```
Algorithm NextValue(k)
// x[1],...,x[k-1] have been assigned integer values in
// the range [1,m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0,m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k,j] ≠ 0) and (x[k] = x[j]))
            // If (k,j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then     // At most m colors have been
                            // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}
```

```
Algorithm HeapSort(a, n)
// a[1 : n] contains n elements to be sorted. HeapSort
// rearranges them inplace into nondecreasing order.
{
    Heapify(a, n); // Transform the array into a heap.
    // Interchange the new maximum with the element
    // at the end of the array.
    for i := n to 2 step -1 do
    {
        t := a[i]; a[i] := a[1]; a[1] := t;
        Adjust(a, 1, i - 1);
    }
}
```

```
Algorithm Heapify(a, n)
// Readjust the elements in a[1 : n] to form a heap.
{
    for i := ⌊n/2⌋ to 1 step −1 do Adjust(a, i, n);
}
```

```
Algorithm Adjust(a, i, n)
// The complete binary trees with roots 2i and 2i + 1 are
// combined with node i to form a heap rooted at i. No
// node has an address greater than n or less than 1.
{
    j := 2i; item := a[i];
    while (j ≤ n) do
    {
        if ((j < n) and (a[j] < a[j + 1])) then j := j + 1;
            // Compare left and right child
            // and let j be the larger child.
        if (item ≥ a[j]) then break;
            // A position for item is found.
        a[⌊j/2⌋] := a[j]; j := 2j;
    }
    a[⌊j/2⌋] := item;
}
```

```
Algorithm DelMax(a, n, x)
// Delete the maximum from the heap a[1 : n] and store it in x.
{
    if (n = 0) then
    {
        write ("heap is empty"); return false;
    }
    x := a[1]; a[1] := a[n];
    Adjust(a, 1, n − 1); return true;
}
```

```
Algorithm Insert(a, n)
{
    // Inserts a[n] into the heap which is stored in a[1 : n − 1].
    i := n; item := a[n];
    while ((i > 1) and (a[⌊i/2⌋] < item)) do
    {
        a[i] := a[⌊i/2⌋]; i := ⌊i/2⌋;
    }
    a[i] := item; return true;
}
```

```
Algorithm Sort(a, n)
// Sort the elements a[1 : n].
{
    for i := 1 to n do Insert(a, i);
    for i := n to 1 step -1 do
    {
        DelMax(a, i, x); a[i] := x;
    }
}
```

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h]; h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

```
Algorithm RQuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order. a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then
    {
        if ((q − p) > 5) then
            Interchange(a, Random() mod (q − p + 1) + p, p);
        j := Partition(a, p, q + 1);
            // j is the position of the partitioning element.
        RQuickSort(p, j − 1);
        RQuickSort(j + 1, q);
    }
}
```

```
Algorithm SumOfSub(s, k, r)
// Find all subsets of w[1 : n] that sum to m. The values of x[j],
// 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
// It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
    // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
    x[k] := 1;
    if (s + w[k] = m) then write (x[1 : k]); // Subset found
        // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
    else if (s + w[k] + w[k + 1] ≤ m)
        then SumOfSub(s + w[k], k + 1, r - w[k]);
    // Generate right child and evaluate B_k.
    if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
    {
        x[k] := 0;
        SumOfSub(s, k + 1, r - w[k]);
    }
}
```