Analysis of Algorithms: Solutions 5

```
                                          X
                                          X
                                          X
                        X  X           X  X
number of            X  X  X  X     X  X  X
homeworks            X  X  X  X  X  X  X  X
                  X  X  X  X  X  X  X  X  X
                  X  X  X  X  X  X  X  X  X
            --------------------------------
            0  1  2  3  4  5  6  7  8  9  10
                         grades
```

The histogram shows the distribution of grades for the homeworks submitted on time.

Problem 1
Suppose that we apply the CONNECTED-COMPONENTS algorithm to an undirected graph G, with vertices $G[V] = \{a, b, c, d, e, f, g, h, i, j, k\}$, and its edges $E[G]$ are processed in the following order: $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Using Figure 22.1 in the textbook as a model, illustrate the steps of CONNECTED-COMPONENTS on this graph.

The final result of applying CONNECTED-COMPONENTS is the following three sets:

$$\{a, e, h\} \quad \{b, d, f, g, i, j, k\} \quad \{c\}$$

Problem 2
Write pseudocode for MAKE-SET, FIND-SET, and UNION, using the linked-list representation of disjoint sets. Your UNION algorithm must always append the shorter list to the longer one. For every linked list, you will need to store its size, a pointer to the first element, and a pointer to the last element.

We use four fields for each element x of a linked list:

> *next*$[x]$: pointer to the next element of the list; NIL if x is the last element
> *rep*$[x]$: pointer to the set representative, that is, to the first element of the list
> *last*$[x]$: if x is the first element of a list, then this field points to the last element
> *size*$[x]$: if x is the first element, then this field contains the size of the list

If x is not the first element of a list, then the algorithms do *not* use its *last* and *size* fields, and the information in these fields may be incorrect.

$\text{MAKE-SET}(x)$
$next[x] \leftarrow \text{NIL}$
$rep[x] \leftarrow x$
$last[x] \leftarrow x$
$size[x] \leftarrow 1$

$\text{FIND-SET}(x)$
return $rep[x]$

$\text{UNION}(x)$
if $size[rep[x]] > size[rep[y]]$
    then $\text{APPEND}(rep[x], rep[y])$
    else $\text{APPEND}(rep[y], rep[x])$

$\text{APPEND}(x, y)$
$next[last[x]] \leftarrow y$
$size[x] \leftarrow size[x] + size[y]$
$z \leftarrow y$
while $z \neq \text{NIL}$ $\triangleright$ Change the *rep* pointers in the second list.
    do $rep[z] \leftarrow x$
        $z \leftarrow next[z]$

Problem 3
Write a *nonrecursive* version of the FIND-SET algorithm with path compression, for disjoint-set forests.

The following algorithm uses a stack S to store the nodes on the path from a given element x to the root. The algorithm finds the root z and sets the *parent* pointers of all nodes in S to z.

$\text{FIND-SET}(x)$
initialize an empty stack S
$z \leftarrow x$
while $z \neq parent[z]$ $\triangleright$ Find the root.
    do $\text{PUSH}(S, z)$
        $z \leftarrow parent[z]$
while not $\text{STACK-EMPTY}(S)$ $\triangleright$ Set the parent pointers to the root z.
    do $y \leftarrow \text{Pop}(S)$
        $parent[y] \leftarrow z$
return z

Problem 4
Write an algorithm CHECK-LOOP(x) that determines whether a given linked list is "looped." The algorithm's argument x is the first element of the list. If a linked list is looped, then CHECK-LOOP returns TRUE; if the list is not looped, then it returns FALSE.

We may think of a linked list as a one-way road. To check whether it is looped, we simultaneously send two cars, one of which is faster than the other. If the road is *not* looped, the faster car reaches the end in linear time. On the other hand, if the road is looped, then the faster car overtakes the other one somewhere on the loop, which also happens in linear time.

To implement this algorithm, we assume that every element y of the linked list has a field $next[y]$, which points to the next element. If y is the last element, then $next[y]$ is NIL.

```
CHECK-LOOP(x)
if x = NIL
    then return FALSE
slow ← x    ▷ Position of the slow car.
fast ← next[x]    ▷ Position of the fast car.
while fast ≠ NIL and fast ≠ slow
    do fast ← next[fast]
        if fast ≠ NIL
            then fast ← next[fast]
        slow ← next[slow]
if fast = NIL
    then return FALSE
    else return TRUE
```