13 Standard algorithms

Each new program has its own challenges, its own unique features. But the same old subproblems keep recurring. So, very often at some point in your program you will have to search for a key value in an ordered collection, or sort some data. These subproblems have been beaten to death by generations of computer theoreticians and programmers. The best approaches are now known and defined by standard algorithms.

This chapter presents four of these. Section 13.1 looks at searching for key values. The next three sections explore various "sorting" algorithms.

The study of algorithms once formed a major part of sophomore/junior level studies in Computer Science. Fashions change. You will study algorithms in some subsequent subjects but probably only to a limited extent, and mainly from a theoretical perspective while learning to analyse the complexity of programs.

13.1 FINDING THE RIGHT ELEMENT: BINARY SEARCH

The code in the "keyword picker' (the example in section 12.4) had to search linearly through the table of keywords because the words were unordered. However, one often has data that are ordered. For instance you might have a list of names:

```
"Aavik",      "Abbas",      "Abbot",      "Abet",
…
"Dyer",       "Dzieran",    "Eady",       "Eames",
…
"O'Keefe",    "O'Leary",    …
…
"Tierney",    "Tomlinson",  "Torrecilla", …
…
"Yusuf",      "Zitnik",     "Zuzic",      "Zylstra"
```

If you are searching for a particular value, e.g. Sreckovic, you should be able to take advantage of the fact that the data are ordered. Think how you might look up the name in a phone book. You would open the book in the middle of the "subscriber" pages to find names like "Meadows". The name you want is later in the alphabet so you pick a point half way between Meadows and Zylstra, and find a name like Terry. Too far, so split again between Meadows and Terry to get to Romanows. Eventually you will get to Sreckovic. May take a bit of time but it is faster than checking of Aavik, Abbas, Abbot, ….

Subject to a few restrictions, the same general approach can be used in computer programs. The restrictions are that all the data must be in main memory, and that it be easy to work out where each individual data item is located. Such conditions are easily satisfied if you have something like an ordered array of names. The array will be in memory; indexing makes it easy to find a particular element. There are other situations where these search techniques are applicable; we will meet some in Part IV when we look at some "tree structures".

The computer version of the approach is known as "binary search". It is "binary" because at each step, the portion of the array that you must search is cut in half ("*bi*-sected"). Thus, in the names example, the first name examined ("Meadows") established that the name sought must be in the second half of the phone book. The second test (on the name "Terry") limited the search to the third quarter of the entire array.

13.1.1 An iterative binary search routine

The binary search algorithm for finding a key value in an array uses two indices that determine the array range wherein the desired element must be located. (Of course, it is possible that the element sought is not present! A binary search routine has to be able to deal with this eventuality as well.) If the array has N elements, then these low and high indices are set to 0 and N-1.

The iterative part involves picking an index midway between the current low and high limits. The data value located at this midway point is compared with the desired key. If they are equal, the data has been found and the routine can return the midway value as the index of the data.

If the value at the midway point is lower than that sought, the low limit should be increased. The desired data value must be located somewhere above the current midway point. So the low limit can be set one greater than the calculate midway point. Similarly, if the value at the midway point is too large, the high limit can be lowered so that it is one less than the midway point.

Each iteration halves the range where the data may be located. Eventually, the required data are located (if present in the array). If the sought key does not occur in the array, the low/high limits will cross (low will exceed high). This condition can be used to terminate the loop.

A version of this binary search for character strings is:

```
typedef char Name[32];

int BinarySearch(const Name key, const Name theNames[], int
num)
{
    int low = 0;
    int high = num - 1;

    while(low<=high) {
           int midpoint = (low + high) / 2;
           int result = strcmp(key, theNames[midpoint]);

           if(result == 0)
                  return midpoint;

           if(result < 0)
                  high = midpoint - 1;
           else
                  low = midpoint + 1;
    }
    return -1;
}
```

The function returns the index of where the given key is located in the array argument. The value -1 is returned if the key is not present. The strings are compared using `strcmp()` from the string library. Versions of the algorithm can be implemented for arrays of doubles (the comparison test would then be expressed in terms of the normal greater than (>), equals (==) and less than (<) operators).

Binary search is such a commonly used algorithm that it is normally packaged in stdlib as `bsearch()`. The stdlib function is actually a general purpose version that can be used with arrays of different types of data. This generality depends on more advanced features of C/C++.

13.1.2 Isn't it a recursive problem?

The binary search problem fits well with the "recursive" model of problem solving (section 4.8). You can imagine the binary search problem being solved by a bureaucracy of clerks. Each clerk follows the rules:

```
look at the array you've been given, if it has no elements
           report failure
else
    look at the midpoint of your array,
    if this contains the sought value then report success
    else
    if the midpoint is greater than the sought value
           pass the bottom half of your array to the next clerk
```

```
                        asking him to report whether the data are
                               present
                 report whatever the next clerk said
          else
                 pass the top half of your array to the next clerk
                        asking him to report whether the data are
                               present
                 report whatever the next clerk said
```

This version is also easy to implement. Typically, a recursive solution uses a "setting up function" and a second auxiliary function that does the actual recursion. The recursive routine starts with the termination test (array with no elements) and then has the code with the recursive calls. This typical style is used in the implementation:

Test for "empty" array

Success, have found item

Recursive call to search bottom half of array

Recursive call to search top half of array

```
int AuxRecBinSearch(const Name key, const Name theNames[],
            int low, int high)
{
    if(low > high)
            return -1;

    int midpoint = (low + high) / 2;

    int result = strcmp(key, theNames[midpoint]);

    if(result == 0)
            return midpoint;
    else
    if(result < 0)
            return AuxRecBinSearch(key, theNames,
                          low, midpoint-1);
    else
            return AuxRecBinSearch(key, theNames,
                          midpoint + 1, high);
}
```

The "setting up" function

```
int RecBinarySearch(const Name key, const Name theNames[],
    int num, )
{
    return AuxRecBinSearch(key, theNames, 0, num-1);
}
```

In this case, the recursive solution has no particular merit. On older computer architectures (with expensive function calls), the iterative version would be much faster. On a modern RISC architecture, the difference in performance wouldn't be as marked but the iterative version would still have the edge.

Essentially, binary search is too simple to justify the use of recursion. Recursive styles are better suited to situations when there is reasonable amount of work to do as the recursion is unwound (when you have to do something to the result returned by the clerk to whom you gave a subproblem) or where the problem gets broken into separate parts that both need to be solved by recursive calls.

13.1.3 What was the cost?

The introduction to this section argued that binary search was going to be better than linear search. But how much better is it?

You can get an idea by changing the search function to print out the names that get examined. You might get output something like:

```
Meadows
Terry
Romanows
Smith
Stone
Sreckovic
```

In this case, six names were examined. That is obviously better than checking "Aavik", "Abbas", "Abbot", "Abet", …; because the linear search would involve examination of hundreds (thousands maybe) of names.

Of course, if you wanted the name Terry, you would get it in two goes. Other names might take more than six steps. To get an idea of the cost of an algorithm, you really have to consider the worst case.

The binary search algorithm can be slightly simplified by having the additional requirement that the key has to be present. If the array has only two elements, then a test against element `[0]` determines the position of the key; either the test returned "equals" in which case the key is at `[0]`, or the key is at position `[1]`. So one test picks from an array size of 2. Two tests suffice for an array size of 4; if the key is greater than element `[1]` then a second test resolves between `[2]` and `[3]`. Similarly, three tests pick from among 8 (or less elements).

Tests	Number of items can pick from
1	2
2	4
3	8
4	16
…	…
…	…
?	N

In general, k tests are needed to pick from 2^k items. So how many tests are needed to pick from an array of size N?

You need:

$$2^k \geq N$$

This equation can be solved for *k*. The equation is simplified by taking the logarithm of both sides:

$$k \log(2) = \log(N)$$

You can have logarithms to the base 10, or logs to any base you want. You can have logarithms to the base 2. Now $\log_2(2)$ is 1. This simplifies the equation further:

$$\textit{number of tests needed} = k = \log_2(N) = \lg(N)$$

The cost of the search is proportional to the number of tests, so the cost of binary search is going to increase as O(lg(N)). Now a lg(N) rate of growth is a lot nicer than linear growth :

10	3.3
50	5.6
100	6.6
250	7.9
500	8.9
1000	9.9
5000	12.2
10000	13.2
50000	15.6
100000	16.6
500000	18.9
1000000	19.9

A linear search for a key in a set of a million items could (in the worst case) involve you in testing all one million. A binary search will do it in twenty tests.

Of course, there is one slight catch. The data have to be in order before you can do binary search. You are going to have to do some work to get the data into order.

13.2 ESTABLISHING ORDER

Sometimes, you get data that are relatively easy to put into order. For example, consider a class roll ordered by name that identifies pupils and term marks (0 … 100):

Armstrong, Alice S.	57
Azur, Hassan M.	64
Bates, Peter	33
…	
Yeung, Chi Wu	81
Young, Paula	81

A common need would be a second copy of these data arranged in increasing order by mark rather than by name.

You can see that a pupil with a mark of 100 would be the last in the second reordered array, while any getting 0 would be first. In fact, you can work out the position in the second array appropriate for each possible mark.

This is done by first checking the distribution of marks. For example, the lowest recorded mark might be 5 with two pupils getting this mark; the next mark might be 7, then 8, each with one pupil. Thus the first two slots in the reordered copy of the array would be for pupils with a mark of 5, the next for the pupil with mark 7, and so on. Once this distribution has been worked out and recorded, you can run through the original list, looking at each pupil's mark and using the recorded details of the distribution to determine where to place that pupil in the reordered array. The following algorithm implements this scheme:

1. Count the number of pupils with each mark (or, more generally, the number of "records" with each "key").

   These counts have to be stored in an array whose size is determined by the range of key values. In this case, the array has 101 elements (`mark_count[0]` … `mark_count[100]`). The data might be something like

```
Mark       0, 1, 2, 3, 4, 5, 6, 7, 8, 9, …
mark_count 0, 0, 0, 0, 0, 2, 0, 1, 1, 0, …
```

2. Change the contents of this `mark_count` array so that its elements record the number of pupils with marks less than or equal to the given mark.

   i.e. code something like:

```
   for(i=1; i<101;i++)
          mark_count[i] += mark_count[i-1];
```

```
Mark       0, 1, 2, 3, 4, 5, 6, 7, 8, 9, …
mark_count 0, 0, 0, 0, 0, 2, 2, 3, 4, 4, …
```

3. Run through the array of pupils, look up their mark in the `mark_count` array. This gives their relative position in the reordered array (when implementing, you have to remember C's zero-based arrays where position 2 is in `[1]`). The data should be copied to the other array. Then the `mark_count` entry should be reduced by one.

   For example, when the first pupil with mark 5 is encountered, the `mark_count` array identifies the appropriate position as being the 2nd in the reordered array (i.e. element `[1]`). The `mark_count` array gets changed to:

```
mark_count 0, 0, 0, 0, 0, 1, 2, 3, 4, 4, …
```

so, when the next pupil with mark 5 is found his/her position can be seen to be the first in the array (element `[0]`).

An implementation of this algorithm is:

```
typedef char Name[40];
…

void MakeOrderedCopy(const Name names[], const int marks[],
    int num,
    Name ord_names[], int ord_marks[])
{
    int mark_count[101];
```

Initialize counts for keys (marks)

```
    for(int i = 0; i< 101; i++)
           mark_count[i] = 0;
```

Get frequencies of each key (mark)

```
    // Count number of occurrences of each mark
    for(i = 0; i < num; i++) {
           int mark = marks[i];
           mark_count[mark]++;
    }
```

Turn counts into "positions"

```
    // Make that count of number of pupils with marks less than
    // or equal to given mark

    for(i=1; i<101; i++)
           mark_count[i] += mark_count[i-1];
```

Copy data across into appropriate positions

```
    for(i=0; i < num; i++) {
           int mark = marks[i];
           int position = mark_count[mark];
           position--; // correct to zero based array
           // copy data
           strcpy(ord_names[position], names[i]);
           ord_marks[position] = marks[i];
           // update mark_count array
           mark_count[mark]--;
    }
}
```

Note how both data elements, name and mark, have to be copied; we have to keep names and marks together. Since arrays can not be assigned, the `strcpy()` function must be used to copy a `Name`.

The function can be tested with the following code:

```
#include <iostream.h>
#include <iomanip.h>
```

```
#include <string.h>

typedef char Name[40];

Name pupils[] = {
    "Armstrong, Alice S.",
    "Azur, Hassan M.",
    "Bates, Peter",
    …
    …
    "Ward, Koren",
    "Yeung, Chi Wu",
    "Young, Paula",
    "Zarra, Daniela"
};

int marks[] = {
    57, 64, 33, 15, 69, 61, 58,
    …
    45, 81, 81, 78
};

int num = sizeof(marks) / sizeof(int);
int num2 = sizeof(pupils) /sizeof(Name);

int main()
{
    Name ordnms[500];
    int ordmks[500];
    MakeOrderedCopy(pupils, marks,  num, ordnms, ordmks);
    cout << "Class ordered by names: " << endl;
    for(int i = 0; i< num; i++)
           cout << setw(40) << pupils[i] << setw(8)
                         << marks[i] << endl;
    cout << "\n\nOrdered by marks:" << endl;
    for( i = 0; i< num; i++)
           cout << setw(40) << ordnms[i] << setw(8)
                  << ordmks[i] << endl;
    return 0;
}
```

which produces output like:

```
Class ordered by names:
                     Armstrong, Alice S.      57
                         Azur, Hassan M.      64
                            Bates, Peter      33
…
…
                           Yeung, Chi Wu      81
                            Young, Paula      81
```

```
                                     Zarra, Daniela      78

Ordered by marks:
                                    McArdie, Hamish       5
                                     Hallinan, Jack       5
…
…
                                       Young, Paula      81
                                      Yeung, Chi Wu      81
                                      Horneman, Sue      87
                                     Goodman, Jerry      91
```

These example data illustrate another minor point. The original list had Yeung and Young in alphabetic order, both with mark 81. Since their marks were equal, there was no need to change them from being in alphabetic order. However the code as given above puts pupils with the same mark in *reverse* alphabetic order.

"Stable sorts"

Sometimes, there is an extra requirement: the sorting process has to be "stable". This means that records that have the same key values (e.g. pupils with the same mark) should be left in the same order as they originally were.

It is easy to change the example code to achieve stability. All you have to do is make the final loop run backwards down through the array:

```
for(i=num-1; i >= 0; i--) {
    int mark = marks[i];
    int position = mark_count[mark];
    position--; // correct to zero based array
    // copy data
    strcpy(ord_names[position], names[i]);
    ord_marks[position] = marks[i];
    // update mark_count array
    mark_count[mark]--;
    }
```

Implementation limitations

This implementation is of course very specific to the given problem (e.g. the `mark_count` array is explicitly given the dimension 101, it does not depend on a data argument). The function can be made more general but this requires more advanced features of the C/C++ language. Nevertheless, the code is simple and you should have no difficulty in adapting it to similar problems where the items to be sorted have a narrow range of keys and you want two copies of the data.

What is the cost?

To get an idea of the cost of this sorting mechanism, you have to look at the number of times each statement gets executed. Suppose that `num`, the number of data elements, was 350, then the number of times that each statement was executed would be as shown below:

```
     void MakeOrderedCopy(const Name names[], const int marks[],
            int num,
            Name ord_names[], int ord_marks[])
     {
            int mark_count[101];

            for(int i = 0; i< 101; i++)
101                mark_count[i] = 0;

            // Count number of occurrences of each mark
            for(i = 0; i < num; i++) {
350                int mark = marks[i];
350                mark_count[mark]++;
            }
            // Make that count of number of pupils with marks
            // less than or equal to given mark
            for(i=1; i<101; i++)
101                mark_count[i] += mark_count[i-1];

            for(i= 0; i < num; i++) {
350                int mark = marks[i];
350                int position = mark_count[mark];
350                position--; // correct to zero based array
                   // copy data
350                strcpy(ord_names[position], names[i]);
350                ord_marks[position] = marks[i];
350                mark_count[mark]--;
            }
     }
```

The costs of these loops is directly proportional to the number of times they are executed. So the cost of this code is going to be proportional to the number of entries in the array (or the range of the key if this was larger, the '101' loops would represent the dominant cost if you had less than 100 records to sort).

This algorithm is O(N) with N the number of items to sort. That makes it a relatively cheap algorithm.

13.3 A SIMPLE SORT OF SORT

The last example showed that in some special cases, you could take data and get them sorted into order in "linear time". However, the approach used is limited.

Normally require sorting "in situ"

There are two limitations. Firstly, it made a sorted copy of the data so that you had to sufficient room in the computer's memory for two sets of data (the initial set with the pupils arranged in alphabetic order, and the sorted set with them arranged by mark). This is usually not acceptable. In most cases where it is necessary to sort data, you can't have two copies of the entire data set. You may have one or two extra items (records) but you can't duplicate the whole set. Data have to be sorted "in situ" (meaning "in place"); the original entries in the data arrays have to be rearranged to achieve the required order.

Small range of key values

The other simplifying factor that does not usually apply was the limited range of the keys. The keys were known to be in the range 0…100. This made it possible to have the `mark_count[101]` array. Usually the keys span a wider range, e.g. bank accounts can (I suppose) have any value from -2000000000 to +2000000000. You can't allocate an array equivalent to `mark_count[101]` if you need four thousand million entires.

When you don't have the special advantages of data duplication and limited key range, you must use a general purpose sorting function and you will end up doing a lot more work.

The late 1950s and early 1960s was a happy time for inventors of sorting algorithms. They invented so many that the topic has gone quite out of fashion. There is a whole variety of algorithms from simple (but relatively inefficient) to complex but optimized. There are also algorithms that define how to sort data collections that are far too large to fit into the memory of the computer . These "external sorts" were important back when a large computer had a total of 32768 words of memory. Now that memory sizes are much larger, external sorts are of less importance; but if you ever do get a very large set of data you can dig out those old algorithms, you don't have to try to invent your own.

There is a fairly intuitive method for sorting the data of an array into ascending order. The first step is to search through the array so as to select the smallest data value; this is then exchanged for the data element that was in `element [0]`. The second step repeats the same process, this time searching the remaining unsorted subarray `[1]`…`[N-1]` so as to select its smallest element, which gets swapped for the data value originally in `[1]`. The selection and swapping processes are repeated until the entire array has been processed.

A version of this code, specialized to handle an array of characters is:

Code to find index of minimum value in rest of array

Code to swap data value with smallest

```
void SelectionSort(char data[], int n)
{
    for(int i = 0; i < n-1; i++) {
           int min = i;
           for(int j=i+1; j<n; j++)
                  if(data[j] < data[min]) min = j;

           char temp = data[min];
           data[min] = data[i];
           data[i] = temp;
    }
}
```

The old cliche states "A picture is worth a thousand words"; so how about a few pictures to illustrate how this sort process works. The following program uses the curses package to draw pictures illustrating the progress of the sort:

```
#include "CG.h"

char testdata[] = {
    'a', 'r', 'g', 'b', 'd',
    'i', 'q', 'j', 'm', 'o',
    't', 'p', 'l', 's', 'n',
    'f', 'c', 'k', 'e', 'h'
};

int n = sizeof(testdata) /sizeof(char);

void ShowArray()
{
    CG_ClearWindow();
    for(int i=0; i < n; i++) {
           int x = i + 1;
           int y = 1 + testdata[i] - 'a';
           CG_PutCharacter(testdata[i], x, y);
    }
    CG_Prompt("continue>");
    char ch = CG_GetChar();
}

void SelectionSort(char data[], int n)
{
    ShowArray();
    for(int i = 0; i < n-1; i++) {
           int min = i;
           for(int j=i+1; j<n; j++)
                  if(data[j] < data[min]) min = j;
           char temp = data[min];
           data[min] = data[i];
           data[i] = temp;
           ShowArray();
    }
}

int main()
{
    CG_Initialize();
    SelectionSort(testdata, n);
    CG_Reset();
    return 0;
}
```

Figure 13.1 illustrates the output of this program.

What is the cost?

The cost is determined by the number of data comparisons and data exchanges that must be done in that inner loop. If there are *N* elements in the array, then:

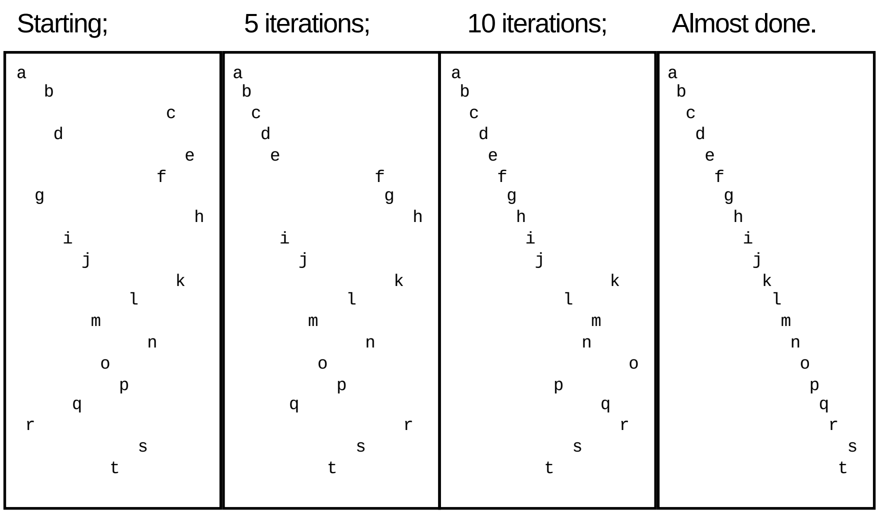


Figure 13.1 Illustrating selection sort.

i	iterations of j loop
0	N-1
1	N-2
2	N-3
…	…
N-2	1

Each iteration of the *j* loop involves a comparison (and, if you are unlucky, an assignment). The cost will be proportional to the sum of the work of all the *j* loops:

$$\text{cost} \propto \sum_{i=1}^{i=N-1} (N-i)$$

It is an arithmetic progression and its sum is "first term plus last term times half the number of terms:

$$(N-1 + 1)\ (N-1)/2$$

$$0.5N^2 - 0.5N$$

An N^2 cost The cost of this sorting process increases as the square of the number of elements.

13.4 QUICKSORT: A BETTER SORT OF SORT

An N^2 sorting process is not that wildly attractive; doubling the number of elements increases the run time by a factor of four. Since you know that a data element can be found in an ordered array in time proportional to lg(N), you might reasonably expect a sort process to be cheaper than N^2. After all, you can think of "sorting" as a matter of adding elements to an ordered set; you have N elements to add, finding the right place for each one should be lg(N), so an overall cost of Nlg(N) seems reasonable.

That argument is a little spurious, but the conclusion is correct. Data can be sorted in Nlg(N) time. There are some algorithms that guarantee an Nlg(N) behaviour. The most popular of the sophisticated sorting algorithms does not guarantee Nlg(N) behaviour; in some cases, e.g. when the data are already more or less sorted, its performance deteriorates to N^2. However, this "Quicksort" algorithm usually runs well.

13.4.1 The algorithm

Quicksort is recursive. It is one of those algorithms that (conceptually!) employs an bureaucracy of clerks each of whom does a little of the work and passes the rest on to its colleagues. The basic rules that the clerks follow are:

1 If you are given an array with one element, say that you have sorted it.

2 If you are given an array with several data elements, "shuffle them" into two groups – one group containing all the "large" values and the other group with all the "small" values. Pass these groups to two colleagues; one colleague sorts the "large" values, the other sorts the "small" values.

Ideally, each clerk breaks the problem down into approximately equal sized parts so that the next two clerks get similar amounts of work. So, if the first clerk in a group is given the data:

```
37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91
```

then (s)he splits this into:

```
37, 21, 12, 33, 50, 44, 28,
```

and

```
62, 62, 64, 97, 82, 103, 91
```

It is up to the next two clerks to sort out these subarrays.

In the example just shown, the clerk knew "intuitively" that the best partitioning value would be around 60 and moved values less than this into the "small" group with the others in the "large" group. Usually, the best partitioning value is not known. The scheme works even if a non-optimal value is used. Suppose that the first clerk picked 37, then the partitions could be:

```
33, 21, 12, 28
```

and

```
82, 97, 64, 62, 62, 103, 44, 50, 37, 91
```

The first partition contains the values less than 37, the second contains all values greater than or equal to 37. The partitions are unequal in size, but they can still be passed on to other clerks to process.

The entire scheme for dealing with the data is shown in Figure 13.2. The pattern of partitions shown reflects the non-optimal choice of partitioning values. In this example, it is quite common for the partitioning to separate just one value from a group of several others rather than arranging an even split. This necessitates extra partitioning steps later on. It is these deviations from perfect splitting that reduce the efficiency of Quicksort from the theoretical Nlg(N) optimum.

How should the data be shuffled to get the required groups?

You start by picking the value to be used to partition the data, and for lack of better information you might as well take the first value in your array, in this case 37. Then you initialize two "pointers" – one off the left of the array, and one off the right end of the array:

```
       37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91
Left ⇑                                                         ↑Right
```

Move the "right" pointer to the left until its pointing to a data value less than or equal to the chosen partitioning value:

```
       37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91
Left ⇑                                                  ↑Right
```

Next, move the "left" pointer to the right until its pointing to a data value greater than or equal to the partitioning value:

```
       37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91
  Left ⇑                                                ↑Right
```

You have now identified a "small" value in the right hand part of the array, and a "large" value in the left hand part of the array. These are in the wrong parts; so exchange them:

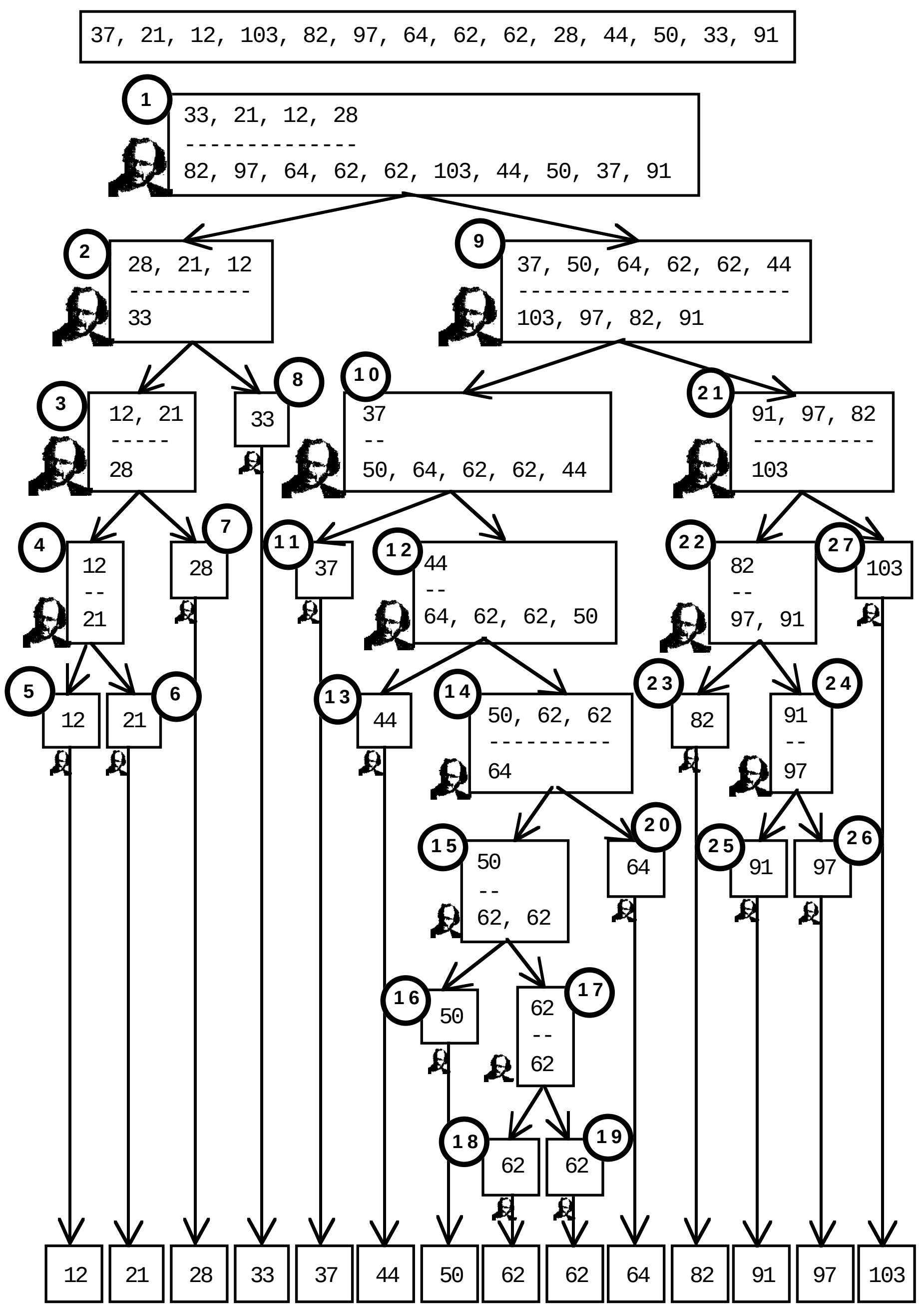


Figure 13.2 A bureaucracy of clerks "Quicksorts" some data. The circled numbers identify the order in which groups of data elements are processed.

```
     33, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 37, 91
Left ⇑                                                ↑Right
```

Do it again. First move the right pointer down to a value less than or equal to 37; then move the left pointer up to a value greater than or equal to 37:

```
     33, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 37, 91
            Left ⇑                        ↑Right
```

Once again, these data values are out of place, so they should be exchanged:

```
     33, 21, 12, 28, 82, 97, 64, 62, 62, 103, 44, 50, 37, 91
            Left ⇑                       ↑Right
```

Again move the right pointer down to a value less than 37, and move the left pointer up to a greater value:

```
     33, 21, 12, 28, 82, 97, 64, 62, 62, 103, 44, 50, 37, 91
                     ↑Right
                     Left ⇑
```

In this case, the pointers have crossed; the "right" pointer is to the left of the "left" pointer. When this happens, it means that there weren't any more data values that could be exchanged.

The "clerk" working on this part of the problem has finished; the rest of the work should be passed on to others. The data in the array up to the position of the "right" pointer are those values less than the value that the clerk chose to use to do the partitioning; the other part of the array holds the larger values. These two parts of the array should be passed separately to the next two clerks.

The scheme works. The rules are easy enough to be understood by a bureaucracy of clerks. So they are easy enough to program.

13.4.2 An implementation

The recursive Quicksort function takes as arguments an array, and details of the range that is to be processed:

```
void Quicksort( int d[], int left, int right);
```

The details of the range will be given as the indices of the leftmost (low) and rightmost (high) data elements. The initial call will specify the entire range, so if for example you had an array `data[100]`, the calling program would invoke the `Quicksort()` function as:

```
Quicksort(data, 0, 99);
```

Subsequent recursive calls to `Quicksort()` will specify subranges e.g. `Quicksort(data, 0, 45)` and `Quicksort(data, 46, 99)`.

The partitioning step that splits an array into two parts (and shuffles data so one part contains the "low" values) is sufficiently complex to merit being a separate function. Like `Quicksort()` itself, this `Partition()` function needs to be given the array and "left and right" index values identifying the range that is to be processed:

```
int Partition( int d[], int left, int right);
```

The `Partition()` function should return details of where the partition point is located. This "split point" will be the index of the array element such that `d[left]` … `d[split_point]` all contain values less than the value chosen to partition the data values. Function `Partition()` has to have some way of picking a partitioning value; it can use the value in the leftmost element of the subarray that it is to process.

The code for `Quicksort()` itself is nice and simple:

```
void Quicksort( int d[], int left, int right)
{
    if(left < right) {
           int split_pt = Partition(d,left, right);
           Quicksort(d, left, split_pt);
           Quicksort(d, split_pt+1, right);
           }
}
```

Shuffle the data
Give the two subarrays to two assistants

If the array length is 1 (`left == right`), then nothing need be done; an array of one element is sorted. Otherwise, use `Partition() to` shuffle the data and find a split point and pass the two parts of the split array on to other incarnations of `Quicksort()`.

The `Partition()` function performs the "pointer movements" described in the algorithm. It actually uses to integer variables (`lm` = left margin pointer, and `rm` = right margin pointer). These are intialized "off left" and "off right" of the subarray to be processed. The partitioning value, `val`, is taken as the leftmost element of the subarray.

```
int Partition( int d[], int left, int right)
{
    int val =d[left];
    int lm = left-1;
    int rm = right+1;
```

Initialize val and "left and right pointers"

The movement of the pointers is handled in a "forever" loop. The loop terminates with a return to the caller when the left and right pointers cross.

Inside the for loop, the are two `do … while` loops. The first `do … while` moves the right margin pointer leftwards until it finds a value less than or equal to `val`. The second `do … while` loop moves the left margin pointer rightwards, stopping when it finds a value greater than or equal to `val`.

If the left and right margin pointers have crossed, function Partition() should return the position of the right margin pointer. Otherwise, a pair of misplaced values have been found; these should be exchanged to get low values on the left and high values on the right. Then the search for misplaced values can resume.

Move right margin pointer leftwards

Move left margin pointer righwards

Exchange misplaced data

Return if all done

```
        for(;;) {
                do
                        rm--;
                while (d[rm] > val);

                do
                        lm++;
                while( d[lm] < val);

                if(lm<rm) {
                        int tempr = d[rm];
                        d[rm] = d[lm];
                        d[lm] = tempr;
                        }
                else
                        return rm;
                }
    }
```

What does it cost?

If you really want to know what it costs, you have to read several pages of a theoretical text book. But you can get a good idea much more simply.

The partitioning step is breaking up the array. If it worked ideally, it would keep splitting the array into halves. This process has to be repeated until the subarrays have only one element. So, in this respect, it is identical to binary search. The number of splitting steps to get to a subarray of size one will be proportional to lg(N).

In the binary search situation, we only needed to visit one of these subarrays of size one – the one where the desired key was located. Here, we have to visit all of them. There are N of them. It costs at least lg(N) to visit each one. So, the cost is Nlg(N). You can see from Figure 13.2 that the partitioning step frequently fails to split a subarray into halves; instead it produces a big part and a little part. Splitting up the big part then requires more partitioning steps. So, the number of steps needed to get to subarrays of size one is often going to be greater than lg(N); consequently, the cost of the sort is greater than Nlg(N).

In the worst case, every partition step just peels off a single low value leaving all the others in the same subarray. In that case you will have to do one partitioning step for the first element, two to get the second, three for the third, … up to N-1 for the last. The cost will then be approximately

1 + 2 + 3 + … + (N-2) + (N-1)

or $0.5N^2$.

13.4.3 An enhanced implementation

For most data, a basic Quicksort() works well. But there are ways of tweaking the algorithm. Many focus on tricks to pick a better value for partitioning the subarrays. It is a matter of trade offs. If you pick a better partitioning value you split the subarrays more evenly and so require fewer steps to get down to the subarrays of size one. But if it costs you a lot to find a better partitioning value then you may not gain that much from doing less partitions.

There is one tweak that is usually worthwhile when you have big arrays (tens of thousands). You combine two sort algorithms. `Quicksort()` is used as the main algorithm, but when the subarrays get small enough you switch to an alternative like selection sort. What is small enough? Probably, a value between 5 and 20 will do.

The following program illustrates this composite sort (Borland users may have to reduce the array sizes or change the "memory model" being used for the project):

```
#include <iostream.h>
#include <stdlib.h>

const int kSMALL_ENOUGH = 15;
const int kBIG = 50000; // Relying on int = long
int data[kBIG];

void SelectionSort(int data[], int left, int right)
{
    for(int i = left; i < right; i++) {
           int min = i;
           for(int j=i+1; j<= right; j++)
                  if(data[j] < data[min]) min = j;
           int temp = data[min];
           data[min] = data[i];
           data[i] = temp;
    }
}

int Partition( int d[], int left, int right)
{
    int val =d[left];
    int lm = left-1;
    int rm = right+1;
```

SelectionSort of a subarray

Partition code

```
    for(;;) {
           do
                  rm--;
           while (d[rm] > val);

           do
                  lm++;
           while( d[lm] < val);

           if(lm<rm) {
                  int tempr = d[rm];
                  d[rm] = d[lm];
                  d[lm] = tempr;
                  }
           else
                  return rm;
           }
}
```

Quicksort driver

```
void Quicksort( int d[], int left, int right)
{
    if(left < (right-kSMALL_ENOUGH)) {
           int split_pt = Partition(d,left, right);
           Quicksort(d, left, split_pt);
           Quicksort(d, split_pt+1, right);
           }
```

Call SelectionSort on smaller subarrays

```
    else SelectionSort(d, left, right);
}

int main()
{
    int i;
    long sum = 0;
```

Get some data, with lots of duplicates

```
    for(i=0;i <kBIG;i++)
           sum += data[i] = rand() % 15000;

    Quicksort(data, 0, kBIG-1);
```

Check the results!

```
    int last = -1;
    long sum2 = 0;
    for(i = 0; i < kBIG; i++)
           if(data[i] < last) {
                  cout << "Oh ....; the data aren't in order"
                                << endl;
                  cout << "Noticed the problem at i = " << i
                                << endl;
                  exit(1);
                  }
           else {
                  sum2 += last = data[i];
                  }
    if(sum != sum2) {
```

```
                cout << "Oh ...; we seem to have changed the data"
                                    << endl;
                }

        return 0;
    }
```

13.5 ALGORITHMS, FUNCTIONS, AND SUBROUTINE LIBRARIES

Searching and sorting – they are just a beginning.

For one of your more advanced computing science subjects you will get a reference book like "Introduction to Algorithms" by Cormen, Leiserson, and Rivest (MIT press). There are dozens of other books with similar titles, but the Cormen book will do (its pseudo-code for the algorithms is correct which is more than can be said for the "code" in some of the alternatives).

Searching and sorting

The Cormen book is fairly typical in content. Like most of the others, it starts with a section on topics such as recurrence relations, probability, and complexity analysis. Then it has several chapters on sorting and searching (things like binary search). The sections on "searching" explore the use of the "tree structures" introduced in Chapter 21 in Part IV of this text. Hashing algorithms, touched on briefly in Chapter 18, may also be covered in detail.

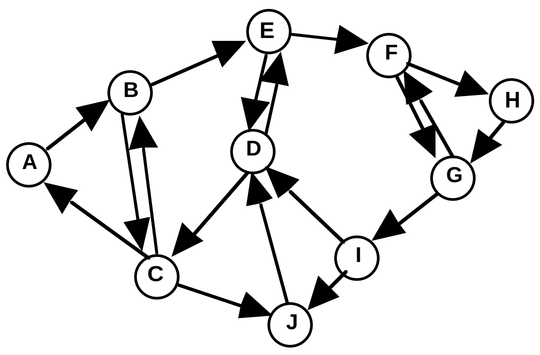
"String processing"

The algorithm books generally review "string processing". The most basic "string processing" task is "find a word in a block of text". Some "string search" algorithm has been implemented in the editor of your IDE; it does the work for the Edit/Search option. This kind of string searching is relatively simple, you just have to match a word (e.g. a misspelled keyword viod) against the text. Even so, there are a number of fairly complex algorithms for doing this; algorithms that implement tricks to minimize the work that must be done, and hence the time used to perform a search. In most IDE editors, there are more powerful search facilities (you will probably never use them). They may be called "grep" searches ("grep" was the name of an early program on Unix that provided very elaborate mechanisms for searching text in files). A "grep" search allows you to specify a pattern – "I want to find a word that starts with a capital letter, contains two adjacent o's, and does not end in 'l'". There are algorithms that define efficient ways of performing such searches.

In some texts, the string processing section also introduces algorithms for "compressing" data files.

Graph algorithms

Usually, these books contain some "Graph algorithms". These graphs are *not* x,y plots of functions. Rather, a graph represents a network – any kind of network; see Figure 13.3. A graph is made up of things (represented as "nodes" or "vertices") that are related (the "edges"). The nodes might represent cities, with the edges being connecting air-routes; or they might be atoms, with the edges being bonds; or they might be electrical components with the edges as wires. There are many properties of graphs that have to be calculated, e.g. the distances between all pairs of cities, or the shortest path, or the existence of cycles (rings) in a graph that represents a molecule. Because graphs turn up in many different practical contexts, graph algorithms have been extensively studied and there are good algorithmic solutions for many graph related problems.



Some things you might need to check:

Is the graph connected?

Can you reach all nodes from node E?

What cycles are there?

Identify all nodes you can reach in 3 steps from node G.

…

Figure 13.3 Graphs – networks of nodes and edges.

Searching, sorting, strings, and graphs; these are the main ingredients of the algorithm books. But, there are algorithms for all sorts of problems; and the books usually illustrate a variety of the more esoteric.

Bipartite graph matching

Suppose you had the task of making up a list of partners for a high school dance. As illustrated in Figure 13.4 you would have groups of boys and girls. There would also be links between pairs, each link going from a boy to a girl. Each link would indicate sufficient mutual attraction that the pair might stay together all evening. Naturally, some individuals are more attractive than others and have more links. Your task is to pair off individuals to keep them all happy.

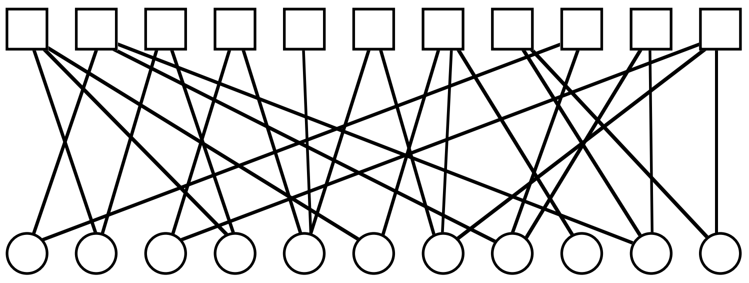


Figure 13.4 The bipartite matching (or "high school prom") problem. Find a way of matching elements from the two sets using the indicated links.

What would you do? Hack around and find your own ad hoc scheme? There is no need. This is a standard problem with a standard solution. Of course, theoreticians have given it an imposing name. This is the "bipartite graph matching" problem. There is an algorithm to solve it; but you would never know if you hadn't flicked through an algorithms book.

How about the problem in Figure 13.5? You have a network of pipes connecting a "source" (e.g. a water reservoir) to a "sink" (the sewers?). The pipes have different capacities (given in some units like cubic metres per minute). What is the maximum rate that which water might flow out of the reservoir? Again, don't roll your own. There is a standard algorithm for this one, Maximal Flow.

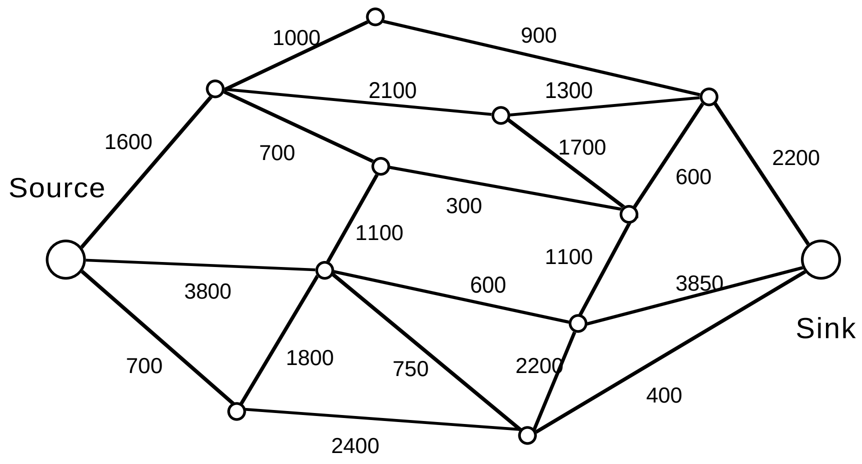


Figure 13.5 Maximal flow problem: what is the maximum flow from source to sink through the "pipes" indicated.

Figure 13.6 illustrates the "package wrapping problem" (what strange things academic computer scientists study!). In this problem, you have a large set of points (as x,y coordinate pairs). The collection is in some random order. You have to find the perimeter points.

The CACM libraries

Despite their five hundred to one thousand pages, the algorithms books are just are beginning. There are thousands more algorithms, and implementations as C functions or FORTRAN subroutines out there waiting to be used. The "Association for Computing Machinery" started publishing algorithms in its journal "Communciations of the ACM" back in the late 1950s. There is now a special ACM journal, ACM Transactions on Mathematical Software, that simply publishes details of new algorithms together with their implementation. In recent years, you could for example find how to code up "Wavelet Transform Algorithms for Finite Duration Discrete Time Signals" or a program to "Generate Niederreiter's Low Discrepancy Sequences".

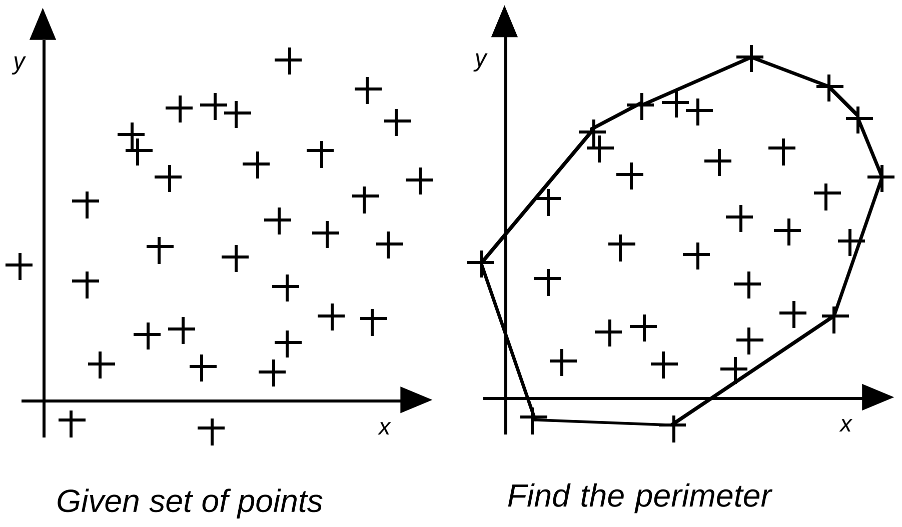


Figure 13.6 The "package wrapping" (convex hull) problem.

Numerical algorithms

Several groups, e.g. Numerical Algorithms Group, have spent years collecting algorithms and coded implementation for commonly needed things like "matrix inversion", "eigenvalues", "ordinary differential equations", "fast fourier transforms" and all the other mathematical processing requirements that turn up in scientific and engineering applications. These numerical routines are widely applicable. Some collections are commercially available; others are free. You may be able to get at the free collections via the "netlib" service at one of ATT's computers.

Discipline specific function libraries

Beyond these, each individual engineering and scientific group has established its own library of functions. Thus, "Quanutm Chemists" have functions in their QCPE library to calculate "overlap integrals" and "exchange integrals" (whatever these may be).

Surf the net

These collections of algorithms and code implementations are a resource that you need to be aware of. In later programs that you get to write, you may be able to utilize code from these collections. Many of these code libraries and similar resources are available over the World Wide Web (though you may have to search diligently to locate them).