MERGING, SORTING AND SEARCHING

Sorting alone has been said to account for more than 30% of all computer time spent. Sorting and merging provide us with a means of organizing information to facilitate the retrieval of specific data.

Sorting refers to ordering data in a set to a predefined ordering relation i.e either increasing or decreasing fashion according to some linear relationship among the data items.

The two most common types of data are string information and numerical information. Sorting can be done on names, numbers and records. The ordering relation for numeric data simply involves arranging items in sequence from smallest to largest (or vice versa) such that each item is less than or equal to its immediate successor. This ordering is referred to as non-descending order. The items in the set below have been arranged in non-descending numeric order.

{7, 11, 13, 16, 16, 19, 23}

Sorted string information is generally arranged in standard lexicographical or dictionary order. The following list has been arranged in dictionary order

{a, abacus, above, be, become, beyond}

For example, it is relatively easy to look up the phone number of a friend from a telephone dictionary because the names in the phone book have been sorted into alphabetical order.

Searching methods are designed to take advantage of the organization of information and thereby reduce the amount of effort to either locate a particular item or to establish that it is not present in a data set.

Sorting algorithms usually fall into one of two classes:

1. The simpler and less sophisticated algorithms are characterized by the fact that they require of the order of n2 comparisons (i.e. 0(n2)) to sort n items.

2. The advanced sorting algorithms take of the order of n log2 n (i.e. 0(n log 2 n)) comparisons to sort n items of data. Algorithms within this set come close to the optimum possible performance for sorting random data.

The advanced methods gain their superiority because of their ability to exchange values over large distances in the early stages of the sort. It can be shown that, on average, for random data, items need to be moved a distance of about n/3. The simpler and less efficient methods tend to only move items over small distances and consequently they end up having to make many more moves before the final ordering is achieved.

No one sorting method is best for all applications. Performances of the various methods depend on parameters like the size of the data set, the degree of relative order already present in the data, the distribution of values of the items, and the amount of information associated with each item. For example, if the data is almost in sorted order, the bubblesort (which is normally the least efficient for random data) can give better performance than one of the advanced methods.

Algorithm 5.1
THE TWO-WAY MERGE

Problem

Merge two arrays of integers, both with their elements in ascending order, into a single ordered array.

Algorithm development

Merging two or more sets of data is a task that is frequently performed in computing. It is simpler than sorting because it is possible to take advantage of the partial order in the data.

Consider the two arrays:

a:	15	18	42	51					*m* elements
b:	8	11	16	17	44	58	71	74	*n* elements

A little thought reveals that the merged result should be as indicated below. The origins (array a or b) are written above each element in the c array.

	b	*b*	*a*	*b*	*b*	*a*	*a*	*b*	*a*	*b*	*b*	*b*	
c:	8	11	15	16	17	18	42	44	51	58	71	74	$(n+m)$ elements

c is longer than either a or b. In fact c must contain a number of elements corresponding to the sum of the elements in a and b (i.e. n+m).

To maintain order in placing elements into c it will be necessary to make comparisons in some way between the elements in a and the elements in b.

To merge the two one-element arrays all we need to do is select the smaller of the a and b elements arid place it in c. The larger element is then placed into c. Consider the example below:

a: | 15 |

b: | 8 |

The 8 is less than 15 and so it must go into c[1] first. The 15 is then placed in c[2] to give:

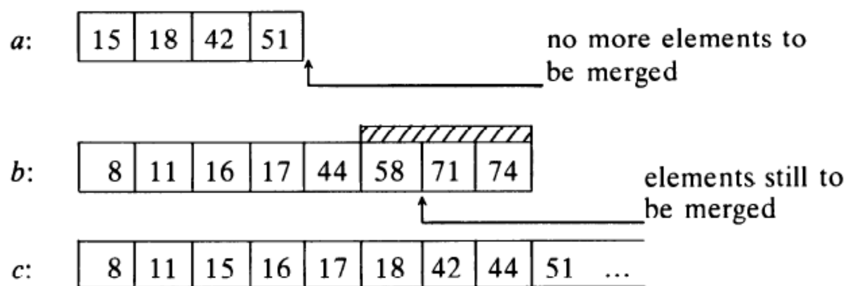
c: | 8 | 15 |

As an element is selected from either a or b the appropriate pointer must be incremented by 1. This ensures that i and j are always kept pointing at the respective first elements of the yet-to-be-merged parts of both arrays. The only other pointer needed is one that keeps track of the number of elements placed in the merged array to date. This pointer, denoted k, is simply incremented by 1 with each new element added to the c array.

One approach we can take with this problem is to include tests to detect when either array runs out. As soon as this phase of the merge is completed another mechanism takes over which copies the yet-to-be-merged elements into the c array. An overall structure we could use is:

1. while ($i \leq m$) and ($j \leq n$) do
   (a) compare $a[i]$ and $b[j]$ then merge the smaller member of the pair into c array,
   (b) update appropriate pointers,

2. if $i<m$ then
   (a) copy rest of a array into c,
   else
   (a′) copy rest of b array into c.



State of merge after *a* array exhausted.

c=

8	11	15	16	17	18	42	44	51	58	71	74

ALGORITHM DESCRIPTION

procedure merge

1. Establish the arrays a[1.. m] and b[1.. n].
2. If last a element less than or equal to last b element then
(a) merge all of a with b,
 (b) copy rest of b,
else
(a') merge all of b with a,
 (b') copy rest of a.
3. Return the merged result c[1..n+m].

procedure *copy*

1. Establish the arrays $b[1..n]$ and $c[1..n+m]$
   and also establish where copying is to begin in b (i.e. at j)
   and where copying is to begin in c (i.e. at k).
2. While the end of the b array is still not reached do
   (a) copy element from current position in b to current position in c;
   (b) advance pointer j to next position in b;
   (c) advance pointer k to next position in c.

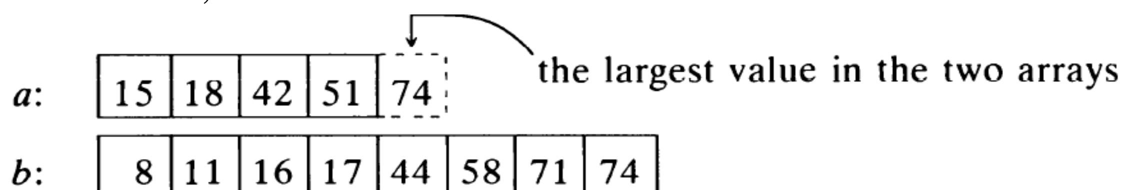
Pascal implementation

```
procedure merge(var a,b: nelements; var c: npmelements; m,n:
integer);

begin {merges the arrays a[1..m] and b[1..n] to give c[1..m+n]
taking advantage of which array is used up first in merge}
  if a[m]<=b[n] then
    mergecopy(a,b,c,m,n)
  else
    mergecopy(b,a,c,n,m)
end
```

Pascal implementation

```
procedure copy(var b: nelements; var c: npmelements; j,n: integer;
var k: integer);
var i {index for section of b array to be copied}: integer;

begin {copy sequence b[j..n] into merged output}
  {assert: k=k0}
{invariant: c[k0]=b[j]∧c[k0+1]=b[j+1]∧....c[k-1]=b[i]∧1=<k0
∧ 1=<i=<n ∧ 1=<j=<n ∧ k0=<k=<m+n+1}
for i := j to n do
begin
  c[k] := b[i];
  k := k+1
end
{assert: c[k0]=b[j]∧c[k0+1]=b[j+1]...∧c[k-1]=b[n]}
end;
```

A place to start on this new line of attack is with the segment that had to be copied in our example. We see that the largest value from the a and b arrays ends up as the last value to be merged into the c array. If for a moment we imagined that the largest value on the end of the a array were a 74 then the merge would progress in such a way that completion of c would occur when the ends of both a and b were reached. That is,



The largest element in the two arrays is present on the ends of both arrays then the last two elements to be merged must be the last element in the a array and the last element in the b array. With this situation guaranteed we no longer have to worry about which array runs out first. We simply continue the merging process until (n+m) elements have been inserted in the c array.

procedure *shortmerge*

1. Establish the arrays $a[1..m]$ and $b[1..n]$ with $a[m] \leq b[n]$.
2. While all of *a* array still not merged do
   (a) if current *a* element less than or equal to current *b* element then
      (a.1) copy current *a* into the current *c* position,
      (a.2) advance pointer to next position in *a* array,
      else
      (a′.1) copy current *b* into the current *c* position,
      (a′.2) advance pointer to next position in *b* array;
   (b) advance pointer for *c* array by one.

```
procedure shortmerge(var a,b: nelements; var c: npmelements; m:
integer; var j,k: integer);
var i {index for the a array that is being merged}: integer;

begin {merges all of a array with b array elements <a[m]}
  {assert: m>0∧n>0∧a[1..m] ordered∧b[1..n] ordered}
  i := 1;
  {invariant: after ith iteration a[1..i−1] merged with b[1..j−1] and
  stored in c[1..k−1] ordered ∧i=<m+1 ∧j=<n+1 ∧
  k=<m+n+1}
  while i<=m do
    begin
      if a[i]<=b[j] then
        begin
          c[k] := a[i];
          i := i+1
        end
      else
      begin
        c[k] := b[j];
        j := j+1
      end;
      k := k+1
    end
    {assert: c[1..k −1] ordered ∧ it is made up of only a[1..m] and
    b[1..j −1]  elements ∧k=<m+n+1 ∧ j=<n+1}
end
```

A single procedure mergecopy can be used to implement these merging and copying steps. The merge and the copying operations can also be implemented as separate procedures. In the merging process it is possible that the two arrays do not overlap. When this happens the a and b data sets should be copied one after the other. After determining which array finishes merging first it is then a simple matter to determine if there is overlap between the two arrays. A comparison of the last element of the array that finishes merging first with the first element of the other array will establish whether or not there is overlap. For example if a ends first we can use:

procedure *mergecopy*

1. Establish the arrays $a[1..m]$ and $b[1..n]$ with $a[m] \leq b[n]$.
2. If last element of a less than or equal to first element of b then
   (a) copy all of a into first m elements of c,
   (b) copy all of b into c starting at $m+1$,
   else
   (a′) merge all of a with b into c,
   (b′) copy rest of b into c starting at position just past where merge finished.

```
procedure mergecopy(var a,b: nelements; var c: npmelements; m,n:
integer);
var i {first position in a array},
  j {current position in b array},
  k {current position in merged array – initially 1}: integer;

begin {merges a[1..m] with b[1..n]}
  {assert: m>0∧n>0∧a[1..m] ordered∧b[1..n] ordered}
  i := 1; j := 1; k := 1;
  if a[m]<=b[i] then
    begin {two sequences do not overlap so copy instead of merge}
      copy(a,c,i,m,k);
      copy(b,c,j,n,k)
    end

  else
    begin {merge all of a with b then copy rest of b}
      shortmerge(a,b,c,m,j,k);
      copy(b,c,j,n,k)
    end
  {assert: c[1..m+n] ordered∧it is made up of only a[1..m] and
  b[1..n] elements}
end
```

Applications

Sorting, tape sorting, data processing.

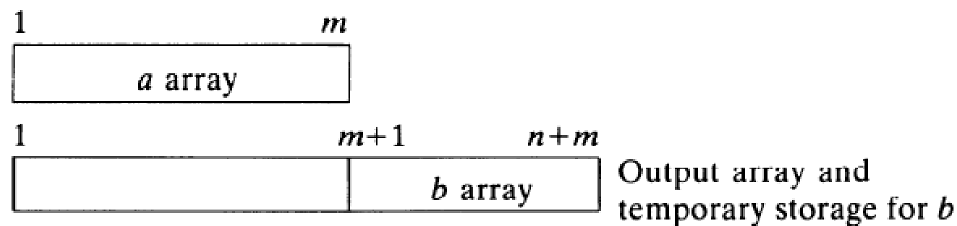
Supplementary Problems

5.1.1 Implement the first merging algorithm that was developed.

5.1.2 Design and implement a merging algorithm that reads the data sets from two files of unknown length. Use end-of-file tests to detect the ends of the data sets.

5.1.3 Design and implement a merging algorithm that uses only two arrays. It can be assumed that the sizes of the two data sets are known in advance. An interesting way to do this is to place the array with the

biggest element so that it fills up the output (merged) array. The following diagram illustrates the idea (the b array has the largest element).



This simplifies the merge because when the merging of a is completed the remaining elements of b will be in place.

5.1.4 Design an algorithm for merging three arrays.

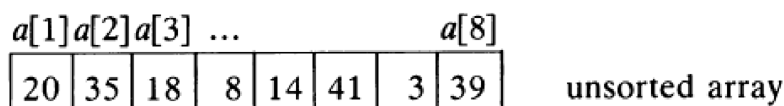
SORTING BY SELECTION

Problem

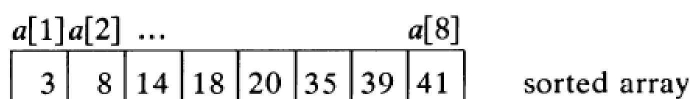
Given a randomly ordered set of n integers, sort them into non-descending order using the selection sort.

Algorithm development

An important idea in sorting of data is to use a selection method to achieve the desired ordering. In its simplest form at each stage in the ordering process, the next smallest value must be found and placed in order. Consider the unsorted array:

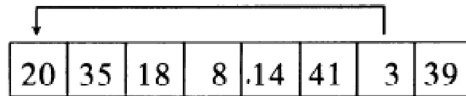


What we are attempting to do is to develop a mechanism that converts the unsorted array to the ordered configuration below:



Comparing the sorted and unsorted arrays we see that one way to start off the sorting process would be to perform the following two steps:

1. Find the smallest element in the unsorted array;
2. Place the smallest element in position a[1].



The following construct can be used to find the smallest element:

```
min := a[1];
for j := 2 to n do
  if a[j]<min then min := a[j]
```

Then the assignment below can be used to put the minimum in position $a[1]$.

$$a[1] := min$$

To achieve these two changes we need a mechanism that not only finds the minimum but also remembers the array location where the minimum is currently stored. That is every time the minimum is updated we must save its position.

This step can be added to our previous code:

```
min := a[1];
p := 1;
for j := 2 to n do
  if a[j]<min then
    begin
      min := a[j];
      p := j
    end
```

Algorithm description

1. Establish the array $a[1..n]$ of n elements.
2. While there are still elements in the unsorted part of the array do
   (a) find the minimum *min* and its location p in the unsorted part of the array $a[i..n]$;
   (b) exchange the minimum *min* in the unsorted part of the array with the first element $a[i]$ in the unsorted array.

Pascal implementation

```
procedure selectionsort(var a: nelements; n: integer);
var i {first element in unsorted part of array},
  j {index for unsorted part of array},
  p {position of minimum in unsorted part of array},
```

```
  min {current minimum in unsorted part of array}: integer;

begin {sorts array a[1..n] into non-descending order using selection
method}
  {assert: n>0∧i=0}
  {invariant: a[1..i] sorted∧all a[1..i]=<all a[i+1..n]}
  for i := 1 to n−1 do
    begin {find minimum in unsorted part of array and exchange it
    with a[i]}
      {assert: 1=<i=<n−1∧j=i}
      min := a[i];
      p := i;
      {invariant: min=<all a[i..j]∧i=<n−1∧i=<j=<n
      ∧i=<p=<j ∧min=a[p]}
      for j := i+1 to n do
        if a[j]<min then
          begin {update current minimum in unsorted part of array}
            min := a[j];
            p := j
          end;
          {assert: min=<all a[i..n]∧min=a[p]}

  a[p] := a[i];
  a[i] := min
end
{assert: a[1..n] sorted in non-descending order∧a permutation
of original data set}
end
```

Notes on design

1. In analyzing the selection sort algorithm there are three parameters that are important: the number of comparisons made, the number of exchanges made, and the number of times the minimum is updated. The first time through the inner loop n-1 comparisons are made, the second time n-2, the third time n-3, and finally 1 comparison is made. The number of comparisons is therefore always: pc= (n -1) + (n-2)+ (n —3)+ +1= n(n —1)/2 (by Gauss' formula)

The number of exchanges is always (n-1) because it is equal to the number of times the outer loop is executed. Calculation of the number of times the minimum is updated involves a more detailed analysis since it is dependent on the data distribution. On average it can be shown that there are (n loge n+ en) updates required for random data.

2. Of the simple sorting algorithms the selection sort is one of the best because it keeps to a minimum the number of exchanges made. This can be important if there is a significant amount of data associated with each element.
3. In this design we saw how a complete algorithm is built by first designing an algorithm to solve the simplest problem. Once this is done it can be generalized to provide the complete solution.
4. The number of comparisons required by the selection sort can be reduced by considering elements in pairs and finding the minimum and maximum at the same time. Some care must be taken in implementing this algorithm.
5. There are more sophisticated and efficient ways of carrying out the selection process.

Applications

Sorting only small amounts of data—much more efficient methods are used for large data sets.

Supplementary problems

5.2.1 Sort an array into descending order.

5.2.2 Implement a selection sort that removes duplicates during the sorting process.

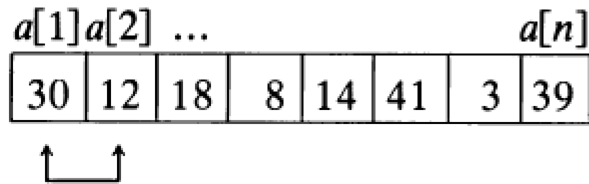
SORTING BY EXCHANGE

Problem

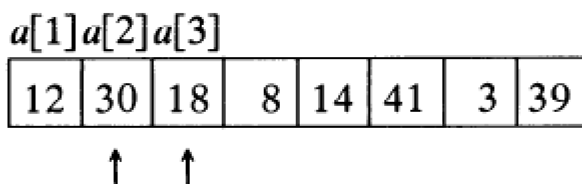
Given a randomly ordered set of n numbers sort them into non-descending order using an exchange method.

Algorithm development

Almost all sorting methods rely on exchanging data to achieve the desired ordering. The method relies heavily on an exchange mechanism. Suppose we start out with the following random data set:



Sorting is a way of increasing the order in the array. The first two elements arc "out of order" in the sense that no matter what the final sorted configuration 30 will need to appear later than 12. If the 30 and 12 are interchanged we will have in a sense "increased the order" in the data. This leads to the configuration below:

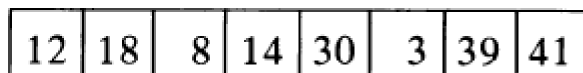


The investigation we have made suggests that the order in the array can be increased using the following steps:

1. For all adjacent pairs in the array do

(a) if the current pair of elements is not in non-descending order then exchange the two elements.

After applying this idea to all adjacent pairs in our current data set we get the configuration below:



The repeated exchange method we have been developing guarantees that with each pass through the data one additional element is sorted. Since there are n elements in the data this implies that (n-1) passes (of decreasing length) must be made through the array to complete the sort.

Algorithm description

1. Establish the array $a[1..n]$ of n elements.
2. While the array is still not *sorted* do
   (a) set the order indicator *sorted* to *true*;
   (b) for all adjacent pairs of elements in the unsorted part of the array do
      (b.1) if current adjacent pair not in non-descending order then
         (1.a) exchange the elements of the pair,
         (1.b) set *sorted* to *false*.
3. Return the sorted array.

Pascal implementation

```
procedure bubblesort(var a: nelements; n: integer);
var i {index for number of passes through the array},
  j {index for unsorted part of array},
  t {temporary variable used in exchange}: integer;
  sorted {if true after current pass then array sorted}: boolean;

begin {sorts array a[1..n] into non-descending order by exchange
method}
  {assert: n > 0}
  sorted := false;
  i := 0;
  {invariant: after ith iteration i =< n ∧a[n - i + 1..n] ordered ∧all
  a[1..n - i] =< all a[n - i + 1..n] ∨ (i < n ∧a[1..n - i] ordered ∧
  a[n - i + 1..n] ordered ∧all a[1..n - i] =< all a[n - i + 1..n])}
  while (i < n) and (not sorted) do
    begin {make next pass through unsorted part of array}
      sorted := true;
      i := i + 1;
      {invariant: after jth iteration j =< n - i ∧all a[1..j] =< a[j + 1]}
      for j := 1 to n - i do
        if a[j] > a[j + 1] then
          begin {exchange pair and indicate another pass required}
            t := a[j];
            a[j] := a[j + 1];
            a[j + 1] := t;
            sorted := false
          end
      {assert: all a[1..n - i] =< a[n - i + 1]}
    end
```

Notes on design

1. The relevant parameters for analyzing this algorithm are the number of comparisons and number of exchanges made. The minimum number of comparisons is (n- 1) when the data is already sorted. The maximum number of comparisons occur when (n-1) passes are made. In this case n(n —1)/2 comparisons are made. If the array is already sorted zero exchanges are made. In the worst case there are as many exchanges as there are comparisons, i.e. n(n —1)/2 exchanges are required. In the average case n(n 1)/4 exchanges are made.

2. A weakness of this algorithm is that it relies more heavily on exchanges than most other sorting methods. Since exchanges are relatively time-consuming. this characteristic makes the method very costly for sorting large random data sets. There is, however, one instance where a bubblesort (as it is usually called) can be efficient. If a data set has only a small percentage of elements out of order a bubblesort may require only a small number of exchanges and comparisons.

Applications

Only for sorting data in which a small percentage of elements are out of order.

Supplementary problems

5.3.1 Use a count of the number of comparisons and exchanges made to compare the selection sort and bubblesort for random data.

5.3.2 Implement a version of the bubblesort that builds tip the sorted array from smallest to largest rather than as in the present algorithm.

5.3.3 Design and implement a modified bubblesort that incorporates exchanges in the reverse direction of fixed length.

5.3.4 Try to design a less efficient bubblesort than the present algorithm.

SORTING BY INSERTION

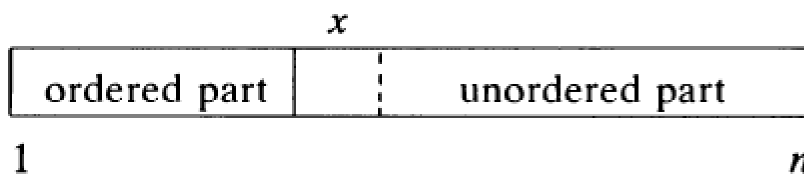
Problem

Given a randomly ordered set of n numbers sort them into non-descending order using an insertion method.

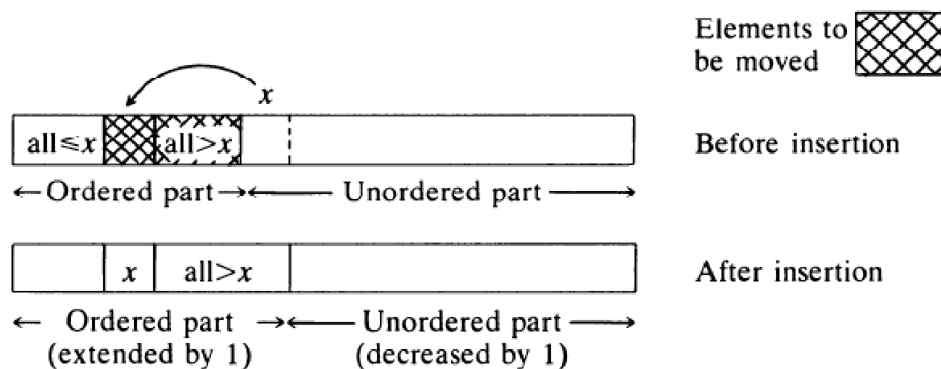
Algorithm development

Sorting by insertion is one of the more obvious and natural ways to sort information. It approximates quite closely the ordering procedure that card players often use. Central to this algorithm is the idea of building up the complete solution by inserting an element from the unordered part into the current partially ordered solution, extending it by one element. This mechanism is suggestive of a selection sort where we selected the smallest element in the unordered part and placed it on the end of the sorted part.

We have:



A simple, systematic, and alternative way we could choose the next item to be inserted is to always pick the first element in the unordered part (i.e. x in our example). We then need to appropriately insert x into the ordered part and, in the process, extend the ordered section by one element. Diagrammatically



Algorithm description

1. Establish the array a[1..n] of n elements.
2. Find the minimum and put it in place to act as sentinel.
3. While there are still elements to be inserted in the ordered part do
    - (a) select next element x to be inserted;
    - (b) while x is less than preceding element do
        - (b.1) move preceding element up one position,
        - (b.2) extend search back one element further;
    - (c) insert x at current position.

Pascal implementation

```
procedure insertionsort(var a: nelements; n: integer);
var i {increasing index of number of elements ordered at each stage},
  j {decreasing index used in search for insertion position},
  first {smallest element in array},
  p {original position of smallest element},
  x {current element to be inserted}: integer;

begin {sorts array a[1..n] into non-descending order using insertion
method}
  {assert: n>0∧i=1}
  {find minimum to act as sentinal}
  first := a[1]; p := 1;
  for i := 2 to n do
    if a[i]<first then
      begin
        first := a[i];
        p := i
      end;
  a[p] := a[1];
  a[1] := first;
 {invariant: 1=<i=<n ∧a[1..i] ordered}
 for i := 3 to n do
   begin {insert ith element — note a[1] is a sentinal}
     x := a[i];
     j := i;
     {invariant: 1=<j=<i∧x=<a[j..i]}
     while x<a[j-1] do
       begin {search for insertion position and move up elements}
         a[j] := a[j-1];
         j := j-1
       end;
     {assert: 1=<j∧x=<a[j+1..i]}
     {insert x in order}
     a[j] := x
   end
   {assert: a[1..n] sorted in non-descending order and a
   permutation of original data}
end
```

Notes on design

1. In analyzing the insertion sort two parameters are important. They are the number of comparisons **(i.e. x<a[j-1])** made and secondly the number of array

elements that need to be moved or shifted. The inner while-loop must be executed at least once for each i value. It follows that at least (2n-3) comparisons must be made. At the other extreme at most (i-1) comparisons must be made for each i value. Using the standard summation formula we can show that in the worst case

$$(n2+n —4) /2$$

comparisons will be required. Usually the average case behavior is of more interest.

Assuming that on average (1+1)12 comparisons are needed before x can be inserted each time it can be shown that

$$(n2+6n— 12)/ 4$$

comparisons are required. The performance of the algorithm is therefore **$0(n^2)$.** Similar arguments can be used to compute the number of move operations.

2. The insertion sort is usually regarded as the best of the **n^2** algorithms for sorting small random data sets.

Applications:

Where there are relatively small data sets. It is sometimes used for this purpose in the more advanced quicksort algorithm.

Supplementary problems

5.4.1 Compare the selection sort and insertion sort for random data. Use the number of moves and the number of comparisons to make the comparative study.

5.4.2 A small saving can be made with the insertion sort by using a method that does other than selection of the next element for insertion. Try to incorporate this suggestion.

BINARY SEARCH

Problem

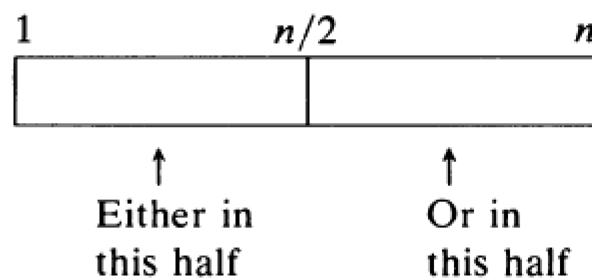
Given an element **x** and a set of data that is in strictly ascending numerical order find whether or not **x** is present in the set.

Algorithm development

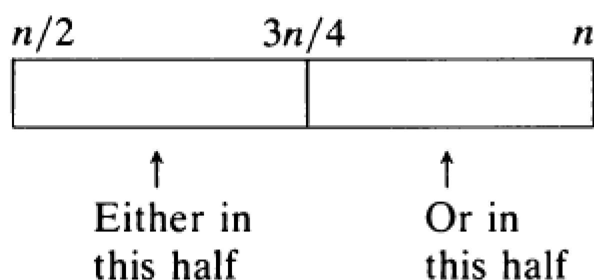
The problem of searching an ordered list such as a dictionary or telephone directory occurs frequently in computing.

Binary search is an efficient algorithm for finding an item from a sorted list of items.

It is easy to see that in all cases the value in the set that we are seeking is either in the first half of the list or the second half of the list (it may also be the middle value in the set). For example,



We can establish the relevant half by comparing the value sought with the value in the middle of the set. This single test will eliminate half of the values in the set from further consideration. Now we have a problem only half the size of the original problem. Suppose it is established that the value we are seeking is in the second half of the list (e.g. somewhere between the (n/2)th value and the nth value).



The halving strategy that we have been considering is one of the most widely used methods in computing science. It is commonly known as the **divide-and-conquer strategy.** The corresponding searching method we are starting to develop is known as the binary search algorithm. At this stage we have the general strategy:

Algorithm description

1. Establish the array $a[1..n]$, and the value sought x.
2. Assign the *upper* and *lower* variables to the array limits.
3. While *lower*<*upper* do
   (a) compute the middle position of remaining array segment to be searched,
   (b) if the value sought is greater than current middle value then
      (b.1) adjust lower limit accordingly
   else
      (b'.1) adjust upper limit accordingly.
4. If the array element at *lower* position is equal to the value sought then
   (a) return found
   else
   (a') return not found.

Pascal implementation

```
procedure binarysearch(var a: nelements; n,x: integer; var found:
boolean);
var lower {lower limit of array segment still to be searched},
  upper {upper limit of array segment still to be searched},
  middle {middle of array segment still to be searched}: integer;

begin {binary searches array a[1..n] for element x}
  {assert: n>0∧a[1..n] sorted in ascending order∧exists k such that
  1=<k=<n∧x=a[k] if x is present}
  lower := 1;
  upper := n;
  {invariant: lower>=1∧upper=<n∧x in a[lower..upper] if present}
  while lower<upper do
    begin {increase lower and decrease upper keeping x in range if
    present}
      middle := (lower+upper) div 2;
      if x>a[middle] then
        lower := middle+1
      else
        upper := middle
    end;

  {assert: lower=upper =k∧x=a[k] if x in a[1..n]}
  found := (a[lower]=x)
end
```

Notes on design

1. The binary search algorithm in general offers a much more efficient alternative than the linear search algorithm. Its performance can best be understood in terms of a binary search tree.

2. With each iteration, if x is *present* in the array, *lower* will be increased and *upper* will be decreased in such a way that the condition:

$$a[lower] \leqslant x \leqslant a[upper] \text{ and } lower \leqslant upper$$

remains true. Termination will occur when $lower = upper$ and hence *if x is present* we will have

$$a[lower] = x = a[upper]$$

Supplementary problems

5.7.1 Design and implement versions of the binary search that have the following loop structures:

(a) **while** $lower < upper+1$ **do**
(b) **while** $lower < upper-1$ **do**

5.7.2 Implement versions of the binary search that make the following paired changes to *lower* and *upper*:

(a) $\left.\begin{array}{l} lower := middle+1 \\ upper := middle-1 \end{array}\right\}$

(b) $\left.\begin{array}{l} lower := middle \\ upper := middle \end{array}\right\}$