Algorithm 3.2.1 Finding the Maximum of Three Numbers.

Input:	Three numbers a, b, and c
Output:	x, the largest of a, b, and c

```
procedure $max(a, b, c)$
    $x := a$
    if $b > x$ then
        $x := b$
    if $c > x$ then
        $x := c$
    return($x$)
end $max$
```

Algorithm 3.2.2 Finding the Largest Element in a Finite Sequence.

Input: The sequence $s_1, s_2, \ldots, s_n$ and the length n of the sequence
Output: *large*, the largest element in this sequence

```
procedure $find\_large(s, n)$
   $large := s_1$
   $i := 2$
   while $i \leq n$ do
      begin
      if $s_i > large$ then
         $large := s_i$
      $i := i + 1$
      end
   return($large$)
end $find\_large$
```

Algorithm 3.2.3 Finding the Largest Element in a Finite Sequence.

Input: The sequence $s_1, s_2, \ldots, s_n$ and the length n of the sequence
Output: *large*, the largest element in this sequence

```
procedure $find\_large(s, n)$
    $large := s_1$
    for $i := 2$ to $n$ do
        if $s_i > large$ then
            $large := s_i$
    return($large$)
end $find\_large$
```

Algorithm 3.2.4 Testing Whether a Positive Integer is Prime.

Input: m, a positive integer
Output: **true**, if m is prime; **false**, if m is not prime

```
procedure is_prime(m)
  for i := 2 to m - 1 do
    if m mod i = 0 then
      return(false)
  return(true)
end is_prime
```

Algorithm 3.2.5 Finding a Prime Larger Than a Given Integer.

Input: n, a positive integer
Output: m, the smallest prime greater than n

```
procedure large_prime(n)
    m := n + 1
    while not is_prime(m) do
        m := m + 1
    return(m)
end large_prime
```

Algorithm 3.3.7 Euclidean Algorithm.

Input: a and b (nonnegative integers, not both zero)
Output: Greatest common divisor of a and b

```
procedure gcd($a, b$)
  if $a < b$ then
    swap($a, b$)
  while $b \neq 0$ do
    begin
    $r := a \bmod b$
    $a := b$
    $b := r$
    end
  return($a$)
end gcd
```

Algorithm 3.4.2 Computing n Factorial.

Input: n, an integer greater than or equal to 0
Output: $n!$

```
procedure factorial(n)
   if $n = 0$ then
      return(1)
   return($n * factorial(n - 1)$)
end factorial
```

Algorithm 3.4.5 Recursively Computing the Greatest Common Divisor.

Input: a and b (nonnegative integers, not both zero)
Output: Greatest common divisor of a and b

```
procedure gcd_recurs(a, b)
   if $a < b$ then
      swap(a, b)
   if $b = 0$ then
      return(a)
   $r := a \bmod b$
   return(gcd_recurs(b, r))
end gcd_recurs
```

Algorithm 3.4.7 Robot Walking.

Input: n
Output: walk(n)

```
procedure robot_walk(n)
    if n = 1 or n = 2 then
        return(n)
    return(robot_walk(n − 1) + robot_walk(n − 2))
end robot_walk
```

Algorithm 3.5.16 Searching an Unordered Sequence.

Input: $s_1, s_2, \ldots, s_n, n$, and *key* (the value to search for)
Output: The location of *key*, or if *key* is not found, 0

```
procedure linear_search(s, n, key)
   for i := 1 to n do
      if key = $s_i$ then
         return(i)
   return(0)
end linear_search
```

Algorithm 4.3.9 Generating Combinations.

Input: r, n

Output: All r-combinations of $\{1, 2, \ldots, n\}$ in increasing lexicographic order.

```
procedure combination($r, n$)
    for $i := 1$ to $r$ do
        $s_i := i$
    print $s_1, \ldots, s_r$
    for $i := 2$ to $C(n, r)$ do
        begin
        $m := r$
        $max\_val := n$
        while $s_m = max\_val$ do
            begin
            $m := m - 1$
            $max\_val := max\_val - 1$
            end
        $s_m := s_m + 1$
        for $j := m + 1$ to $r$ do
            $s_j := s_{j-1} + 1$
        print $s_1, \ldots, s_r$
        end
end combination
```

Algorithm 4.3.14 Generating Permutations.

Input: n
Output: All permutations of $\{1, 2, \ldots, n\}$ in increasing lexicographic order.

```
procedure permutation($n$)
    for $i := 1$ to $n$ do
        $s_i := i$
    print $s_1, \ldots, s_n$
    for $i := 2$ to $n!$ do
        begin
        $m := n - 1$
        while $s_m > s_{m+1}$ do
            $m := m - 1$
        $k := n$
        while $s_m > s_k$ do
            $k := k - 1$
        swap($s_m, s_k$)
        $p := m + 1$
        $q := n$
        while $p < q$ do
            begin
            swap($s_p, s_q$)
            $p := p + 1$
            $q := q - 1$
            end
        print $s_1, \ldots, s_n$
        end
end permutation
```

Algorithm 5.3.1 Selection Sort.

Input: $s_1, s_2, \ldots, s_n$ and the length n of the sequence
Output: $s_1, s_2, \ldots, s_n$, arranged in increasing order

```
procedure selection_sort(s, n)
    if $n = 1$ then
        return
    $max\_index := 1$
    for $i := 2$ to $n$ do
        if $s_i > s_{max\_index}$ then
            $max\_index := i$
    swap($s_n, s_{max\_index}$)
    call selection_sort($s, n - 1$)
end selection_sort
```

Algorithm 5.3.2 Binary Search.

Input: A sequence $s_i, s_{i+1}, \ldots, s_j$, $i \geq 1$, sorted in increasing order, a value *key*, i, and j

Output: The output is an index k for which $s_k = key$, or if *key* is not in the sequence, the output is the value 0.

```
procedure binary_search(s, i, j, key)
    if $i > j$ then
        return(0)
    $k := \lfloor (i + j)/2 \rfloor$
    if $key = s_k$ then
        return($k$)
    if $key < s_k$ then
        $j := k - 1$
    else
        $i := k + 1$
    return(binary_search($s, i, j, key$))
end binary_search
```

Algorithm 5.3.5 Merging Two Sequences.

Input: Two increasing sequences: $s_i, \ldots, s_m$ and $s_{m+1}, \ldots, s_j$, and indexes i, m, and j

Output: The sequence $c_i, \ldots, c_j$ consisting of the elements $s_i, \ldots, s_m$ and $s_{m+1}, \ldots, s_j$ combined into one increasing sequence

```
procedure merge($s, i, m, j, c$)
  $p := i$
  $q := m + 1$
  $r := i$
  while $p \leq m$ and $q \leq j$ do
    begin
    if $s_p < s_q$ then
      begin
      $c_r := s_p$
      $p := p + 1$
      end
    else
      begin
      $c_r := s_q$
      $q := q + 1$
      end
    $r := r + 1$
    end
  while $p \leq m$ do
    begin
    $c_r := s_p$
    $p := p + 1$
    $r := r + 1$
    end
  while $q \leq j$ do
    begin
    $c_r := s_q$
    $q := q + 1$
    $r := r + 1$
```

```
        end
end merge
```

Algorithm 5.3.8 Merge Sort.

Input: $s_i, \ldots, s_j$, i, and j
Output: $s_i, \ldots, s_j$ arranged in increasing order

```
procedure merge_sort(s, i, j)
    if $i = j$ then
        return
    $m := \lfloor (i + j)/2 \rfloor$
    call merge_sort(s, i, m)
    call merge_sort(s, m + 1, j)
    call merge(s, i, m, j, c)
    for $k := i$ to $j$ do
        $s_k := c_k$
end merge_sort
```

Algorithm 6.4.1 Dijkstra's Shortest-Path Algorithm.

Input: A connected, weighted graph in which all weights are positive. Vertices a and z.

Output: $L(z)$, the length of a shortest path from a to z.

```
procedure dijkstra(w, a, z, L)
    $L(a) := 0$
    for all vertices $x \neq a$ do
        $L(x) := \infty$
    $T :=$ set of all vertices
    while $z \in T$ do
        begin
        choose $v \in T$ with minimum $L(v)$
        $T := T - \{v\}$
        for each $x \in T$ adjacent to $v$ do
            $L(x) := \min\{L(x), L(v) + w(v, x)\}$
        end
end dijkstra
```

Algorithm 7.1.9 Constructing an Optimal Huffman Code.

Input: A sequence of n frequencies, $n \geq 2$
Output: A rooted tree that defines an optimal Huffman code

```
procedure huffman($f, n$)
    if $n = 2$ then
        begin
        let $f_1$ and $f_2$ denote the frequencies
        let $T$ be as in Figure 7.1.11
        return($T$)
        end
    let $f_i$ and $f_j$ denote the smallest frequencies
    replace $f_i$ and $f_j$ in the list $f$ by $f_i + f_j$
    $T' :=$ huffman($f, n - 1$)
    replace a vertex in $T'$ labeled $f_i + f_j$ by the tree shown in Figure 7.1.12
        to obtain the tree $T$
    return($T$)
end huffman
```

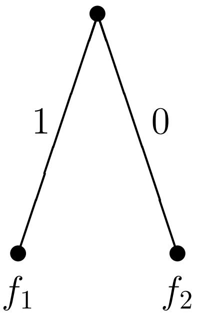

Figure 7.1.11

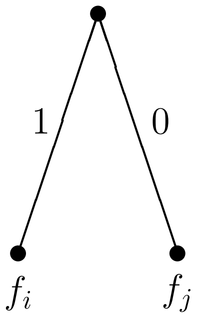

Figure 7.1.12

Algorithm 7.3.6 Breadth-First Search for a Spanning Tree.

Input: A connected graph G with vertices ordered $v_1, v_2, \ldots, v_n$
Output: A spanning tree T

```
procedure bfs(V, E)
    $S := (v1)$
    $V' := \{v_1\}$
    $E' := \emptyset$
    while true do
        begin
        for each $x \in S$, in order, do
            for each $y \in V - V'$, in order, do
                if $(x, y)$ is an edge then
                    add edge $(x, y)$ to $E'$ and $y$ to $V'$
        if no edges were added then
            return($T$)
        $S :=$ children of $S$ ordered consistently with the original vertex ordering
        end
end bfs
```

Algorithm 7.3.7 Depth-First Search for a Spanning Tree.

Input: A connected graph G with vertices ordered $v_1, v_2, \ldots, v_n$
Output: A spanning tree T

```
procedure $dfs(V, E)$
    $V' := \{v_1\}$
    $E' := \emptyset$
    $w := v_1$
    while true do
        begin
        while there is an edge $(w, v)$ that when added to $T$ does not create
            a cycle in $T$ do
            begin
            choose the edge $(w, v_k)$ with minimum $k$ that when added to $T$
                does not create a cycle in $T$
            add $(w, v_k)$ to $E'$
            add $v_k$ to $V'$
            $w := v_k$
            end
        if $w = v_1$ then
            return($T$)
        $w :=$ parent of $w$ in $T$
        end
end $dfs$
```

Algorithm 7.3.10 Solving the Four-Queens Problem Using Backtracking.

Input: An array *row* of size 4

Output: **true**, if there is a solution
false, if there is no solution
[If there is a solution, the kth queen is in column k, row $row(k)$.]

```
procedure four_queens(row)
   $k := 1$
   $row(1) := 0$
   while $k > 0$ do
      begin
      $row(k) := row(k) + 1$
      while $row(k) \leq 4$ and column $k$, $row(k)$ conflicts do
         $row(k) := row(k) + 1$
      if $row(k) \leq 4$ then
         if $k = 4$ then
            return(true)
         else
            begin
            $k := k + 1$
            $row(k) := 0$
            end
      else
         $k := k - 1$
      end
   return(false)
end four_queens
```

Algorithm 7.4.3 Prim's Algorithm.

Input: A connected, weighted graph with vertices $1, \ldots, n$ and start vertex s. If (i, j) is an edge, $w(i, j)$ is equal to the weight of (i, j); if (i, j) is not an edge, $w(i, j)$ is equal to ∞ (a value greater than any actual weight).

Output: The set of edges E in a minimal spanning tree

```
procedure $prim(w, n, s)$
  for $i := 1$ to $n$ do
    $v(i) := 0$
  $v(s) := 1$
  $E := \emptyset$
  for $i := 1$ to $n - 1$ do
    begin
    $min := \infty$
    for $j := 1$ to $n$ do
      if $v(j) = 1$ then
        for $k = 1$ to $n$ do
          if $v(k) = 0$ and $w(j, k) < min$ then
            begin
            $add\_vertex := k$
            $e := (j, k)$
            $min := w(j, k)$
            end
    $v(add\_vertex) := 1$
    $E := E \cup \{e\}$
    end
  return$(E)$
end $prim$
```

Algorithm 7.5.10 Constructing a Binary Search Tree.

Input: A sequence $w_1, \ldots, w_n$ of distinct words and the length n of the sequence

Output: A binary search tree T

```
procedure make_bin_search_tree($w, n$)
  let $T$ be the tree with one vertex, root
  store $w_1$ in root
  for $i := 2$ to $n$ do
    begin
    $v :=$ root
    search := true
    while search do
      begin
      $s :=$ word in $v$
      if $w_i < s$ then
        if $v$ has no left child then
          begin
          add a left child $l$ to $v$
          store $w_i$ in $l$
          search := false
          end
        else
          $v :=$ left child of $v$
      else
        if $v$ has no right child then
          begin
          add a right child $r$ to $v$
          store $w_i$ in $r$
          search := false
          end
        else
          $v :=$ right child of $v$
      end
    end
```

```
    return($T$)
end make_bin_search_tree
```

Algorithm 7.6.1 Preorder Traversal.

Input: PT, the root of a binary tree
Output: Dependent on how "process" is interpreted

```
procedure preorder(PT)
    if PT is empty then
        return
    process PT
    l := left child of PT
    preorder(l)
    r := right child of PT
    preorder(r)
end preorder
```

Algorithm 7.6.3 Inorder Traversal.

Input: PT, the root of a binary tree
Output: Dependent on how "process" is interpreted

```
procedure inorder(PT)
   if PT is empty then
      return
   l := left child of PT
   inorder(l)
   process PT
   r := right child of PT
   inorder(r)
end inorder
```

Algorithm 7.6.5 Postorder Traversal.

Input: PT, the root of a binary tree
Output: Dependent on how "process" is interpreted

```
procedure postorder(PT)
    if PT is empty then
        return
    l := left child of PT
    postorder(l)
    r := right child of PT
    postorder(r)
    process PT
end postorder
```

Algorithm 7.8.13 Testing Whether Two Binary Trees Are Isomorphic.

Input: The roots r_1 and r_2 of two binary trees. (If the first tree is empty, r_1 has the special value *null*. If the second tree is empty, r_2 has the special value *null*.)

Output: **true**, if the trees are isomorphic
false, if the trees are not isomorphic

```
procedure bin_tree_isom($r_1, r_2$)
  if $r_1 = null$ and $r_2 = null$ then
    return(true)
  if $r_1 = null$ or $r_2 = null$ then
    return(false)
  $lc\_r_1$ := left child of $r_1$
  $lc\_r_2$ := left child of $r_2$
  $rc\_r_1$ := right child of $r_1$
  $rc\_r_2$ := right child of $r_2$
  return(bin_tree_isom($lc\_r_1, lc\_r_2$) and bin_tree_isom($rc\_r_1, rc\_r_2$))
end bin_tree_isom
```

Algorithm 8.2.4 Finding a Maximal Flow in a Network.

Input: A network with source a, sink z, capacity C, vertices $a = v_0, \ldots, v_n = z$, and n

Output: A maximal flow F

```
procedure $max\_flow(a, z, C, v, n)$
   for each edge $(i, j)$ do
      $F_{ij} := 0$
   while true do
      begin
      for $i := 0$ to $n$ do
         begin
         $predecessor(v_i) := null$
         $val(v_i) := null$
         end
      $predecessor(a) := -$
      $val(a) := \infty$
      $U := \{a\}$
      while $val(z) = null$ do
         begin
         if $U = \emptyset$ then
            return$(F)$
         choose $v$ in $U$
         $U := U - \{v\}$
         $\Delta := val(v)$
         for each edge $(v, w)$ with $val(w) = null$ do
            if $F_{vw} < C_{vw}$ then
               begin
               $predecessor(w) := v$
               $val(w) := \min\{\Delta, C_{vw} - F_{vw}\}$
               $U := U \cup \{w\}$
               end
         for each edge $(w, v)$ with $val(w) = null$ do
            if $F_{wv} > 0$ then
               begin
```

```
                $predecessor(w) := v$
                $val(w) := \min\{\Delta, F_{wv}\}$
                $U := U \cup \{w\}$
                **end**
            **end**
        $w_0 := z$
        $k := 0$
        **while** $w_k \neq a$ **do**
            **begin**
            $w_{k+1} := predecessor(w_k)$
            $k := k + 1$
            **end**
        $P := (w_k, w_{k-1}, \ldots, w_1, w_0)$
        $\Delta := val(z)$
        **for** $i := 1$ **to** $k$ **do**
            **begin**
            $e := (w_i, w_{i-1})$
            **if** $e$ is properly oriented in $P$ **then**
                $F_e := F_e + \Delta$
            **else**
                $F_e := F_e - \Delta$
            **end**
        **end**
**end** $max\_flow$
```

Algorithm 11.1.2 Finding the Distance Between a Closest Pair of Points.

Input: $p_1, \ldots, p_n$ ($n \geq 2$ points in the plane)
Output: δ, the distance between a closest pair of points

```
procedure closest_pair(p, n)
    sort $p_1, \ldots, p_n$ by $x$-coordinate
    return(rec_cl_pair(p, 1, n))
end closest_pair

procedure rec_cl_pair(p, i, j)
    if $j - i < 3$ then
        begin
        sort $p_i, \ldots, p_j$ by $y$-coordinate
        directly find the distance $\delta$ between a closest pair
        return($\delta$)
        end
    $k := \lfloor (i + j)/2 \rfloor$
    $l := p_k.x$
    $\delta_L :=$ rec_cl_pair(p, i, k)
    $\delta_R :=$ rec_cl_pair(p, k + 1, j)
    $\delta := \min\{\delta_L, \delta_R\}$
    merge $p_i, \ldots, p_k$ and $p_{k+1}, \ldots, p_j$ by $y$-coordinate
    $t := 0$
    for $k := i$ to $j$ do
        if $p_k.x > l - \delta$ and $p_k.x < l + \delta$ then
            begin
            $t := t + 1$
            $v_t := p_k$
            end
    for $k := 1$ to $t - 1$ do
        for $s := k + 1$ to $\min\{t, k + 7\}$ do
            $\delta := \min\{\delta, \text{dist}(v_k, v_s)\}$
    return($\delta$)
end rec_cl_pair
```

Algorithm 11.3.6 Graham's Algorithm to Compute the Convex Hull.

Input: $p_1, \ldots, p_n$ and n
Output: $p_1, \ldots, p_k$ (the convex hull of $p_1, \ldots, p_n$) and k

```
procedure graham_scan(p, n, k)
    if $n = 1$ then
        begin
        $k := 1$
        return
        end
    $min := 1$
    for $i := 2$ to $n$ do
        if $p_i.y < p_{min}.y$ then
            $min := i$
    for $i := 1$ to $n$ do
        if $p_i.y = p_{min}.y$ and $p_i.x < p_{min}.x$ then
            $min := i$
    swap($p_1, p_{min}$)
    sort $p_2, \ldots, p_n$
    $p_0 := p_n$
    $k := 2$
    for $i := 3$ to $n$ do
        begin
        while $p_{k-1}, p_k, p_i$ do not make a left turn do
            $k := k - 1$
        $k := k + 1$
        swap($p_i, p_k$)
        end
end graham_scan
```