2^5 = 128 \cdot 2^5 \) so \( E(n) < P(n) \).

For \( n = 13 \) we have \( P(n) = 169 \cdot 2^5 \) and \( E(n) = 2^{13} = 2^8 \cdot 2^5 = 256 \cdot 2^5 \) so \( E(n) > P(n) \).

So for \( n \leq 12 \) the exponential algorithm \( E \) is faster than the quadratic algorithm \( P \).