INTRODUCTION

The array is a powerful tool that is widely used in computing. Arrays provide for a very special way of storing or organizing data in a computer's memory. The power of the array is largely derived from the fact that it provides us with a very simple and efficient way of referring to and performing computations on collections of data that share some common attribute.

processing of arrays is simplified by using variables to specify suffixes. In processing a particular array it does not matter what name we give the array

suffix—it is only the suffix value which determines which array location is referenced (i.e. if the variables *i* and *j* both have the value 49 then the references *a*[*i*] and *a*[*j*] *both* reference the *same* array location which is *a*[49]).

The concept of a one-dimensional array which we have been considering extends in a natural way to multidimensional arrays.

Arrays play an integral part in many computer algorithms. They simplify the implementation of algorithms that must perform the same computations on collections of data. Furthermore, employment of arrays often leads to implementations that are more efficient than they would otherwise be.

The most important basic ways in which we change the contents of an array location are by direct computation and assignment, by exchange of the contents of two array locations, and by counting. In the algorithms which follow we will examine a variety of basic array techniques and applications.

In more advanced computer applications arrays can be used to build and simulate finite state automata.

Algorithm 4.1
ARRAY ORDER REVERSAL

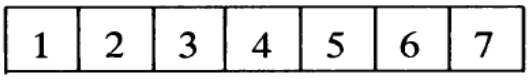
Problem

Rearrange the elements in an array so that they appear in reverse order.

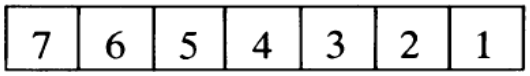
Algorithm development

The problem of reversing the order of an array of numbers appears to be completely straightforward. Whilst this is essentially true, some care and thought must go into implementing the algorithm.

We can start the design of this algorithm by careful examination of the elements of an array before and after it has been reversed; for example,

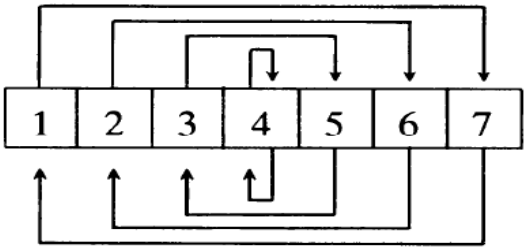


before reversal



after reversal

What we observe from our diagram is that the *first* element ends up in the *last* position. The *second* element ends up in the *second last* position and so on. Carrying this process through we get the following set of exchanges:



array $a[1..n]$

In terms of suffixes the exchanges are:

step [1]	$a[1] <=> a[7]$	
step [2]	$a[2] <=> a[6]$	
step [3]	$a[3] <=> a[5]$	
step [4]	$a[4] <=> a[4]$	there is no exchange here

Examining the effects of these exchanges we discover that after step [3] the array is completely reversed. We see that with each step the suffixes on the left are increasing by one while at the same time the suffixes on the right are decreasing by one. In setting up our algorithm we need a pair of suffixes that model this *increasing–decreasing* behavior. Our increasing suffix can be the variable i which is simply incremented by one with each step.

For our decreasing suffix we might try $[n-i]$ since this decreases by 1 with each increase in i by 1. The problem with this suggestion is that when $i=1$ we find that $[n-i]$ is equal to $n-1$ rather than n as we require for our exchange. We can correct this by adding 1. The suffix $[n-i+1]$ can then be used as our decreasing suffix. With these suffixes we have

i	$n-i+1$
1	$7-1+1=7$
2	$7-2+1=6$
3	$7-3+1=5$
4	$7-4+1=4$

Each exchange (cf. algorithm 2.1) can be achieved by a mechanism of the form

```
t := a[1];
a[i] := a[n-i+1];
a[n-i+1] := t
```

the number of exchanges r to reverse the order of an array is always the nearest integer that is less than or equal half the magnitude of n.

Algorithm description

1. Establish the array $a[1..n]$ of n elements to be reversed.
2. Compute r the number of exchanges needed to reverse the array.
3. While there are still pairs of array elements to be exchanged
   (a) exchange the i^{th} element with the $[n-i+1]^{th}$ element.
4. Return the reversed array.

The algorithm can be suitably implemented as a procedure that accepts as input the array to be reversed and returns as output the reversed array.

Pascal implementation

```
procedure reverse (var a: nelements; n: integer);
var i {increasing index for array},
  r {number of exchanges required}: integer;
  t {temporary variable needed for exchange}: real;

begin {reverses an array of n elements}
  {assert: n>0∧a[1]=a1, a[2]=a2,...,a[n]=a(n)}
  r := n div 2;
  {invariant: 1=<i=<⌊n/2⌋∧a[1]=a(n), a[2]=a(n-1),...,
  a[i]=a(n-i+1),
  a[i+1]=a(i+1), a[n-i]=a(n-i), a[n-i+1]=a(i), ..., a[n]=a1}
  for i := 1 to r do
    begin {exchange next pair}
      t := a[i];
      a[i] := a[n-i+1];
      a[n-i+1] := t
    end
```

Notes on design

1. To reverse an array of n elements $\lfloor n/2 \rfloor$ exchanges are required.
2. There are $r=\lfloor n/2 \rfloor$ pairs of elements in an array of n elements. To reverse an array of n elements r pairs of elements must be exchanged. After the i^{th} iteration (for i in the range $1 \leq i \leq r$) the first i pairs of elements have been interchanged. This relation remains invariant. The i^{th} pair consists of the i^{th} element and the $(n-i+1)^{th}$ element. The algorithm will terminate because with each iteration i is advanced by 1 so eventually r pairs will have been exchanged.

Applications

Vector and matrix processing.

Algorithm 4.2
ARRAY COUNTING OR HISTOGRAMMING

Problem

Given a set of n students' examination marks (in the range 0 to 100) make a count of the number of students that obtained each possible mark.

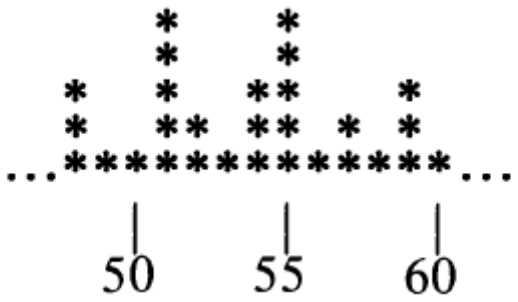
Algorithm development

we are required to do in this case is obtain the distribution of a set of marks. This problem is typical of frequency counting problems. One approach we could take is to set up 101 variables $C0$ $C1$, $C2$, ..., $C100$ each corresponding to a particular mark. The counting strategy we could then employ might be as follows:

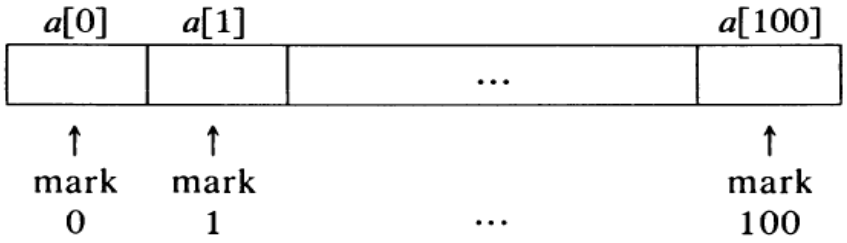
while less than n marks have been examined do
(a) get next mark m,
(b0) if $m = 0$ then $C0 := C0+1$;
(b1) if $m = 1$ then $C1 := C1+1$;
(b2) if $m = 2$ then $C2 := C2+1$;
(b3) if $m = 3$ then $C3 := C3+1$;
.
.
:
(b100) if $m = 100$ then $C100 := C100+1$.

The difficulty with this approach is that we need to make 101 tests (only one of which is successful) just to update the count for one particular mark.

An approach we could take is illustrated by the following diagram:



It involves taking a piece of graph paper or dividing plain paper up into slots each of which can be identified with a particular mark value. The next step is to examine each mark, and, depending on its value, we place a star in the corresponding mark's slot. If we applied this step to all marks when the task was completed the number of stars associated with each slot would represent the mark's count for that particular mark. The method we have outlined is certainly a workable hand solution. In the method we have proposed it is not necessary to compare each mark with all possible marks' values. Instead, the value of a particular mark leads us directly to the particular slot that must be updated. This one-step procedure for each mark would certainly be very attractive if it could be carried across to the computer algorithm. It is at this point that we need to recognize that an array can be usefully employed in the solution to our problem. We can very easily set up an array with 101 locations, each location corresponding to a particular mark value. For example.



If we store in each array location the count of the number of students that obtained that mark we will have the required solution to the problem (e.g. if 15 students obtained the mark 57, then we will have the number 15 in location 57 when all marks have been examined). Initially we can consider what happens when

a particular mark is encountered. Suppose the current mark to be counted is 57. In using the array for counting we must at this stage add one to the count stored in location 57. For this step we can use the actual mark's value (i.e. 57) to reference the array location that we wish to update. That is, the mark's value can be employed as an array suffix. Because it is necessary to add one to the previous count in location 57, we will need a statement of the form:

new count in location 57 := *previous count in location* 57+1

Since location $a[57]$ must play both the "previous count" and "new count" roles, we can write

$$a[57] := a[57]+1$$

or for the general mark m we can write

$$a[m] := a[m]+1$$

Algorithm description

1. Prompt and read in n the number of marks to be processed.
2. Initialize all elements of the counting array $a[0..100]$ to zero.
3. While there are still marks to be processed, repeatedly do
   (a) read next mark m,
   (b) add one to the count in location m in the counting array.
4. Write out the marks frequency count distribution.

Pascal implementation

```
program histogram (input, output);
var i {current number of marks processed},
  m {current mark},
  n {number of marks to be processed}: integer;
  a: array [0..100] of integer;

begin {compute marks frequency distribution}
  writeln ('enter number of marks n on a separate line followed by
  marks');
  readln (n);
  for i := 1 to 100 do a[i] := 0;
  {assert: n >= 0 ∧ all a[0..100] are set to 0}
  {invariant: when i marks read, for j in range 0 =< j =< 100, all a[j]
  will represent the number of marks j in the first i read ∧ i =< n}
  for i := 1 to n do
    begin {read next mark and update appropriate array elements}
      read (m);
      if eoln (input) then readln;
      {assert: m in range 0 =< m =< 100}
      a[m] := a[m] + 1
    end;
    {assert: when n marks read, for j in range 0 =< j =< 100, all a[j]
    will represent the number of marks j in the set}
      for i := 0 to 100 do
        begin
          write (a[i]);
          if i mod 8 = 0 then writeln
        end
```

Notes on design

1. Essentially n steps are required to generate the frequency histogram for a set of n marks.

2. After i iterations the nth element in the a array will contain an integer representing the number of marks j encountered in the first i marks. This relation holds for all j in the range 0<j-c-100 and for all i in the range 1-s:i.s.n. On termination, when i= n, all array elements will reflect the appropriate marks' counts for the complete set. It follows from the definition of the for-loop that both loops terminate.

3. The idea of indexing by value is important in many algorithms because of its efficiency.

Applications

Statistical analyses.

Algorithm 4.3
FINDING THE MAXIMUM NUMBER IN A SET

Problem

Find the maximum number in a set of n numbers.

Algorithm development

The maximum is that number which is greater than or equal to all other numbers in the set. This definition accommodates the fact that the maximum may not be unique. It also implies that the maximum is only defined for sets of one or more elements. For example,

8	6	5	15	7	19	21	6	13

After studying this example we can conclude that *all* numbers need to be examined to establish the maximum. A second conclusion is that comparison of the relative magnitude of numbers must be made.

When the first number appears on the screen we have no way of knowing whether or not it is the maximum. In this situation the best that we can do is write it down as our temporary candidate for the maximum. Having made the decision to write down the first number we must now decide what to do when the second number appears

on the screen. Three situations are possible:

1. the second number can be less than our temporary candidate for the maximum;

2. the second number can be equal to our temporary candidate for the maximum;

3. the second number can be greater than our temporary candidate for the maximum.

If situations (1) or (2) apply our temporary candidate for the maximum is still valid and so there is no need to change it. In these circumstances we can simply go ahead and compare the third number with our temporary maximum which we will call max. However, if the second number is greater than our temporary max-imum, we must cross out our original temporary maximum and write down the second number as the new temporary maximum. We then move on and compare the third number with the new temporary maximum. The whole process will need to continue until all elements in the set have been examined. As larger values are encountered they assume the role of the temporary maximum. At the time when all numbers have been examined the temporary maximum that is written down is the maximum for the complete set. This strategy can form the basis of our computer algorithm.

Our initial proposal might therefore be:

1. While all array elements not examined do
   (a) if the current array element > temporary maximum then update the temporary maximum.

Algorithm description

1. Establish an array a[1.. n] of n elements where n>= 1

2. Set temporary maximum max to first array element.

3. While less than n array elements have been considered do

(a) if next element greater than current maximum max then assign it to max.

4. Return maximum max for the array of n elements.

Pascal implementation

```
function amax (a: nelements; n: integer): real;
var i {array index}: integer;
  max {current maximum}: real;

begin {find the maximum in an array of n numbers}
  {assert: n>0}
  i := 1;
  max := a[i];
  {invariant: max is maximum in a[1..i]∧i=<n}
  for i := 2 to n do
    if a[i]>max then max := a[i];
  {assert: max is maximum in a[1..n]}
  amax := max
end
```

Notes on design

1. The number of comparisons needed to find the maximum in an array of n elements is n-1.

2. For all i in the range 1 ≤ i≤ n when i elements have been examined the variable max is greater than or equal to all elements in the range 1 to i.

Applications

Plotting, scaling, sorting.

Algorithm 4.4
REMOVAL OF DUPLICATES FROM AN ORDERED ARRAY

Problem

Remove all duplicates from an ordered array and contract the array accordingly.

As a starting point for this design let us focus on a specific example so that we have a clear idea of exactly what is required.

1	2	3	4	5	6	7	8	9	10	11	12	13
2	2	8	15	23	23	23	23	26	29	30	32	32

Before duplicate removal

. . . .

After a brief examination of the array we will be able to produce the contracted array below:

1	2	3	4	5	6	7	8
2	8	15	23	26	29	30	32

After duplicate removal

Comparing the two arrays we see that all elements, apart from the first, have shifted their positions in the array. In other words, each *unique* element in the original array has been moved as far to the left as possible.

Whatever mechanism we finally decide upon is going to need to be built around the detection of duplicates in the original data. A duplicate pair is identified when two adjacent elements are equal in value. With each comparison, only two situations are possible:

1. a pair of duplicates has been encountered;
2. the two elements are different.

Study of our example reveals that the position in the array where each most recently encountered unique element must be located is determined at each instance by the number of unique elements met so far. In the case of the 26, it is the fifth unique element and so it must accordingly be placed in position 5. This suggests the use of a counter, the value of which at each instance reflects the number of unique elements encountered to date. If i is the position where the most recently encountered unique element is found and j is the count of the number of unique elements to date, then an assignment of the form:

$$a[j] := a[i]$$

Summarizing the basic steps so far in our mechanism we have:

while all adjacent pairs of elements have not been compared do
(a) if they are not equal, shift the rightmost element in the next pair to the array position determined by the current unique element count.

When we consider how the algorithm must terminate,

$i := 2$
$a[i-1] = a[i]?$ (comparison)

is better because it allows the algorithm to terminate directly when i is equal to n, the number of elements in the original array.

We may have noticed in exploring the problem that if there are no duplicates in the array, then the repeated assignment

$$a[j] := a[i]$$

is unnecessary because all elements are already in their correct place. Even if duplicates are present in an array it is only necessary to start shifting elements *after* the first duplicate is encountered.

One way is to compare pairs of elements until a duplicate is encountered. For this we can use a loop of the form

while $a[i-1] <> a[i]$ **do** $i := i+1$

When a duplicate is encountered, the unique element count will be $i-1$. The variable j should be set to this value.

Algorithm description

1. Establish the array $a[1..n]$ of n elements.
2. Set loop index i to 2 to allow correct termination.
3. Compare successive pairs of elements until a duplicate is encountered then set unique element count j.
4. While all pairs have not been examined do
   (a) if next pair not duplicates then
      (a.1) add one to unique element count j,
      (a.2) move later element of pair to array position determined by the unique element count j.

Pascal implementation

```
procedure duplicates (var a: nelements; var n: integer);
var i {at all times i-1 is equal to the number of pairs examined},
  j {current count of the number of unique elements encountered}:
  integer;

begin {deletes duplicates from an ordered array}
  {assert: n>1∧elements a[1..n]in non-descending order}
  i := 2;
  while (a[i-1]<>a[i]) and (i<n) do i := i+1;
  if a[i-1]<>a[i] then i := i+1;
  {assert: i>=2∧a[1..i-1]unique∧in ascending order}
  j := i-1;
  {invariant: after the ith iteration j<=i-1∧i=<n+1∧there are no
  equal adjacent pairs in the set a[1..j]}
  while i<n do
    begin {examine next pair}
      i := i+1;
      if a[i-1]<>a[i] then
        begin {shift latest unique element to unique count position}
          j := j+1;
          a[j] := a[i]
        end
    end;
```

Notes on design

1. To delete duplicates from an array of n elements $(n-1)$ comparisons are required. The number of data movements (i.e. $a[j] := a[i]$ instructions) required is at best 0 and at worst $(n-2)$. In general it will be somewhere between these two extremes.
2. At the end of each iteration, the variable j represents the number of unique elements encountered in examining the first $(i-1)$ pairs of elements. The j unique elements encountered are located in the first j locations of the array. On termination when $i = n$ or $n+1$ the value of j will represent the number of unique elements in the original input data. The algorithm will terminate because the variable i advances by one towards n with each iteration. The algorithm will function correctly for values of $n>1$. In the case when all elements are unique the second **while**-loop is not entered.

Applications

Data compression and text processing problems.

Supplementary problems

4.4.1 Remove from an ordered array *all* numbers that occur more than once.

4.4.2 Delete from an ordered array all elements that occur more than k times.

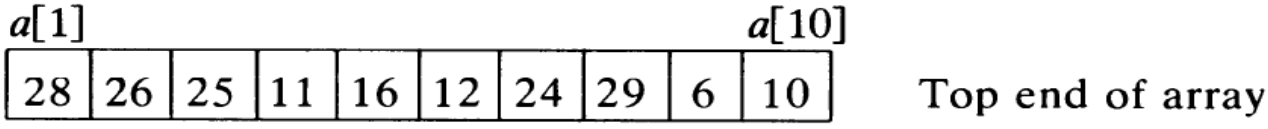
Algorithm 4.5
PARTITIONING AN ARRAY

Problem

Given a randomly ordered array of n elements, partition the elements into two subsets such that elements $\leqslant x$ are in one subset and elements $>x$ are in the other subset.

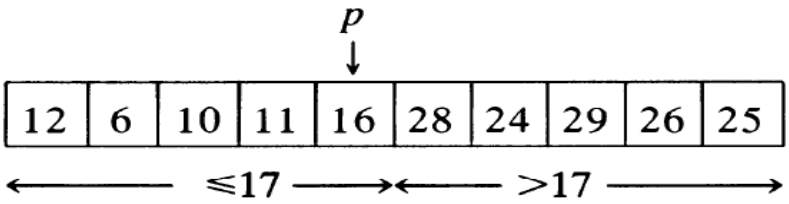
Algorithm development

This problem is relevant to some sorting and median finding algorithms. To try to focus on what must be done, we can consider a particular example. Given the random data set below, we are asked to partition it into two subsets, one containing elements $\leqslant 17$ and the other containing elements >17.

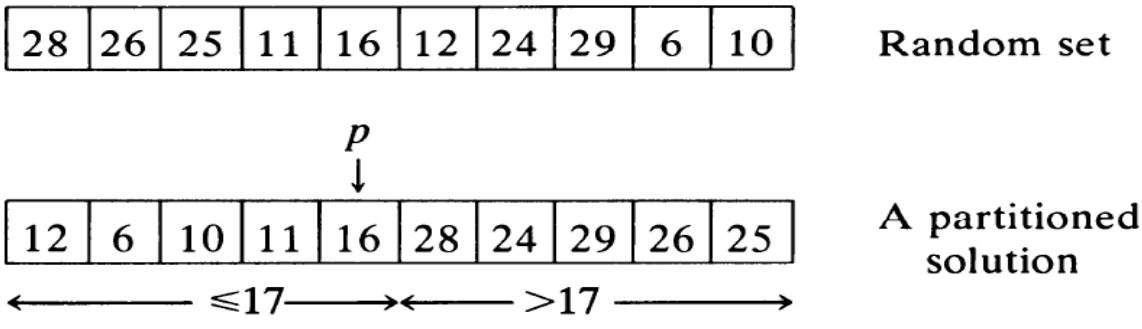


Clearly we need to be able to separate the two subsets. To do this, we could put those elements >17 at the top end (the high suffix end) of the arrav and those $\leqslant 17$ at the bottom of the arrav.

With this solution to the problem, we have actually *ordered* the two subsets in addition to separating them. In our original statement of the problem it was *not* required that the elements be ordered. For our example the configuration below:

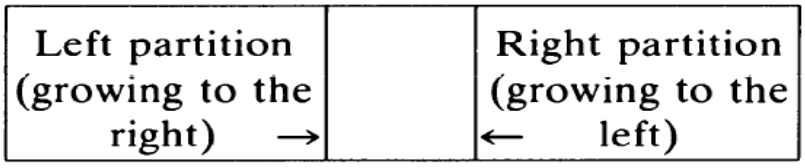


would equally well have satisfied the requirements of the problem. Notice in this data set that while the data is still partitioned into two subsets the elements are no longer ordered. Sorting of data is usually a costly operation.

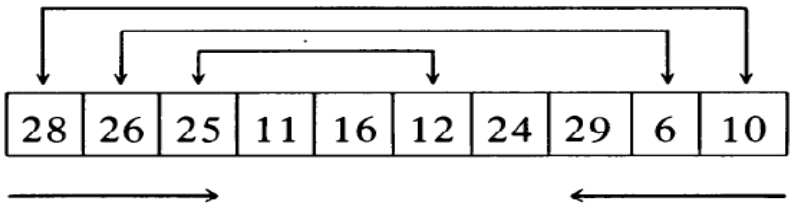


Comparing these two data sets, we see that elements at the left-hand end >17 must be moved to the right-hand end of the partitioning point p.

Elements $\leqslant 17$ that are to the left of p need not be moved because they are already in their proper partition. This saving will result in fewer exchanges being required than in the sort method. When we are initially presented with the random set we do not know how many elements are $\leqslant 17$. One way to overcome this would be to make a pass through the array counting all values $\leqslant 17$. Once we know the value of p we can then make another pass through the array. This time when we encounter a value >17 on the left side of p we must move it to the right of p.

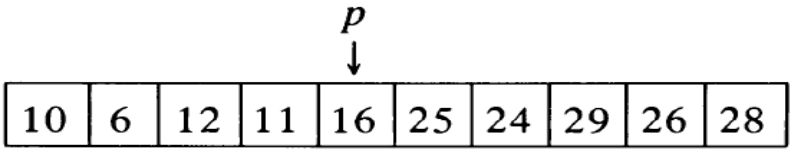


If we continue this "pincer" process, the two partitions will eventually meet and when they do we will have the desired partition for our complete data set. Working completely through our example above, we get:



Random set

With these exchanges we end up with the partitioned set below:



It can be seen that this approach enables us to partition the data by making *just one* pass rather than two passes through the array. We might therefore propose the following basic partitioning mechanism:

while the two partitions have not met do

(a) extend the left and right partitions inwards exchanging any wrongly placed pairs in the process.

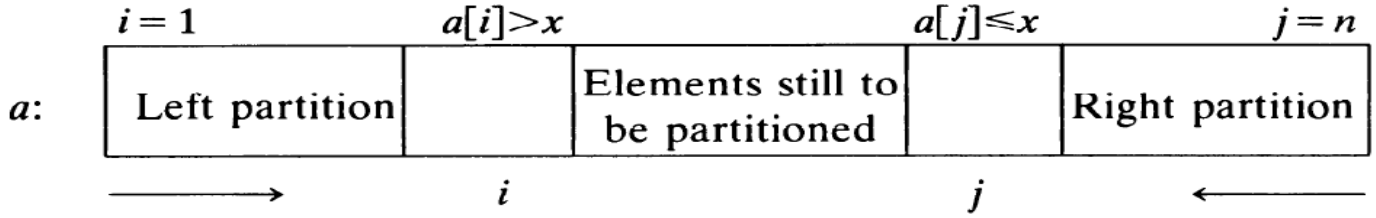
The first consideration is to model the "moving inwards" process. Movement inwards from the left can proceed until we encounter an element larger than the partitioning value x (in the above example $x = 17$). This can be accomplished by a loop of the form:

$$\textbf{while } a[i] \leqslant x \textbf{ do } i := i+1$$

Movement inwards from the right can proceed until we encounter an element smaller than or equal to x; for this we can use a decreasing loop:

$$\textbf{while } a[j] > x \textbf{ do } j := j-1$$

The starting value for i must be 1 and the starting value for j must be n, the number of elements in the array. As soon as both loops have terminated we have detected an element at position i, that must be moved to the right partition, and an element at j, that must be moved to the left partition.



At this point the i^{th} and j^{th} elements can be exchanged. For the exchange we can use the standard technique:

$$\begin{aligned} t &:= a[i]; \\ a[i] &:= a[j]; \\ a[j] &:= t \end{aligned}$$

After the exchange we can start the "moving inwards" process again at positions $(i+1)$ and $(j-1)$.

The only other implementation considerations we need to make involve termination of the loops. Only when the two partitions meet do we need to terminate the "pincer" process. That is, the main loop should progress only while the i index is less than the j index. That is,

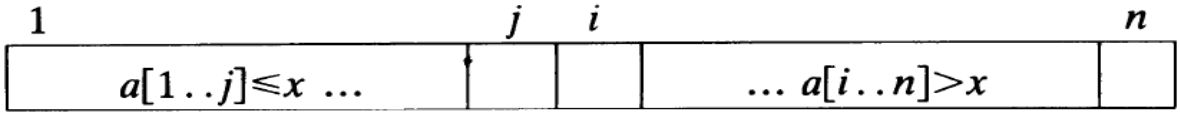
while $i<j$ do

(a) move i and j towards each other exchanging any wrongly placed pairs in the process.

Incorporating the ideas for moving inwards and exchanging wrongly placed pairs we get:

```
while i<j do
  begin
    while a[i]≤x do i := i+1;
    while a[j]>x do j := j−1;
    t := a[i];
    a[i] := a[j];
    a[j] := t;
    i := i+1;
    j := j−1
  end
```

With this loop structure we see that because of the way i and j are changed after each exchange, they can cross over when the two partitions meet. It follows that we can end up with the configuration shown below:



The index j will therefore indicate the upper limit for the left partition.

Algorithm description

1. Establish the array $a[1..n]$ and the partitioning value x.
2. Move the two partitions towards each other until a wrongly placed pair of elements is encountered. Allow for special cases of x being outside the range of array values.
3. While the two partitions have not met or crossed over do
   (a) exchange the wrongly partitioned pair and extend both partitions inwards by one element;
   (b) extend left partition while elements less than or equal to x;
   (c) extend the right partition while elements are greater than x.
4. Return the partitioning index p and the partitioned array.

Pascal implementation

```
procedure xpartition (var a: nelements; n: integer; var p: integer; x:
real);
var i {current upper boundary for values in partition =<x},
  j {current lower boundary for values in partition >x}: integer;
  t {temporary variable for exchange}: real;

begin {partition array into two subsets (1) elements =<x
(2) elements>x}
  {assert: n>0}
  i := 1; j := n;
  while (i<j) and (a[i]<=x) do i := i+1;
  while (i<j) and (a[j]>x) do j := j-1;
  if a[j]>x then j := j-1;
  {invariant: after the ith iteration a[1..i-1]=<x ∧a[j+1..n]>x
  ∧i=<n+1∧j>=0∧i=<j+1}
  while i<j do
    begin {exchange current pair that are in wrong positions}
      t := a[i];
      a[i] := a[j];
      a[j] := t;
      {move inwards past the two exchanged values}
      i := i+1;
      j := j-1;

      {extend lower partition}
      while a[i]<=x do i := i+1;
      {extend upper partition}
      while a[j]>x do j := j-1
    end;
  {assert: a [1..i-1]=<x∧a[j+1..n]>x∧i>j}
  p := j
end
```

Notes on design

1. To partition an array into two subsets at most $(n+2)$ comparisons of the form $a[i] \leqslant x$ and $a[j]>x$ must be made. The number of exchanges required can vary between 0 and $\lfloor n/2 \rfloor$ depending on the distribution of the array elements. If a sorting method had been used for partitioning,

of the order of $n\log_2 n$ (written $O(n\log_2 n)$) comparisons would have been required.

2. After each iteration the first $(i-1)$ elements in the array are $\leq x$ and the last $(n-j)$ elements are $>x$. The variables i and j are increased and decreased respectively in such a way that these two relations hold. The outer **while**-loop will terminate because with each iteration the distance between i and j is decreased by at least 2 because the statements $i := i+1$ and $j := 1$ are executed at least once.

Applications

Sorting, statistical classification.

Algorithm 4.6
FINDING THE k^{th} SMALLEST ELEMENT

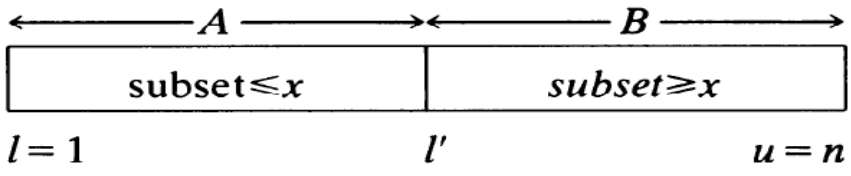
Problem

Given a randomly ordered array of n elements determine the k^{th} smalles element in the set.

Problem

Given a randomly ordered array of n elements determine the k^{th} smallest element in the set.

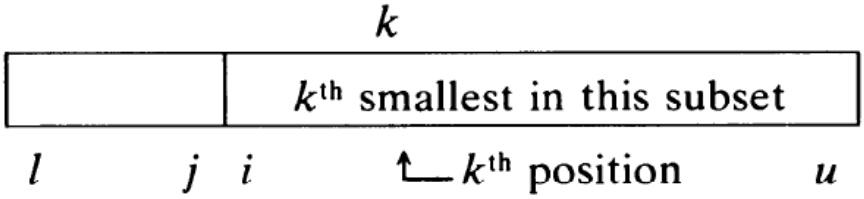
Since we have no idea in advance what the k^{th} smallest value is, we might therefore be tempted to choose a value x at random from the array and partition the array about x. The variables l and u are initially assigned to the bounds of the array. For example,



However, the value x will cause the original data set to be divided into two *smaller* subsets. The k^{th} smallest value will have to be in one or the other of these two subsets A and B.

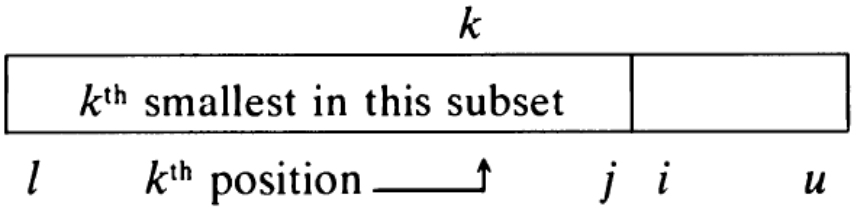
However, the value x will cause the original data set to be divided into two *smaller* subsets. The k^{th} smallest value will have to be in one or the other of these two subsets A and B. If, for example, the k^{th} smallest value is in subset B (i.e. in the subset with elements $\geqslant x$) then we can completely disregard subset A and start trying to find the k^{th} smallest value in subset B. The easiest way to do this is to replace l by l' and start searching for the k^{th} smallest value again in the smaller subset B. If we repeatedly apply this partitioning process to smaller and smaller subsets that contain the k^{th} smallest value we will eventually obtain the desired result.

1. k^{th} smallest in subset $\geqslant x$:



Here we set $l := i$ and repeat the partitioning process for the new limits of l and u.

2. k^{th} smallest in subset $\leqslant x$:



Here we set $u := j$ and repeat the partitioning process for the new limits of u and l. By examining the values of i and j on termination of the partitioning loop we know which subset contains the k^{th} smallest value and hence which limit must be updated. The two tests we can use are

if $j<k$ **then** $l := i$;
if $i>k$ **then** $u := j$

The partitioning process need only continue while $l<u$. The variables i and j will need to be reset to the adjusted limits l and u before beginning each new partitioning phase.

The basic mechanism may therefore take the form:

while $l<u$ do
(a) choose some value x about which to partition the array,
(b) partition the array into two partitions marked by i and j,
(c) update limits using the tests
if $j<k$ **then** $l := i$
if $i>k$ **then** $u := j$

The difference we may anticipate at this stage is that x will be selected using

$$x := a[k]$$

Algorithm description

1. Establish $a[1..n]$ and the requirement that the k^{th} smallest element is sought.
2. While the left and right partitions do not overlap do
   (a) choose $a[k]$ as the current partitioning value x;
   (b) set i to the upper limit l of the left partition;
   (c) set j to the lower limit u of the right partition;
   (d) while i has not advanced beyond k and j is greater than or equal to k do

(d.1) extend the left partition while $a[i]<x$;
(d.2) extend the right partition while $x<a[j]$;
(d.3) exchange $a[i]$ with $a[j]$;
(d.4) extend i by 1 and reduce j by 1;

(e) if k^{th} smallest in left partition, update upper limit u of left partition;

(f) if k^{th} smallest in right partition, update lower limit l of right partition.

3. Return the partitioned array with elements $\leq a[k]$ in the first k positions in the array.

Pascal implementation

```
procedure kselect (var a: nelements; k, n: integer);
var i {temporary extension of left partition for current guess at kth
smallest element x},
   j {temporary extension of right partition for current guess at kth
   smallest element x},
   l {upper limit for left partition},
   u {lower limit for right partition} integer;
   x {current guess at kth element in array},
   t {temporary variable used to exchange a[i] with a[j] }: real;

begin {finds kth smallest element in array a. on termination kth
smallest is in position k}

{assert: n>0∧1=<k=<n}
l := 1;
u := n;
{invariant: all a[1..l-1]=<all a[u+1..n]∧l=<k+1∧k-1=<u}
while l<u do
  begin {using new estimate of kth smallest x try to extend left
  and right partitions}
  i := l;
  j := u;
  x := a[k];
  {invariant: all a[l..i-1]=<all a[j+1..u]∧i=<k+1∧j>=k-1
  ∧1=<k=<n}
  while (i<=k) and (j>=k) do
    begin {extend left and right partitions as far as possible, then
    exchange}
```

```
            while a[i]<x do i := i+1;
            while x<a[j] do j := j-1;
            t := a[i];
            a[i] := a[j];
            a[j] := t;
            i := i+1;
            j := j-1
          end;
        {update limits of left and right positions as required}
        if j<k then l := i;
        if i>k then u := j
        end
        {assert: all a[1..k]=<all a[k..n]}
end
```