# 9. Ordered sets

## Learning objectives:

- searching in ordered sets
- sequential search. proof of program correctness
- binary search
- in-place permutation
- nondeterministic algorithms
- cycle rotation
- cycle clipping

Sets of elements processed on a computer are always ordered according to some criterion. In the preceding example of the "population count" operation, a set is ordered arbitrarily and implicitly simply because it is mapped onto linear storage; a programmer using that set can ignore any order imposed by the implementation and access the set through functions that hide irrelevant details. In most cases, however, the order imposed on a set is not accidental, but is prescribed by the problem to be solved and/or the algorithm to be used. In such cases the programmer explicitly deals with issues of how to order a set and how to use any existing order to advantage.

Searching in ordered sets is one of the most frequent tasks performed by computers: whenever we operate on a data item, that item must be selected from a set of items. Searching is also an ideal ground for illustrating basic concepts and techniques of programming.

At times, ordered sets need to be rearranged (permuted). The chapter “Sorting and its complexity” is dedicated to the most frequent type of rearrangement: permuting a set of elements into ascending order. Here we discuss another type of rearrangement: reordering a set according to a given permutation.

## Sequential search

Consider the simple case where a fixed set of n data elements is given in an array A:

```
const  n = … ;  { n > 0 }
type  index = 0 .. n;  elt = … ;
var  A: array[1 .. n] of elt;    or    var A: array[0 .. n] of elt;
```

Sequential or linear search is the simplest technique for determining whether A contains a given element x. It is a trivial example of an *incremental algorithm*, which processes a set of data one element at a time. If the search for x is successful, we return an index i, 1 ≤ i ≤ n, to point to x. The convention that i = 0 signals unsuccessful search is convenient and efficient, as it encodes all possible outcomes in a single parameter.

```
function find(x: elt): index;
var  i: index;
begin
   i := n;
   while  (i > 0)  { can access A }  cand  (A[i] ≠ x)  { not yet
found }  do
(1)   { (1 ≤ i ≤ n) ∧ (∀ k, i ≤ k: A[k] ≠ x) }
      i := i - 1;
```

```
(2)    { (∀k, i < k: A[k] ≠ x) ∧ ((i= 0) ∧ ((1 ≤ i ≤ n) ∧ (A[i] = x))) }
       return(i)
  end;
```

The 'cand' operator used in the termination condition is the *conditional* 'and'. Evaluation proceeds from left to right and stops as soon as the value of the boolean expression is determined: If i > 0 yields 'false', we immediately terminate evaluation of the boolean expression without accessing A[i], thus avoiding an out-of-bounds error.

We have included two assertions, (1) and (2), that express the main points necessary for a formal proof of correctness: mainly, that each iteration of the loop extends by one element the subarray known *not* to contain the search argument x. Assertion (1) is trivially true after the initialization i := n, and remains true whenever the body of the while loop is about to be executed. Assertion (2) states that the loop terminates in one of two ways:

- i = 0 signals that the entire array has been scanned unsuccessfully.
- x has been found at index i.

A formal correctness proof would have to include an argument that the loop does indeed terminate—a simple argument here, since i is initialized to n, decreases by 1 in each iteration, and thus will become 0 after a finite number of steps.

The loop is terminated by a Boolean expression composed of two terms: reaching the end of the array, i = 0, and testing the current array element, A[i] = x. The second term is unavoidable, but the first one can be spared by making sure that x is always found before the index i drops off the end of the array. This is achieved by extending the array by one cell A[0] and placing the search argument x in it as a *sentinel*. If no true element x stops the scan of the array, the sentinel will. Upon exit from the loop, the value of i reveals the outcome of the search, with the convention that 0 signals an unsuccessful search:

```
function find(x: elt): index;
var  i: index;
begin
   A[0] := x;  i := n;
   while  A[i] ≠ x  do  i := i - 1;
   return(i)
end;
```

How efficient is sequential search? An unsuccessful search always scans the entire array. If all n array elements have equal probability of being searched for, the average number of iterations of the while loop in a successful search is

$$\frac{1}{n}(1+2+\mathrm{K}\ +n)=\frac{n+1}{2}.$$

This algorithm needs time proportional to n in the *average* and the *worst case*.

## Binary search

If the data elements stored in the array A are ordered according to the order relation ≤ defined on their domain, that is

∀ k, 1 ≤ k < n: A[k] ≤ A[k + 1]

the search for an element x can be made much faster because a comparison of x with any array element A[m] provides more information than it does in the unordered case. The result x ≠ A[m] excludes not only A[m], but also all elements on one or the other side of A[m], depending on whether x is greater or smaller than A[m] (Exhibit 9.1).

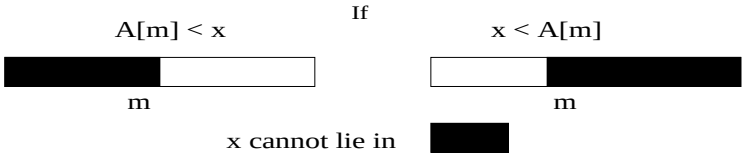


Exhibit 9.1: Binary search identifies regions where the search argument is guaranteed to be absent.

The following function exploits this additional information:

```
  const  n = … ;  { n > 0 }
  type  index = 1 .. n;  elt = … ;
  var  A: array[1 .. n] of elt;

  function find(x: elt;  var m: index): boolean;
  var  u, v: index;
  begin
    u := 1;  v := n;
    while  u ≤ v  do  begin
(1)    { (u ≤ v) ∧ (∀ k, 1 ≤ k < u: A[k] < x) ∧(∀ k, v < k ≤ n: A[k] >
x) }
      m := any value such that u ≤ m ≤ v ;
      if    x < A[m]     thenv := m – 1
      elsif   x > A[m]   then u := m + 1
(2)     else  {x = A[m] }    return(true)
    end;
(3)    { (u = v + 1) ∧(∀ k, 1 ≤ k < u: A[k]< x) ∧ (∀ k, v < k ≤ n:
A[k] > x) }
    return(false)
  end;
```

u and v bound the interval of uncertainty that might contain x. Assertion (1) states that A[1], … , A[u – 1] are known to be smaller than x; A[v + 1], … , A[n] are known to be greater than x. Assertion (2), before exit from the function, states that x has been found at index m. In assertion (3), u = v + 1 signals that the interval of uncertainty has shrunk to become empty. If there exists more than one match, this algorithm will find one of them.

This algorithm is correct independently of the choice of m but is most efficient when m is the midpoint of the current search interval:

```
m := (u + v) div 2;
```

With this choice of m each comparison either finds x or eliminates half of the remaining elements. Thus at most $\lceil \log_2 n \rceil$ iterations of the loop are performed in the worst case.

## Exercise: binary search

The array

```
var A: array [1 .. n] of integer;
```

contains n integers in ascending order: A[1] ≤ A[2] ≤ … ≤ A[n].

(a) Write a recursive binary search

```
function rbs (x, u, v: integer): integer;
```

that returns 0 if x is not in A, and an index i such that A[i] = x if x is in A.

(b) What is the maximal depth of recursive calls of 'rbs' in terms of n?

(c) Describe the advantages and disadvantages of this recursive binary search as compared to the iterative binary search.

## Exercise: searching in a partially ordered two-dimensional array

Consider the n by m array:

```
var  A: array[1 .. n, 1 .. m] of integer;
```

and assume that the integers in each row and in each column are in ascending order; that is,

$A[i, j] \le A[i, j + 1]$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m - 1$;
$A[i, j] \le A[i + 1, j]$ for $i = 1, \ldots, n - 1$ and $j = 1, \ldots, m$.

(a) Design an algorithm that determines whether a given integer x is stored in the array A. Describe your algorithm in words and figures. *Hint:* Start by comparing x with A[1, m] (Exhibit 9.2).

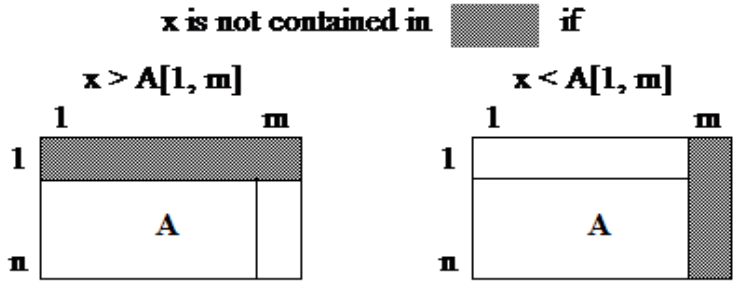


Exhibit 9.2: Another example of the idea of excluded regions.

(b) Implement your algorithm by a

```
function IsInArray (x: integer): boolean;
```

(c) Show that your algorithm is correct and terminates, and determine its worst case time complexity.

## Solution

(a) The algorithm compares x first with A[1, m]. If x is smaller than A[1, m], then x cannot be contained in the last column, and the search process is continued by comparing x with A[1, m – 1]. If x is greater than A[1, m], then x cannot be contained in the first row, and the search process is continued by comparing x with A[2, m]. Exhibit 9.3 shows part of a typical search process.

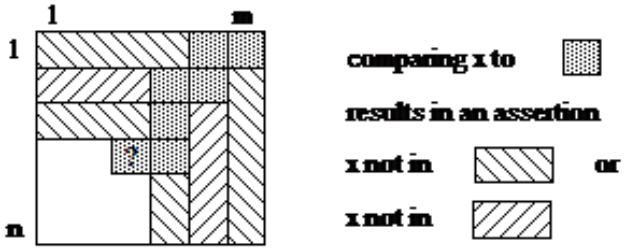


Exhibit 9.3: Excluded regions combine to leave only a staircase-shaped strip to examine.

```
(b)   function IsInArray(x: integer): boolean;
      var  r, c: integer;
      begin
         r := 1;  c := m;
         while  (r ≤ n) and (c ≥ 1)  do
         {1}  if x < A[r, c] then c := c - 1
           elsif  x > A[r, c] then r := r + 1
```

```
      else { x = A[r, c] }  {2}  return(true);
    {3}  return(false)
end;
```

(c) At positions {1}, {2}, and {3}, the invariant

$\forall$ i, 1 ≤ i ≤ n,$\forall$ j, 1 ≤ j ≤ m:

(j > c ⇒ x ≠ A[i, j]) ∧ (i < r ⇒ x ≠ A[i, j] (*)

states that the hatched rows and columns of A do not contain x. At {2},

(1 ≤ r ≤ n) ∧ (1 ≤ c ≤ m) ∧ (x = A[r, c])

states that r and c are within index range and x has been found at (r, c). At {3},

(r = n + 1)∨(c = 0)

states that r or c are outside the index range. This coupled with (*) implies that x is not in A:

(r = n + 1) ∨ (c = 0) ⇒ $\forall$ i, 1 ≤ i ,≤ n, $\forall$ j. 1 ≤ j ≤ m: x ≠ A[i, j].

Each iteration through the loop either decreases c by one or increases r by one. If x is not contained in the array, either c becomes zero or r becomes greater than n after a finite number of steps, and the algorithm terminates. In each step, the algorithm eliminates either a row from the top or a column from the right. In the worst case it works its way from the upper right corner to the lower left corner in n + m – 1 steps, leading to a complexity of Θ(n + m).

## In-place permutation

**Representations of a permutation.** Consider an array D[1 .. n] that holds n data elements of type 'elt'. These are ordered by their position in the array and must be rearranged according to a specific permutation given in another array. Exhibit 9.4 shows an example for n = 5. Assume that a, b, c, d, e, stored in this order, are to be rearranged in the order c, e, d, a, b. This permutation is represented naturally by either of the two permutation arrays t (to) or f (from) declared as

```
var  t, f: array[1 .. n] of 1 .. n;
```

The exhibit also shows a third representation of the same permutation: the decomposition of this permutation into cycles. The element in D[1] moves into D[4], the one in D[4] into D[3], the one in D[3] into D[1], closing a cycle that we abbreviate as (1 4 3), or (4 3 1), or (3 1 4). There is another cycle (2 5), and the entire permutation is represented by (1 4 3) (2 5).

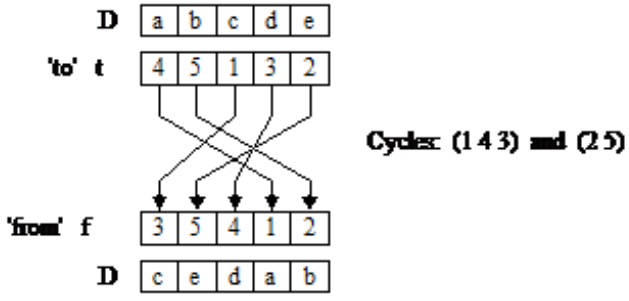


Exhibit 9.4: A permutation and its representations in terms of 'to', 'from', and cycles.

The cycle representation is intuitively most informative, as it directly reflects the decomposition of the problem into independent subproblems, and both the 'to' and 'from' information is easily extracted from it. But 'to' and 'from' dispense with parentheses and lead to more concise programs.

Consider the problem of executing this permutation *in place*: Both the given data and the result are stored in the same array D, and only a (small) constant amount of auxiliary storage may be used, independently of n. Let us use the example of in-place permutation to introduce a notation that is frequently convenient, and to illustrate how the choice of primitive operations affects the solution.

A *multiple assignment* statement will do the job, using either 'to' or 'from':

// (1 ≤ i ≤ n) **{** D[t[i]] := D[i] **}**

or

// (1 ≤ i ≤ n) **{** D[i]} := D[f[i]] **}**

The characteristic properties of a multiple assignment statement are:

- The left-hand side is a sequence of variables, the right-hand side is a sequence of expressions, and the two sequences are matched according to length and type. The value of the i-th expression on the right is assigned to the i-th variable on the left.
- All the expressions on the right-hand side are evaluated using the original values of all variables that occur in them, and the resulting values are assigned "simultaneously" to the variables on the left-hand side. We use the sign // to designate concurrent or parallel execution.

Few of today's programming languages offer multiple assignments, in particular those of variable length used above. Breaking a multiple assignment into single assignments usually forces the programmer to introduce temporary variables. As an example, notice that the direct sequentialization:

```
for i := 1 to n do D[t[i]] := D[i]
```

or

```
for i := 1 to n do D[i] := D[f[i]]
```

is faulty, as some of the elements in D will be overwritten before they can be moved. Overwriting can be avoided at the cost of nearly doubling memory requirements by allocating an array A[1 .. n] of data elements for temporary storage:

```
for i := 1 to n do A[t[i]] := D[i];
for i := 1 to n do D[i] := A[i];
```

This, however, is not an in-place computation, as the amount of auxiliary storage grows with n. It is unnecessarily inefficient: There are elegant in-place permutation algorithms based on the conventional primitive of the single assignment statement. They all assume that the permutation array may be destroyed as the permutation is being executed. If the representation of the permutation must be preserved, additional storage is required for bookkeeping, typically of a size proportional to n. Although this additional space may be as little as n bits, perhaps in order to distinguish the elements processed from those yet to be moved, such an algorithm is not technically in place.

**Nondeterministic algorithms.** Problems of rearrangement always appear to admit many different solutions —a phenomenon that is most apparent when one considers the multitude of sorting algorithms in the literature. The reason is clear: When n elements must be moved, it may not matter much which elements are moved first and which ones later. Thus it is useful to look for *nondeterministic algorithms* that refrain from specifying the precise sequence of all actions taken, and instead merely iterate *condition* ⇒ *action* statements, with the meaning "wherever *condition* applies perform the corresponding *action*". These algorithms are nondeterministic because each of several distinct conditions may apply at lots of different places, and we may "fire" any action that is

currently enabled. Adding sequential control to a nondeterministic algorithm turns it into a deterministic algorithm. Thus a nondeterministic algorithm corresponds to a class of deterministic ones that share common invariants, but differ in the order in which steps are executed. The correctness of a nondeterministic algorithm implies the correctness of all its sequential instances. Thus it is good algorithm design practice to develop a correct nondeterministic algorithm first, then turn it into a deterministic one by ordering execution of its steps with the goal of efficiency in mind.

Deterministic sequential algorithms come in a variety of forms depending on the choice of primitive (assignment or swap), data representation ('to' or 'from'), and technique. We focus on the latter and consider two techniques: cycle rotation and cycle clipping. *Cycle rotation* follows naturally from the idea of decomposing a permutation into cycles and processing one cycle at a time, using temporary storage for a single element. It fits the 'from' representation somewhat more efficiently than the 'to' representation, as the latter requires a swap of two elements where the former uses an assignment. *Cycle clipping* uses the primitive 'swap two elements' so effectively as a step toward executing a permutation that it needs no temporary storage for elements. Because no temporary storage is tied up, it is not necessary to finish processing one cycle before starting on the next one–elements can be clipped from their cycles in any order. Clipping works efficiently with either representation, but is easier to understand with 'to'. We present cycle rotation with 'from' and cycle clipping with 'to' and leave the other two algorithms as exercises.

## Cycle rotation

A search for an in-place algorithm naturally leads to the idea of processing a permutation one cycle at a time: every element we place at its destination bumps another one, but we avoid holding an unbounded number of bumped elements in temporary storage by rotating each cycle, one element at a time. This works best using the 'from' representation. The following loop rotates the cycle that passes through an arbitrary index i:

Rotate the cycle starting at index i, updating f:

```
j := i;{ initialize a two-pronged fork to travel along the cycle }
p := f[j];  { p is j's predecessor in the cycle }
A := D[j];  { save a single element in an auxiliary variable A }
while  p ≠ i  do  { D[j] := D[p];  f[j] := j;  j := p;  p := f[j]} ;
D[j] := A;  { reinsert the saved element into the former cycle … }
f[j] := j;  { … but now it is a fixed point }
```

This code works trivially for a cycle of length 1, where p = f[i] = i guards the body of the loop from ever being executed. The statement f[j] := j in the loop is unnecessary for rotating the cycle. Its purpose is to identify an element that has been placed at its final destination, so this code can be iterated for $1 \le i \le n$ to yield an in-place permutation algorithm. For the sake of efficiency we add two details: (1) We avoid unnecessary movements A := D[j]; D[j] := A of a possibly voluminous element by guarding cycles of length 1 with the test 'i ≠ f[i]', and (2) we terminate the iteration at n – 1 on the grounds that when n – 1 elements of a permutation are in their correct place, the n-th one is also. Using the code above, this leads to

for i := 1 to n – 1 do  if i ≠ f[i] then rotate the cycle starting at index i, updating f

## Exercise

Implement cycle rotation using the 'to' representation. *Hint:* Use the swap primitive rather than element assignment.

## Cycle clipping

Cycle clipping is the key to elegant in-place permutation using the 'to' representation. At each step, we clip an arbitrary element d out of an arbitrary cycle of length > 1, thus reducing the latter's length by 1. As shown in Exhibit 9.5, we place d at its destination, where it forms a cycle of length 1 that needs no further processing. The element it displaces, c, can find a (temporary) home in the cell vacated by d. It is probably out of place there, but no more so than it was at its previous home; its time will come to be relocated to its final destination. Since we have permuted elements, we must update the permutation array to reflect accurately the permutation yet to be performed. This is a local operation in the vicinity of the two elements that were swapped, somewhat like tightening a belt by one notch —all but two of the elements in the clipped cycle remain unaffected. The Exhibit below shows an example. In order to execute the permutation (1 4 3) (2 5), we clip d from its cycle (1 4 3) by placing d at its destination D[3], thus bumping c into the vacant cell D[4]. This amounts to representing the cycle (1 4 3) as a product of two shorter cycles: the swap (3 4), which can be done right away, and the cycle (1 4) to be executed later. The cycle (2 5) remains unaffected. The ovals in Exhibit 9.5 indicate that corresponding entries of D and t are moved together. Exhibit 9.6 shows what happens to a cycle clipped by a swap

```
// { t[i], D[i]  :=:  t[t[i]], D[t[i]] }
```

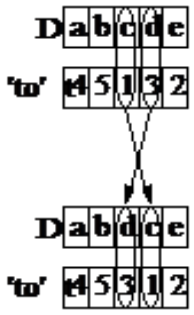


Exhibit 9.5: Clipping one element out of a cycle of a permutation.

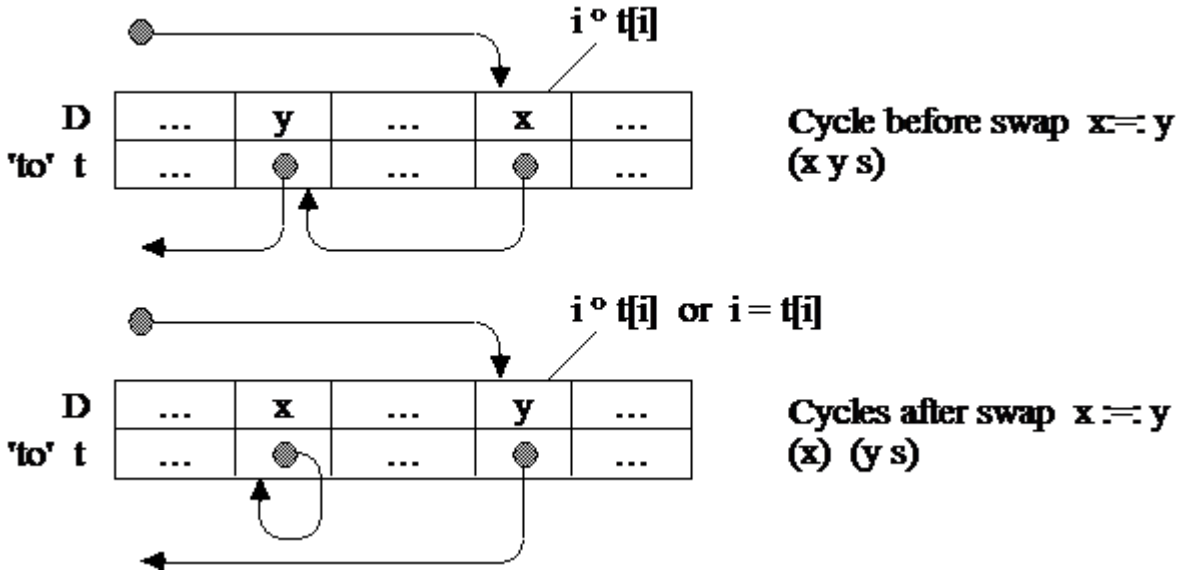


Exhibit 9.6: Effect of a swap caused by the condition i ≠ t[i].

Cycles of length 1 are left alone, and the absence of cycles of length > 1 signals termination. Thus the following *condition* ⇒ *action* statement, iterated as long as the condition i ≠ t[i] can be met, executes a permutation represented in the array t:

```
∃ i:i ≠ t[i]  ⇒  // { t[i], D[i]  :=:  t[t[i]], D[t[i]] }
```

We use the multiple swap operator // { :=: } with the meaning: evaluate all four expressions using the original values of all the variables involved, then perform all four assignments simultaneously. It can be implemented using six single assignments and two auxiliary variables, one of type 1 .. n, the other of type 'elt'. Each swap places (at least) one element into its final position, say j, where it is guarded from any further swaps by virtue of j = t[j]. Thus the nondeterministic algorithm above executes at most n – 1 swaps: When n – 1 elements are in final position, the n-th one is also.

The conditions on i can be checked in any order, as long as they are checked exhaustively, for example:

```
{ (0)  (1 ≤ j < 0) ⇒ j = t[j] }
for  i := 1  to  n - 1  do
   { (1)  (1 ≤ j < i) ⇒ j = t[j] }
   while  i ≠ t[i]  do  // { t[i], D[i]  :=:  t[t[i]], D[t[i]] }
   { (2)  (1 ≤ j ≤ i) ⇒ j = t[j] }
{ (3)  (1 ≤ j ≤ n - 1) ⇒ j = t[j] }
```

For each value of i, i is the leftmost position of the cycle that passes through i. As the while loop reduces this cycle to cycles of length 1, all swaps involve i and t[i] > i, as asserted by the invariant (1) (1 ≤ j < I) ⇒ j = t[j], which precedes the while loop. At completion of the while loop, the assertion is strengthened to include i, as stated in invariant (2) (1 ≤ j ≤ I) ⇒ j = t[j]. This reestablishes (1) for the next higher value of i. The vacuously true assertion (0) serves as the basis of this proof by induction. The final assertion (3) is just a restatement of assertion (2) for the last value of i. Since t[1] ... t[n] is a permutation of 1 ...n, (3) implies that n = t[n].

## Exercise: cycle clipping using the 'from' representation

The nondeterministic algorithm expressed as a multiple assignment

// (1 ≤ i ≤ n) { D[i]} := D[f[i]] }

is equally as valid for the 'from' representation as its analog

// (1 ≤ i ≤ n) { D[t[i]] := D[i] }

was for the 'to' representation. But in contrast to the latter, the former cannot be translated into a simple iteration of the *condition* ⇒ *action* statement:

∃ i: i ≠ f[i] ⇒ // { f[i], D[i] :=: f[f[i]], D[f[i]] }

Why not? Can you salvage the idea of cycle clipping using the 'from' representation

## Exercises

1. Write two functions that implement sequential search, one with sentinel as shown in the first section, "Sequential search" the other without sentinel. Measure and compare their running time on random arrays of various sizes
2. Measure and compare the running times of sequential search and binary search on random arrays of size n, for n = 1 to n = 100. Sequential search is obviously faster for small values of n, and binary search for large n, but where is the crossover? Explain your observations.