1. For example: find a number with a certain property (even, prime, . . .) in a sequence of numbers.
   Or: check whether a graph contains a cycle.

2. Let $T(n)$ denote the minimum running time of algorithm $A$ for inputs of size $n$.
   The statement in the formulation of this exercise says that there is a function $f \in O(n^2)$ such that $T(n) \geq f(n)$ for all $n$.
   Take for $f$ the function $f(n) = 0$ for all $n$. Then $T(n) \geq f(n)$ trivially holds for all $n$.
   So the statement in the formulation of this exercise holds for any (non-negative) function $T(n)$.

3. No.
   That the worst-case time complexity of an algorithm isn’t in $O(n)$ only means there are (infinitely many) inputs and corresponding executions by the algorithm of which the execution time grows more than linearly, with regard to input size.
   But there can be other inputs for which the algorithm provides a very fast computation, faster than the algorithm with a worst-case time complexity in $O(n)$.

4. (a) $f = O(g)$, but $g \neq O(f)$.
   (b) $f = O(g)$ and $g = O(f)$.
   (c) $g = O(f)$, but $f \neq O(g)$.
   (d) $f = O(g)$, but $g \neq O(f)$.

5. This algorithm computes the maximum sum of values in $A[i..j]$, where $i$ ranges over \{1, . . . , $n$\} and $j$ ranges over \{i $-$ 1, . . . , $n$\}, and returns this sum. (The value $j$ = $i$ $-$ 1 represents the empty array, in which case the sum is 0.)
   Initialization (assigning 0 to max) takes $\Theta(1)$, and assigning 0 to sum at the start of each run of the outer for loop in total takes $\Theta(n)$.
   For each value of left in the outer for loop, the inner for loop takes $\Theta(n$ $-$ left $+$ 1) time Namely, the effort within the inner for loop for every value of right takes (at most) the following (number of) operations: look up the values of sum and $A[right]$; add these
values; assign the result as new value to \textit{sum}; look up the value of \textit{max}; check whether \textit{sum} is greater than \textit{max}; if so, assign \textit{sum} as new value to \textit{max}.

So overall, the worst-case time complexity of this algorithm is $\Theta(n+(n-1)+\cdots+1) = \Theta(\frac{1}{2}n(n+1)) = \Theta(n^2)$.

6. Pseudocode:

\begin{algorithm}
\textbf{Algorithm} \textit{Power}(x, n)
\begin{algorithmic}
\State \textbf{Input} : real number $x$ and non-negative integer $n$
\State \textbf{Output} : the value of $x^n$
\If {$n = 0$}
\State return 1
\Else
\State $y \leftarrow \textit{Power}(x, n-1)$
\State return $x \cdot y$
\EndIf
\end{algorithmic}
\end{algorithm}

The time complexity is linear in the size of $n$ because \textit{Power}(x, n) makes $n$ recursive subcalls in sequence: \textit{Power}(x, n - 1), \textit{Power}(x, n - 2), \ldots, \textit{Power}(x, 0). And each subcall has, on top of the subsequent subcall it makes, an overhead of $O(1)$.

For example, for $n = 5$ and some $x$:

\[
\begin{align*}
\textit{Power}(x, 5) &= x \cdot \textit{Power}(x, 4) \\
&= x \cdot (x \cdot \textit{Power}(x, 3)) \\
&= x \cdot (x \cdot (x \cdot \textit{Power}(x, 2))) \\
&= x \cdot (x \cdot (x \cdot (x \cdot \textit{Power}(x, 1)))) \\
&= x \cdot (x \cdot (x \cdot (x \cdot (x \cdot \textit{Power}(x, 0)))))) \\
&= x \cdot (x \cdot (x \cdot (x \cdot (x \cdot 1))))
\end{align*}
\]

7. Pseudocode:

\begin{algorithm}
\textbf{Algorithm} \textit{Qower}(x, n)
\begin{algorithmic}
\State \textbf{Input} : real number $x$ and non-negative integer $n$
\State \textbf{Output} : the value of $x^n$
\If {$n = 0$}
\State return 1
\Else
\If {$n$ is even}
\State $y \leftarrow \textit{Qower}(x, \frac{n}{2})$
\State return $y \cdot y$
\Else
\State $y \leftarrow \textit{Qower}(x, \frac{n-1}{2})$
\State return $x \cdot y \cdot y$
\EndIf
\EndIf
\end{algorithmic}
\end{algorithm}

The time complexity is logarithmic in the size of $n$ because \textit{Qower}(x, n) makes in the order of $\log_2 n$ recursive subcalls in sequence. For instance, if $n = 2^k$: \textit{Qower}(x, 2^{k-1}), \textit{Qower}(x, 2^{k-2}), \ldots, \textit{Qower}(x, 2^0), \textit{Qower}(x, 0). And each subcall has, on top of the subsequent subcall it makes, an overhead of $O(1)$.
For example, for \( n = 5 \) and some \( x \):

\[
Qower(x, 5) = x \cdot y \cdot y \quad (\text{with } y = Qower(x, 2))
\]
\[
= x \cdot y' \cdot y' \cdot y' \quad (\text{with } y' = Qower(x, 1))
\]
\[
= x \cdot x \cdot y'' \cdot x \cdot y'' \cdot y'' \cdot x \cdot y'' \cdot y'' \quad (\text{with } y'' = Qower(x, 0) = 1)
\]
\[
= x \cdot x \cdot x \cdot x \cdot x
\]

8. [5] [3, 5] [5] [2, 5] [8, 2, 5] [2, 5] [5] [9, 5] [1, 9, 5]

9. We start with an empty stack, and read the arithmetic expression from left to right, one symbol at a time.

In case a symbol isn’t a bracket, we leave the stack unchanged.

If the symbol is an opening bracket, we push it onto the stack.

If the symbol is a closing bracket, we pop an (opening) bracket from the stack.

If at the end the stack is empty, then we report that the brackets are placed correctly.

If at the end the stack is non-empty, or at some point we try to pop from an empty stack, then we report that the brackets are not placed correctly.

10. The idea is to let one stack start at index 0 of the array, going upward, while the other stack starts at index \( n - 1 \) of the array, going down.

Variable \( t_1 \), initially carrying the value \(-1\), points at the highest index in the array occupied by the first stack. When an element is pushed onto or popped from the first stack, \( t_1 \) is increased or decreased by one, respectively.

Likewise, variable \( t_2 \), initially carrying the value \( n \), points at the lowest index in the array occupied by the second stack. When an element is pushed onto or popped from the second stack, \( t_2 \) is decreased or increased by one, respectively.

Only if we try to push an element onto a stack while \( t_2 = t_1 + 1 \), the array remains unchanged, and an overflow message is produced as response.