Chapter 8. Sorting

8.0 Introduction

This chapter almost doesn't belong in a book on *numerical* methods. However, some practical knowledge of techniques for sorting is an indispensable part of any good programmer's expertise. We would not want you to consider yourself expert in numerical techniques while remaining ignorant of so basic a subject.

In conjunction with numerical work, sorting is frequently necessary when data (either experimental or numerically generated) are being handled. One has tables or lists of numbers, representing one or more independent (or "control") variables, and one or more dependent (or "measured") variables. One may wish to arrange these data, in various circumstances, in order by one or another of these variables. Alternatively, one may simply wish to identify the "median" value, or the "upper quartile" value of one of the lists of values. This task, closely related to sorting, is called *selection*.

Here, more specifically, are the tasks that this chapter will deal with:

- Sort, i.e., rearrange, an array of numbers into numerical order.
- Rearrange an array into numerical order while performing the corresponding rearrangement of one or more additional arrays, so that the correspondence between elements in all arrays is maintained.
- Given an array, prepare an *index table* for it, i.e., a table of pointers telling which number array element comes first in numerical order, which second, and so on.
- Given an array, prepare a *rank table* for it, i.e., a table telling what is the numerical rank of the first array element, the second array element, and so on.
- Select the Mth largest element from an array.

For the basic task of sorting N elements, the best algorithms require on the order of several times $N \log_2 N$ operations. The algorithm inventor tries to reduce the constant in front of this estimate to as small a value as possible. Two of the best algorithms are *Quicksort* (§8.2), invented by the inimitable C.A.R. Hoare, and *Heapsort* (§8.3), invented by J.W.J. Williams.

For large N (say > 1000), Quicksort is faster, on most machines, by a factor of 1.5 or 2; it requires a bit of extra memory, however, and is a moderately complicated program. Heapsort is a true "sort in place," and is somewhat more compact to program and therefore a bit easier to modify for special purposes. On balance, we recommend Quicksort because of its speed, but we implement both routines.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

For small N one does better to use an algorithm whose operation count goes as a higher, i.e., poorer, power of N, if the constant in front is small enough. For $N < 20$, roughly, the method of *straight insertion* (§8.1) is concise and fast enough. We include it with some trepidation: It is an N^2 algorithm, whose potential for misuse (by using it for too large an N) is great. The resultant waste of computer time is so awesome, that we were tempted not to include any N^2 routine at all. We *will* draw the line, however, at the inefficient N^2 algorithm, beloved of elementary computer science texts, called *bubble sort*. If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

For $N < 50$, roughly, *Shell's method* (§8.1), only slightly more complicated to program than straight insertion, is competitive with the more complicated Quicksort on many machines. This method goes as $N^{3/2}$ in the worst case, but is usually faster.

See references [1,2] for further information on the subject of sorting, and for detailed references to the literature.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1 Straight Insertion and Shell's Method

Straight insertion is an N^2 routine, and should be used only for small N, say < 20.

The technique is exactly the one used by experienced card players to sort their cards: Pick out the second card and put it in order with respect to the first; then pick out the third card and insert it into the sequence among the first two; and so on until the last card has been picked out and inserted.

```
void piksrt(int n, float arr[])
Sorts an array arr[1..n] into ascending numerical order, by straight insertion. n is input; arr
is replaced on output by its sorted rearrangement.
{
    int i,j;
    float a;

    for (j=2;j<=n;j++) {                      Pick out each element in turn.
        a=arr[j];
        i=j-1;
        while (i > 0 && arr[i] > a) {         Look for the place to insert it.
            arr[i+1]=arr[i];
            i--;
        }
        arr[i+1]=a;                           Insert it.
    }
}
```

What if you also want to rearrange an array `brr` at the same time as you sort `arr`? Simply move an element of `brr` whenever you move an element of `arr`:

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

For small N one does better to use an algorithm whose operation count goes as a higher, i.e., poorer, power of N, if the constant in front is small enough. For $N < 20$, roughly, the method of *straight insertion* (§8.1) is concise and fast enough. We include it with some trepidation: It is an N^2 algorithm, whose potential for misuse (by using it for too large an N) is great. The resultant waste of computer time is so awesome, that we were tempted not to include any N^2 routine at all. We *will* draw the line, however, at the inefficient N^2 algorithm, beloved of elementary computer science texts, called *bubble sort*. If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

For $N < 50$, roughly, *Shell's method* (§8.1), only slightly more complicated to program than straight insertion, is competitive with the more complicated Quicksort on many machines. This method goes as $N^{3/2}$ in the worst case, but is usually faster.

See references [1,2] for further information on the subject of sorting, and for detailed references to the literature.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1 Straight Insertion and Shell's Method

Straight insertion is an N^2 routine, and should be used only for small N, say < 20.

The technique is exactly the one used by experienced card players to sort their cards: Pick out the second card and put it in order with respect to the first; then pick out the third card and insert it into the sequence among the first two; and so on until the last card has been picked out and inserted.

```
void piksrt(int n, float arr[])
Sorts an array arr[1..n] into ascending numerical order, by straight insertion. n is input; arr
is replaced on output by its sorted rearrangement.
{
    int i,j;
    float a;

    for (j=2;j<=n;j++) {                    Pick out each element in turn.
        a=arr[j];
        i=j-1;
        while (i > 0 && arr[i] > a) {       Look for the place to insert it.
            arr[i+1]=arr[i];
            i--;
        }
        arr[i+1]=a;                         Insert it.
    }
}
```

What if you also want to rearrange an array `brr` at the same time as you sort `arr`? Simply move an element of `brr` whenever you move an element of `arr`:

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
void piksr2(int n, float arr[], float brr[])
Sorts an array arr[1..n] into ascending numerical order, by straight insertion, while making
the corresponding rearrangement of the array brr[1..n].
{
    int i,j;
    float a,b;

    for (j=2;j<=n;j++) {                    Pick out each element in turn.
        a=arr[j];
        b=brr[j];
        i=j-1;
        while (i > 0 && arr[i] > a) {       Look for the place to insert it.
            arr[i+1]=arr[i];
            brr[i+1]=brr[i];
            i--;
        }
        arr[i+1]=a;                         Insert it.
        brr[i+1]=b;
    }
}
```

For the case of rearranging a larger number of arrays by sorting on one of them, see §8.4.

Shell's Method

This is actually a variant on straight insertion, but a very powerful variant indeed. The rough idea, e.g., for the case of sorting 16 numbers $n_1 \dots n_{16}$, is this: First sort, by straight insertion, each of the 8 groups of 2 (n_1, n_9), (n_2, n_{10}), …, (n_8, n_{16}). Next, sort each of the 4 groups of 4 (n_1, n_5, n_9, n_{13}), …, $(n_4, n_8, n_{12}, n_{16})$. Next sort the 2 groups of 8 records, beginning with $(n_1, n_3, n_5, n_7, n_9, n_{11}, n_{13}, n_{15})$. Finally, sort the whole list of 16 numbers.

Of course, only the *last* sort is *necessary* for putting the numbers into order. So what is the purpose of the previous partial sorts? The answer is that the previous sorts allow numbers efficiently to filter up or down to positions close to their final resting places. Therefore, the straight insertion passes on the final sort rarely have to go past more than a "few" elements before finding the right place. (Think of sorting a hand of cards that are already almost in order.)

The spacings between the numbers sorted on each pass through the data (8,4,2,1 in the above example) are called the *increments*, and a Shell sort is sometimes called a *diminishing increment sort*. There has been a lot of research into how to choose a good set of increments, but the optimum choice is not known. The set $\dots, 8, 4, 2, 1$ is in fact not a good choice, especially for N a power of 2. A much better choice is the sequence

$$(3^k - 1)/2, \dots, 40, 13, 4, 1 \tag{8.1.1}$$

which can be generated by the recurrence

$$i_1 = 1, \qquad i_{k+1} = 3i_k + 1, \quad k = 1, 2, \dots \tag{8.1.2}$$

It can be shown (see [1]) that for this sequence of increments the number of operations required in all is of order $N^{3/2}$ for the worst possible ordering of the original data.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

For "randomly" ordered data, the operations count goes approximately as $N^{1.25}$, at least for $N < 60000$. For $N > 50$, however, Quicksort is generally faster. The program follows:

```
void shell(unsigned long n, float a[])
Sorts an array a[1..n] into ascending numerical order by Shell's method (diminishing increment
sort). n is input; a is replaced on output by its sorted rearrangement.
{
    unsigned long i,j,inc;
    float v;
    inc=1;                              Determine the starting increment.
    do {
        inc *= 3;
        inc++;
    } while (inc <= n);
    do {                                Loop over the partial sorts.
        inc /= 3;
        for (i=inc+1;i<=n;i++) {        Outer loop of straight insertion.
            v=a[i];
            j=i;
            while (a[j-inc] > v) {      Inner loop of straight insertion.
                a[j]=a[j-inc];
                j -= inc;
                if (j <= inc) break;
            }
            a[j]=v;
        }
    } while (inc > 1);
}
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 8.

8.2 Quicksort

Quicksort is, on most machines, on average, for large N, the fastest known sorting algorithm. It is a "partition-exchange" sorting method: A "partitioning element" a is selected from the array. Then by pairwise exchanges of elements, the original array is partitioned into two subarrays. At the end of a round of partitioning, the element a is in its final place in the array. All elements in the left subarray are $\leq$ a, while all elements in the right subarray are $\geq$ a. The process is then repeated on the left and right subarrays independently, and so on.

The partitioning process is carried out by selecting some element, say the leftmost, as the partitioning element a. Scan a pointer up the array until you find an element $>$ a, and then scan another pointer down from the end of the array until you find an element $<$ a. These two elements are clearly out of place for the final partitioned array, so exchange them. Continue this process until the pointers cross. This is the right place to insert a, and that round of partitioning is done. The

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

For “randomly” ordered data, the operations count goes approximately as $N^{1.25}$, at least for $N < 60000$. For $N > 50$, however, Quicksort is generally faster. The program follows:

```
void shell(unsigned long n, float a[])
Sorts an array a[1..n] into ascending numerical order by Shell's method (diminishing increment
sort). n is input; a is replaced on output by its sorted rearrangement.
{
    unsigned long i,j,inc;
    float v;
    inc=1;                              Determine the starting increment.
    do {
        inc *= 3;
        inc++;
    } while (inc <= n);
    do {                                Loop over the partial sorts.
        inc /= 3;
        for (i=inc+1;i<=n;i++) {        Outer loop of straight insertion.
            v=a[i];
            j=i;
            while (a[j-inc] > v) {      Inner loop of straight insertion.
                a[j]=a[j-inc];
                j -= inc;
                if (j <= inc) break;
            }
            a[j]=v;
        }
    } while (inc > 1);
}
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 8.

8.2 Quicksort

Quicksort is, on most machines, on average, for large N, the fastest known sorting algorithm. It is a “partition-exchange” sorting method: A “partitioning element” a is selected from the array. Then by pairwise exchanges of elements, the original array is partitioned into two subarrays. At the end of a round of partitioning, the element a is in its final place in the array. All elements in the left subarray are $\leq$ a, while all elements in the right subarray are $\geq$ a. The process is then repeated on the left and right subarrays independently, and so on.

The partitioning process is carried out by selecting some element, say the leftmost, as the partitioning element a. Scan a pointer up the array until you find an element $>$ a, and then scan another pointer down from the end of the array until you find an element $<$ a. These two elements are clearly out of place for the final partitioned array, so exchange them. Continue this process until the pointers cross. This is the right place to insert a, and that round of partitioning is done. The

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

question of the best strategy when an element is equal to the partitioning element is subtle; we refer you to Sedgewick [1] for a discussion. (Answer: You should stop and do an exchange.)

For speed of execution, we do not implement Quicksort using recursion. Thus the algorithm requires an auxiliary array of storage, of length $2\log_2 N$, which it uses as a push-down stack for keeping track of the pending subarrays. When a subarray has gotten down to some size M, it becomes faster to sort it by straight insertion (§8.1), so we will do this. The optimal setting of M is machine dependent, but $M = 7$ is not too far wrong. Some people advocate leaving the short subarrays unsorted until the end, and then doing one giant insertion sort at the end. Since each element moves at most 7 places, this is just as efficient as doing the sorts immediately, and saves on the overhead. However, on modern machines with paged memory, there is increased overhead when dealing with a large array all at once. We have not found any advantage in saving the insertion sorts till the end.

As already mentioned, Quicksort's *average* running time is fast, but its *worst case* running time can be very slow: For the worst case it is, in fact, an N^2 method! And for the most straightforward implementation of Quicksort it turns out that the worst case is achieved for an input array that is already in order! This ordering of the input array might easily occur in practice. One way to avoid this is to use a little random number generator to choose a random element as the partitioning element. Another is to use instead the median of the first, middle, and last elements of the current subarray.

The great speed of Quicksort comes from the simplicity and efficiency of its inner loop. Simply adding one unnecessary test (for example, a test that your pointer has not moved off the end of the array) can almost double the running time! One avoids such unnecessary tests by placing "sentinels" at either end of the subarray being partitioned. The leftmost sentinel is $\leq$ a, the rightmost $\geq$ a. With the "median-of-three" selection of a partitioning element, we can use the two elements that were not the median to be the sentinels for that subarray.

Our implementation closely follows [1]:

```
#include "nrutil.h"
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7
#define NSTACK 50
```
Here M is the size of subarrays sorted by straight insertion and NSTACK is the required auxiliary storage.

```
void sort(unsigned long n, float arr[])
```
Sorts an array arr[1..n] into ascending numerical order using the Quicksort algorithm. n is input; arr is replaced on output by its sorted rearrangement.
```
{
    unsigned long i,ir=n,j,k,l=1,*istack;
    int jstack=0;
    float a,temp;

    istack=lvector(1,NSTACK);
    for (;;) {                          Insertion sort when subarray small enough.
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                for (i=j-1;i>=l;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
            }
            arr[i+1]=a;
        }
        if (jstack == 0) break;
        ir=istack[jstack--];            Pop stack and begin a new round of parti-
        l=istack[jstack--];                 tioning.
    } else {
        k=(l+ir) >> 1;                  Choose median of left, center, and right el-
        SWAP(arr[k],arr[l+1])               ements as partitioning element a. Also
        if (arr[l] > arr[ir]) {             rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
            SWAP(arr[l],arr[ir])
        }
        if (arr[l+1] > arr[ir]) {
            SWAP(arr[l+1],arr[ir])
        }
        if (arr[l] > arr[l+1]) {
            SWAP(arr[l],arr[l+1])
        }
        i=l+1;                          Initialize pointers for partitioning.
        j=ir;
        a=arr[l+1];                     Partitioning element.
        for (;;) {                      Beginning of innermost loop.
            do i++; while (arr[i] < a);         Scan up to find element > a.
            do j--; while (arr[j] > a);         Scan down to find element < a.
            if (j < i) break;           Pointers crossed. Partitioning complete.
            SWAP(arr[i],arr[j]);        Exchange elements.
        }                               End of innermost loop.
        arr[l+1]=arr[j];                Insert partitioning element.
        arr[j]=a;
        jstack += 2;
        Push pointers to larger subarray on stack, process smaller subarray immediately.
        if (jstack > NSTACK) nrerror("NSTACK too small in sort.");
        if (ir-i+1 >= j-l) {
            istack[jstack]=ir;
            istack[jstack-1]=i;
            ir=j-1;
        } else {
            istack[jstack]=j-1;
            istack[jstack-1]=l;
            l=i;
        }
    }
}
free_lvector(istack,1,NSTACK);
}
```

As usual you can move any other arrays around at the same time as you sort arr. At the risk of being repetitious:

```
#include "nrutil.h"
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7
#define NSTACK 50

void sort2(unsigned long n, float arr[], float brr[])
Sorts an array arr[1..n] into ascending order using Quicksort, while making the corresponding
rearrangement of the array brr[1..n].
{
    unsigned long i,ir=n,j,k,l=1,*istack;
    int jstack=0;
    float a,b,temp;
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
istack=lvector(1,NSTACK);
for (;;) {                              Insertion sort when subarray small enough.
    if (ir-l < M) {
        for (j=l+1;j<=ir;j++) {
            a=arr[j];
            b=brr[j];
            for (i=j-1;i>=l;i--) {
                if (arr[i] <= a) break;
                arr[i+1]=arr[i];
                brr[i+1]=brr[i];
            }
            arr[i+1]=a;
            brr[i+1]=b;
        }
        if (!jstack) {
            free_lvector(istack,1,NSTACK);
            return;
        }
        ir=istack[jstack];              Pop stack and begin a new round of parti-
        l=istack[jstack-1];                 tioning.
        jstack -= 2;
    } else {
        k=(l+ir) >> 1;                  Choose median of left, center and right el-
        SWAP(arr[k],arr[l+1])               ements as partitioning element a. Also
        SWAP(brr[k],brr[l+1])               rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
        if (arr[l] > arr[ir]) {
            SWAP(arr[l],arr[ir])
            SWAP(brr[l],brr[ir])
        }
        if (arr[l+1] > arr[ir]) {
            SWAP(arr[l+1],arr[ir])
            SWAP(brr[l+1],brr[ir])
        }
        if (arr[l] > arr[l+1]) {
            SWAP(arr[l],arr[l+1])
            SWAP(brr[l],brr[l+1])
        }
        i=l+1;                          Initialize pointers for partitioning.
        j=ir;
        a=arr[l+1];                     Partitioning element.
        b=brr[l+1];
        for (;;) {                      Beginning of innermost loop.
            do i++; while (arr[i] < a);     Scan up to find element > a.
            do j--; while (arr[j] > a);     Scan down to find element < a.
            if (j < i) break;           Pointers crossed. Partitioning complete.
            SWAP(arr[i],arr[j])         Exchange elements of both arrays.
            SWAP(brr[i],brr[j])
        }                               End of innermost loop.
        arr[l+1]=arr[j];                Insert partitioning element in both arrays.
        arr[j]=a;
        brr[l+1]=brr[j];
        brr[j]=b;
        jstack += 2;
        Push pointers to larger subarray on stack, process smaller subarray immediately.
        if (jstack > NSTACK) nrerror("NSTACK too small in sort2.");
        if (ir-i+1 >= j-l) {
            istack[jstack]=ir;
            istack[jstack-1]=i;
            ir=j-1;
        } else {
            istack[jstack]=j-1;
            istack[jstack-1]=l;
            l=i;
        }
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
        }
    }
}
```

You could, in principle, rearrange any number of additional arrays along with brr, but this becomes wasteful as the number of such arrays becomes large. The preferred technique is to make use of an index table, as described in §8.4.

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers $a_i,\ i = 1, \dots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

Here the division in $j/2$ means "integer divide," i.e., is an exact integer or else is rounded down to the closest integer. Definition (8.3.1) will make sense if you think of the numbers a_i as being arranged in a binary tree, with the top, "boss," node being a_1, the two "underling" nodes being a_2 and a_3, *their* four underling nodes being a_4 through a_7, etc. (See Figure 8.3.1.) In this form, a heap has every "supervisor" greater than or equal to its two "supervisees," down through the levels of the hierarchy.

If you have managed to rearrange your array into an order that forms a heap, then sorting it is very easy: You pull off the "top of the heap," which will be the largest element yet unsorted. Then you "promote" to the top of the heap its largest underling. Then you promote *its* largest underling, and so on. The process is like what happens (or is supposed to happen) in a large corporation when the chairman of the board retires. You then repeat the whole process by retiring the new chairman of the board. Evidently the whole thing is an $N \log_2 N$ process, since each retiring chairman leads to $\log_2 N$ promotions of underlings.

Well, how do you arrange the array into a heap in the first place? The answer is again a "sift-up" process like corporate promotion. Imagine that the corporation starts out with $N/2$ employees on the production line, but with no supervisors. Now a supervisor is hired to supervise two workers. If he is less capable than one of his workers, that one is promoted in his place, and he joins the production line.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
        }
    }
}
```

You could, in principle, rearrange any number of additional arrays along with `brr`, but this becomes wasteful as the number of such arrays becomes large. The preferred technique is to make use of an index table, as described in §8.4.

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers $a_i,\ i = 1, \dots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

Here the division in $j/2$ means "integer divide," i.e., is an exact integer or else is rounded down to the closest integer. Definition (8.3.1) will make sense if you think of the numbers a_i as being arranged in a binary tree, with the top, "boss," node being a_1, the two "underling" nodes being a_2 and a_3, *their* four underling nodes being a_4 through a_7, etc. (See Figure 8.3.1.) In this form, a heap has every "supervisor" greater than or equal to its two "supervisees," down through the levels of the hierarchy.

If you have managed to rearrange your array into an order that forms a heap, then sorting it is very easy: You pull off the "top of the heap," which will be the largest element yet unsorted. Then you "promote" to the top of the heap its largest underling. Then you promote *its* largest underling, and so on. The process is like what happens (or is supposed to happen) in a large corporation when the chairman of the board retires. You then repeat the whole process by retiring the new chairman of the board. Evidently the whole thing is an $N \log_2 N$ process, since each retiring chairman leads to $\log_2 N$ promotions of underlings.

Well, how do you arrange the array into a heap in the first place? The answer is again a "sift-up" process like corporate promotion. Imagine that the corporation starts out with $N/2$ employees on the production line, but with no supervisors. Now a supervisor is hired to supervise two workers. If he is less capable than one of his workers, that one is promoted in his place, and he joins the production line.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

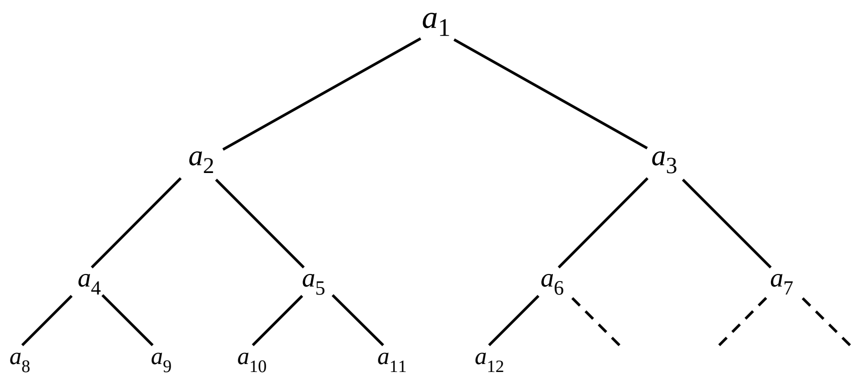

Figure 8.3.1. Ordering implied by a "heap," here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only "laterally."

After supervisors are hired, then supervisors of supervisors are hired, and so on up the corporate ladder. Each employee is brought in at the top of the tree, but then immediately sifted down, with more capable workers promoted until their proper corporate level has been reached.

In the Heapsort implementation, the same "sift-up" code can be used for the initial creation of the heap and for the subsequent retirement-and-promotion phase. One execution of the Heapsort function represents the entire life-cycle of a giant corporation: $N/2$ workers are hired; $N/2$ potential supervisors are hired; there is a sifting up in the ranks, a sort of super Peter Principle: in due course, each of the original employees gets promoted to chairman of the board.

```
void hpsort(unsigned long n, float ra[])
Sorts an array ra[1..n] into ascending numerical order using the Heapsort algorithm. n is
input; ra is replaced on output by its sorted rearrangement.
{
    unsigned long i,ir,j,l;
    float rra;

    if (n < 2) return;
    l=(n >> 1)+1;
    ir=n;
    The index l will be decremented from its initial value down to 1 during the "hiring" (heap
    creation) phase. Once it reaches 1, the index ir will be decremented from its initial value
    down to 1 during the "retirement-and-promotion" (heap selection) phase.
    for (;;) {
        if (l > 1) {                    Still in hiring phase.
            rra=ra[--l];
        } else {                        In retirement-and-promotion phase.
            rra=ra[ir];                 Clear a space at end of array.
            ra[ir]=ra[1];               Retire the top of the heap into it.
            if (--ir == 1) {            Done with the last promotion.
                ra[1]=rra;              The least competent worker of all!
                break;
            }
        }
        i=l;                            Whether in the hiring phase or promotion phase, we
        j=l+l;                              here set up to sift down element rra to its proper
        while (j <= ir) {                   level.
            if (j < ir && ra[j] < ra[j+1]) j++;   Compare to the better underling.
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
            if (rra < ra[j]) {          Demote rra.
                ra[i]=ra[j];
                i=j;
                j <<= 1;
            } else break;               Found rra's level. Terminate the sift-down.
        }
        ra[i]=rra;                      Put rra into its slot.
    }
}
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 11. [2]

8.4 Indexing and Ranking

The concept of *keys* plays a prominent role in the management of data files. A data *record* in such a file may contain several items, or *fields*. For example, a record in a file of weather observations may have fields recording time, temperature, and wind velocity. When we sort the records, we must decide which of these fields we want to be brought into sorted order. The other fields in a record just come along for the ride, and will not, in general, end up in any particular order. The field on which the sort is performed is called the *key* field.

For a data file with many records and many fields, the actual movement of N records into the sorted order of their keys $K_i,\ i = 1, \ldots, N$, can be a daunting task. Instead, one can construct an *index table* $I_j,\ j = 1, \ldots, N$, such that the smallest K_i has $i = I_1$, the second smallest has $i = I_2$, and so on up to the largest K_i with $i = I_N$. In other words, the array

$$K_{I_j} \quad j = 1, 2, \ldots, N \tag{8.4.1}$$

is in sorted order when indexed by j. When an index table is available, one need not move records from their original order. Further, different index tables can be made from the same set of records, indexing them to different keys.

The algorithm for constructing an index table is straightforward: Initialize the index array with the integers from 1 to N, then perform the Quicksort algorithm, moving the elements around *as if* one were sorting the keys. The integer that initially numbered the smallest key thus ends up in the number one position, and so on.

```
#include "nrutil.h"
#define SWAP(a,b) itemp=(a);(a)=(b);(b)=itemp;
#define M 7
#define NSTACK 50

void indexx(unsigned long n, float arr[], unsigned long indx[])
```
Indexes an array `arr[1..n]`, i.e., outputs the array `indx[1..n]` such that `arr[indx[j]]` is in ascending order for `j` $= 1, 2, \ldots, N$. The input quantities `n` and `arr` are not changed.
```
{
    unsigned long i,indxt,ir=n,itemp,j,k,l=1;
    int jstack=0,*istack;
    float a;
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
            if (rra < ra[j]) {          Demote rra.
                ra[i]=ra[j];
                i=j;
                j <<= 1;
            } else break;               Found rra's level. Terminate the sift-down.
        }
        ra[i]=rra;                      Put rra into its slot.
    }
}
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 11. [2]

8.4 Indexing and Ranking

The concept of *keys* plays a prominent role in the management of data files. A data *record* in such a file may contain several items, or *fields*. For example, a record in a file of weather observations may have fields recording time, temperature, and wind velocity. When we sort the records, we must decide which of these fields we want to be brought into sorted order. The other fields in a record just come along for the ride, and will not, in general, end up in any particular order. The field on which the sort is performed is called the *key* field.

For a data file with many records and many fields, the actual movement of N records into the sorted order of their keys $K_i,\ i = 1,\dots,N$, can be a daunting task. Instead, one can construct an *index table* $I_j,\ j = 1,\dots,N$, such that the smallest K_i has $i = I_1$, the second smallest has $i = I_2$, and so on up to the largest K_i with $i = I_N$. In other words, the array

$$K_{I_j} \quad j = 1, 2, \dots, N \tag{8.4.1}$$

is in sorted order when indexed by j. When an index table is available, one need not move records from their original order. Further, different index tables can be made from the same set of records, indexing them to different keys.

The algorithm for constructing an index table is straightforward: Initialize the index array with the integers from 1 to N, then perform the Quicksort algorithm, moving the elements around *as if* one were sorting the keys. The integer that initially numbered the smallest key thus ends up in the number one position, and so on.

```
#include "nrutil.h"
#define SWAP(a,b) itemp=(a);(a)=(b);(b)=itemp;
#define M 7
#define NSTACK 50

void indexx(unsigned long n, float arr[], unsigned long indx[])
Indexes an array arr[1..n], i.e., outputs the array indx[1..n] such that arr[indx[j]] is
in ascending order for j = 1, 2, ..., N. The input quantities n and arr are not changed.
{
    unsigned long i,indxt,ir=n,itemp,j,k,l=1;
    int jstack=0,*istack;
    float a;
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

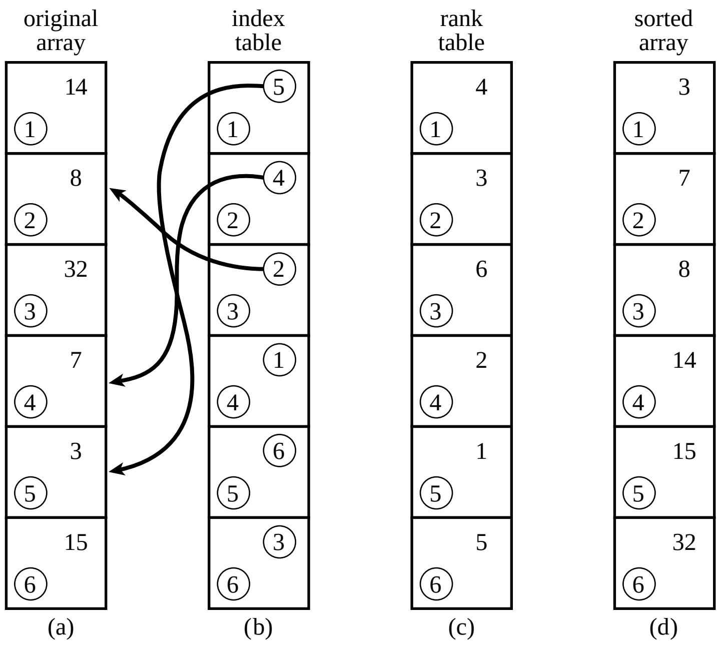

Figure 8.4.1. (a) An unsorted array of six numbers. (b) Index table, whose entries are pointers to the elements of (a) in ascending order. (c) Rank table, whose entries are the ranks of the corresponding elements of (a). (d) Sorted array of the elements in (a).

```
    istack=ivector(1,NSTACK);
    for (j=1;j<=n;j++) indx[j]=j;
    for (;;) {
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                indxt=indx[j];
                a=arr[indxt];
                for (i=j-1;i>=l;i--) {
                    if (arr[indx[i]] <= a) break;
                    indx[i+1]=indx[i];
                }
                indx[i+1]=indxt;
            }
            if (jstack == 0) break;
            ir=istack[jstack--];
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;
            SWAP(indx[k],indx[l+1]);
            if (arr[indx[l]] > arr[indx[ir]]) {
                SWAP(indx[l],indx[ir])
            }
            if (arr[indx[l+1]] > arr[indx[ir]]) {
                SWAP(indx[l+1],indx[ir])
            }
            if (arr[indx[l]] > arr[indx[l+1]]) {
                SWAP(indx[l],indx[l+1])
            }
            i=l+1;
            j=ir;
            indxt=indx[l+1];
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
            a=arr[indxt];
            for (;;) {
                do i++; while (arr[indx[i]] < a);
                do j--; while (arr[indx[j]] > a);
                if (j < i) break;
                SWAP(indx[i],indx[j])
            }
            indx[l+1]=indx[j];
            indx[j]=indxt;
            jstack += 2;
            if (jstack > NSTACK) nrerror("NSTACK too small in indexx.");
            if (ir-i+1 >= j-l) {
                istack[jstack]=ir;
                istack[jstack-1]=i;
                ir=j-1;
            } else {
                istack[jstack]=j-1;
                istack[jstack-1]=l;
                l=i;
            }
        }
    }
    free_ivector(istack,1,NSTACK);
}
```

If you want to sort an array while making the corresponding rearrangement of several or many other arrays, you should first make an index table, then use it to rearrange each array in turn. This requires two arrays of working space: one to hold the index, and another into which an array is temporarily moved, and from which it is redeposited back on itself in the rearranged order. For 3 arrays, the procedure looks like this:

```
#include "nrutil.h"

void sort3(unsigned long n, float ra[], float rb[], float rc[])
```
Sorts an array `ra[1..n]` into ascending numerical order while making the corresponding rearrangements of the arrays `rb[1..n]` and `rc[1..n]`. An index table is constructed via the routine `indexx`.
```
{
    void indexx(unsigned long n, float arr[], unsigned long indx[]);
    unsigned long j,*iwksp;
    float *wksp;

    iwksp=lvector(1,n);
    wksp=vector(1,n);
    indexx(n,ra,iwksp);                        Make the index table.
    for (j=1;j<=n;j++) wksp[j]=ra[j];          Save the array ra.
    for (j=1;j<=n;j++) ra[j]=wksp[iwksp[j]];   Copy it back in rearranged order.
    for (j=1;j<=n;j++) wksp[j]=rb[j];          Ditto rb.
    for (j=1;j<=n;j++) rb[j]=wksp[iwksp[j]];
    for (j=1;j<=n;j++) wksp[j]=rc[j];          Ditto rc.
    for (j=1;j<=n;j++) rc[j]=wksp[iwksp[j]];
    free_vector(wksp,1,n);
    free_lvector(iwksp,1,n);
}
```

The generalization to any other number of arrays is obviously straightforward.

A *rank table* is different from an index table. A rank table's jth entry gives the

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

rank of the jth element of the original array of keys, ranging from 1 (if that element was the smallest) to N (if that element was the largest). One can easily construct a rank table from an index table, however:

```
void rank(unsigned long n, unsigned long indx[], unsigned long irank[])
Given indx[1..n] as output from the routine indexx, returns an array irank[1..n], the
corresponding table of ranks.
{
    unsigned long j;

    for (j=1;j<=n;j++) irank[indx[j]]=j;
}
```

Figure 8.4.1 summarizes the concepts discussed in this section.

8.5 Selecting the Mth Largest

Selection is sorting's austere sister. (Say *that* five times quickly!) Where sorting demands the rearrangement of an entire data array, selection politely asks for a single returned value: What is the kth smallest (or, equivalently, the $m = N+1-k$th largest) element out of N elements? The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger elements to the right, and scrambling the order within each subset. This side effect is at best innocuous, at worst downright inconvenient. When the array is very long, so that making a scratch copy of it is taxing on memory, or when the computational burden of the selection is a negligible part of a larger calculation, one turns to selection algorithms without side effects, which leave the original array undisturbed. Such *in place* selection is slower than the faster selection methods by a factor of about 10. We give routines of both types, below.

The most common use of selection is in the statistical characterization of a set of data. One often wants to know the median element in an array, or the top and bottom quartile elements. When N is odd, the median is the kth element, with $k = (N+1)/2$. When N is even, statistics books define the median as the arithmetic mean of the elements $k = N/2$ and $k = N/2 + 1$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you accept such pedantry, you must perform two separate selections to find these elements. For $N > 100$ we usually define $k = N/2$ to be the median element, pedants be damned.

The fastest general method for selection, allowing rearrangement, is *partitioning,* exactly as was done in the Quicksort algorithm (§8.2). Selecting a "random" partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right. As in Quicksort, it is important to optimize the inner loop, using "sentinels" (§8.2) to minimize the number of comparisons. For sorting, one would then proceed to further partition both subsets. For selection, we can ignore one subset and attend only to the one that contains our desired kth element. Selection by partitioning thus does not need a stack of pending operations, and its operations count scales as N rather than as $N \log N$ (see [1]). Comparison with `sort` in §8.2 should make the following routine obvious:

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

rank of the jth element of the original array of keys, ranging from 1 (if that element was the smallest) to N (if that element was the largest). One can easily construct a rank table from an index table, however:

```
void rank(unsigned long n, unsigned long indx[], unsigned long irank[])
Given indx[1..n] as output from the routine indexx, returns an array irank[1..n], the
corresponding table of ranks.
{
    unsigned long j;

    for (j=1;j<=n;j++) irank[indx[j]]=j;
}
```

Figure 8.4.1 summarizes the concepts discussed in this section.

8.5 Selecting the Mth Largest

Selection is sorting's austere sister. (Say *that* five times quickly!) Where sorting demands the rearrangement of an entire data array, selection politely asks for a single returned value: What is the kth smallest (or, equivalently, the $m = N+1-k$th largest) element out of N elements? The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger elements to the right, and scrambling the order within each subset. This side effect is at best innocuous, at worst downright inconvenient. When the array is very long, so that making a scratch copy of it is taxing on memory, or when the computational burden of the selection is a negligible part of a larger calculation, one turns to selection algorithms without side effects, which leave the original array undisturbed. Such *in place* selection is slower than the faster selection methods by a factor of about 10. We give routines of both types, below.

The most common use of selection is in the statistical characterization of a set of data. One often wants to know the median element in an array, or the top and bottom quartile elements. When N is odd, the median is the kth element, with $k = (N+1)/2$. When N is even, statistics books define the median as the arithmetic mean of the elements $k = N/2$ and $k = N/2 + 1$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you accept such pedantry, you must perform two separate selections to find these elements. For $N > 100$ we usually define $k = N/2$ to be the median element, pedants be damned.

The fastest general method for selection, allowing rearrangement, is *partitioning,* exactly as was done in the Quicksort algorithm (§8.2). Selecting a "random" partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right. As in Quicksort, it is important to optimize the inner loop, using "sentinels" (§8.2) to minimize the number of comparisons. For sorting, one would then proceed to further partition both subsets. For selection, we can ignore one subset and attend only to the one that contains our desired kth element. Selection by partitioning thus does not need a stack of pending operations, and its operations count scales as N rather than as $N \log N$ (see [1]). Comparison with `sort` in §8.2 should make the following routine obvious:

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;

float select(unsigned long k, unsigned long n, float arr[])
```
Returns the kth smallest value in the array arr[1..n]. The input array will be rearranged to have this value in location arr[k], with all smaller elements moved to arr[1..k-1] (in arbitrary order) and all larger elements in arr[k+1..n] (also in arbitrary order).
```
{
    unsigned long i,ir,j,l,mid;
    float a,temp;

    l=1;
    ir=n;
    for (;;) {
        if (ir <= l+1) {                     Active partition contains 1 or 2 elements.
            if (ir == l+1 && arr[ir] < arr[l]) {      Case of 2 elements.
                SWAP(arr[l],arr[ir])
            }
            return arr[k];
        } else {
            mid=(l+ir) >> 1;                 Choose median of left, center, and right el-
            SWAP(arr[mid],arr[l+1])              ements as partitioning element a. Also
            if (arr[l] > arr[ir]) {              rearrange so that arr[l] ≤ arr[l+1],
                SWAP(arr[l],arr[ir])             arr[ir] ≥ arr[l+1].
            }
            if (arr[l+1] > arr[ir]) {
                SWAP(arr[l+1],arr[ir])
            }
            if (arr[l] > arr[l+1]) {
                SWAP(arr[l],arr[l+1])
            }
            i=l+1;                           Initialize pointers for partitioning.
            j=ir;
            a=arr[l+1];                      Partitioning element.
            for (;;) {                       Beginning of innermost loop.
                do i++; while (arr[i] < a);         Scan up to find element > a.
                do j--; while (arr[j] > a);         Scan down to find element < a.
                if (j < i) break;            Pointers crossed. Partitioning complete.
                SWAP(arr[i],arr[j])
            }                                End of innermost loop.
            arr[l+1]=arr[j];                 Insert partitioning element.
            arr[j]=a;
            if (j >= k) ir=j-1;              Keep active the partition that contains the
            if (j <= k) l=i;                     kth element.
        }
    }
}
```

In-place, nondestructive, selection is conceptually simple, but it requires a lot of bookkeeping, and it is correspondingly slower. The general idea is to pick some number M of elements at random, to sort them, and then to make a pass through the array *counting* how many elements fall in each of the $M + 1$ intervals defined by these elements. The kth largest will fall in one such interval — call it the "live" interval. One then does a second round, first picking M random elements in the live interval, and then determining which of the new, finer, $M + 1$ intervals all presently live elements fall into. And so on, until the kth element is finally localized within a single array of size M, at which point direct selection is possible.

How shall we pick M? The number of rounds, $\log_M N = \log_2 N/\log_2 M$, will be smaller if M is larger; but the work to locate each element among $M + 1$ subintervals will be larger, scaling as $\log_2 M$ for bisection, say. Each round

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

requires looking at all N elements, if only to find those that are still alive, while the bisections are dominated by the N that occur in the first round. Minimizing $O(N \log_M N) + O(N \log_2 M)$ thus yields the result

$$M \sim 2^{\sqrt{\log_2 N}} \tag{8.5.1}$$

The square root of the logarithm is so slowly varying that secondary considerations of machine timing become important. We use $M = 64$ as a convenient constant value.

Two minor additional tricks in the following routine, `selip`, are (i) augmenting the set of M random values by an $M + 1$st, the arithmetic mean, and (ii) choosing the M random values "on the fly" in a pass through the data, by a method that makes later values no less likely to be chosen than earlier ones. (The underlying idea is to give element $m > M$ an M/m chance of being brought into the set. You can prove by induction that this yields the desired result.)

```
#include "nrutil.h"
#define M 64
#define BIG 1.0e30
#define FREEALL free_vector(sel,1,M+2);free_lvector(isel,1,M+2);

float selip(unsigned long k, unsigned long n, float arr[])
Returns the kth smallest value in the array arr[1..n]. The input array is not altered.
{
    void shell(unsigned long n, float a[]);
    unsigned long i,j,jl,jm,ju,kk,mm,nlo,nxtmm,*isel;
    float ahi,alo,sum,*sel;

    if (k < 1 || k > n || n <= 0) nrerror("bad input to selip");
    isel=lvector(1,M+2);
    sel=vector(1,M+2);
    kk=k;
    ahi=BIG;
    alo = -BIG;
    for (;;) {                              Main iteration loop, until desired ele-
        mm=nlo=0;                               ment is isolated.
        sum=0.0;
        nxtmm=M+1;
        for (i=1;i<=n;i++) {                Make a pass through the whole array.
            if (arr[i] >= alo && arr[i] <= ahi) {
                Consider only elements in the current brackets.
                mm++;
                if (arr[i] == alo) nlo++;       In case of ties for low bracket.
                Now use statistical procedure for selecting m in-range elements with equal
                probability, even without knowing in advance how many there are!
                if (mm <= M) sel[mm]=arr[i];
                else if (mm == nxtmm) {
                    nxtmm=mm+mm/M;
                    sel[1 + ((i+mm+kk) % M)]=arr[i];  The % operation provides a some-
                }                                         what random number.
                sum += arr[i];
            }
        }
        if (kk <= nlo) {                    Desired element is tied for lower bound;
            FREEALL                             return it.
            return alo;
        }
        else if (mm <= M) {                 All in-range elements were kept. So re-
            shell(mm,sel);                      turn answer by direct method.
            ahi = sel[kk];
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
            FREEALL
            return ahi;
        }
        sel[M+1]=sum/mm;                        Augment selected set by mean value (fixes
        shell(M+1,sel);                              degeneracies), and sort it.
        sel[M+2]=ahi;
        for (j=1;j<=M+2;j++) isel[j]=0;         Zero the count array.
        for (i=1;i<=n;i++) {                    Make another pass through the array.
            if (arr[i] >= alo && arr[i] <= ahi) {          For each in-range element..
                jl=0;
                ju=M+2;
                while (ju-jl > 1) {             ...find its position among the select by
                    jm=(ju+jl)/2;                    bisection...
                    if (arr[i] >= sel[jm]) jl=jm;
                    else ju=jm;
                }
                isel[ju]++;                     ...and increment the counter.
            }
        }
        j=1;                                    Now we can narrow the bounds to just
        while (kk > isel[j]) {                       one bin, that is, by a factor of order
            alo=sel[j];                              m.
            kk -= isel[j++];
        }
        ahi=sel[j];
    }
}
```

Approximate timings: `selip` is about 10 times slower than `select`. Indeed, for N in the range of $\sim 10^5$, `selip` is about 1.5 times slower than a full sort with `sort`, while `select` is about 6 times faster than `sort`. You should weigh time against memory and convenience carefully.

Of course neither of the above routines should be used for the trivial cases of finding the largest, or smallest, element in an array. Those cases, you code by hand as simple `for` loops. There are also good ways to code the case where k is modest in comparison to N, so that extra memory of order k is not burdensome. An example is to use the method of Heapsort (§8.3) to make a single pass through an array of length N while saving the m *largest* elements. The advantage of the heap structure is that only $\log m$, rather than m, comparisons are required every time a new element is added to the candidate list. This becomes a real savings when $m > O(\sqrt{N})$, but it never hurts otherwise and is easy to code. The following program gives the idea.

```
void hpsel(unsigned long m, unsigned long n, float arr[], float heap[])
```
Returns in `heap[1..m]` the largest `m` elements of the array `arr[1..n]`, with `heap[1]` guaranteed to be the the `m`th largest element. The array `arr` is not altered. For efficiency, this routine should be used only when $m \ll n$.
```
{
    void sort(unsigned long n, float arr[]);
    void nrerror(char error_text[]);
    unsigned long i,j,k;
    float swap;

    if (m > n/2 || m < 1) nrerror("probable misuse of hpsel");
    for (i=1;i<=m;i++) heap[i]=arr[i];
    sort(m,heap);                     Create initial heap by overkill! We assume m ≪ n.
    for (i=m+1;i<=n;i++) {            For each remaining element...
        if (arr[i] > heap[1]) {       Put it on the heap?
            heap[1]=arr[i];
            for (j=1;;) {             Sift down.
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
                k=j << 1;
                if (k > m) break;
                if (k != m && heap[k] > heap[k+1]) k++;
                if (heap[j] <= heap[k]) break;
                swap=heap[k];
                heap[k]=heap[j];
                heap[j]=swap;
                j=k;
            }
        }
    }
}
```

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

8.6 Determination of Equivalence Classes

A number of techniques for sorting and searching relate to data structures whose details are beyond the scope of this book, for example, trees, linked lists, etc. These structures and their manipulations are the bread and butter of computer science, as distinct from numerical analysis, and there is no shortage of books on the subject.

In working with experimental data, we have found that one particular such manipulation, namely the determination of equivalence classes, arises sufficiently often to justify inclusion here.

The problem is this: There are N "elements" (or "data points" or whatever), numbered $1, \ldots, N$. You are given pairwise information about whether elements are in the same *equivalence class* of "sameness," by whatever criterion happens to be of interest. For example, you may have a list of facts like: "Element 3 and element 7 are in the same class; element 19 and element 4 are in the same class; element 7 and element 12 are in the same class, ..." Alternatively, you may have a procedure, given the numbers of two elements j and k, for deciding whether they are in the same class or different classes. (Recall that an equivalence relation can be anything satisfying the *RST properties*: reflexive, symmetric, transitive. This is compatible with any intuitive definition of "sameness.")

The desired output is an assignment to each of the N elements of an equivalence class number, such that two elements are in the same class if and only if they are assigned the same class number.

Efficient algorithms work like this: Let $F(j)$ be the class or "family" number of element j. Start off with each element in its own family, so that $F(j) = j$. The array $F(j)$ can be interpreted as a tree structure, where $F(j)$ denotes the parent of j. If we arrange for each family to be its own tree, disjoint from all the other "family trees," then we can label each family (equivalence class) by its most senior great-great-...grandparent. The detailed topology of the tree doesn't matter at all, as long as we graft each related element onto it *somewhere*.

Therefore, we process each elemental datum "j is equivalent to k" by (i) tracking j up to its highest ancestor, (ii) tracking k up to its highest ancestor, (iii) giving j to k as a new parent, or vice versa (it makes no difference). After processing all the relations, we go through all the elements j and reset their $F(j)$'s to their highest possible ancestors, which then label the equivalence classes.

The following routine, based on Knuth [1], assumes that there are `m` elemental pieces of information, stored in two arrays of length `m`, `lista,listb`, the interpretation being that `lista[j]` and `listb[j]`, `j=1...m`, are the numbers of two elements which (we are thus told) are related.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
                k=j << 1;
                if (k > m) break;
                if (k != m && heap[k] > heap[k+1]) k++;
                if (heap[j] <= heap[k]) break;
                swap=heap[k];
                heap[k]=heap[j];
                heap[j]=swap;
                j=k;
            }
        }
    }
}
```

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

8.6 Determination of Equivalence Classes

A number of techniques for sorting and searching relate to data structures whose details are beyond the scope of this book, for example, trees, linked lists, etc. These structures and their manipulations are the bread and butter of computer science, as distinct from numerical analysis, and there is no shortage of books on the subject.

In working with experimental data, we have found that one particular such manipulation, namely the determination of equivalence classes, arises sufficiently often to justify inclusion here.

The problem is this: There are N "elements" (or "data points" or whatever), numbered $1, \dots, N$. You are given pairwise information about whether elements are in the same *equivalence class* of "sameness," by whatever criterion happens to be of interest. For example, you may have a list of facts like: "Element 3 and element 7 are in the same class; element 19 and element 4 are in the same class; element 7 and element 12 are in the same class, ..." Alternatively, you may have a procedure, given the numbers of two elements j and k, for deciding whether they are in the same class or different classes. (Recall that an equivalence relation can be anything satisfying the *RST properties*: reflexive, symmetric, transitive. This is compatible with any intuitive definition of "sameness.")

The desired output is an assignment to each of the N elements of an equivalence class number, such that two elements are in the same class if and only if they are assigned the same class number.

Efficient algorithms work like this: Let $F(j)$ be the class or "family" number of element j. Start off with each element in its own family, so that $F(j) = j$. The array $F(j)$ can be interpreted as a tree structure, where $F(j)$ denotes the parent of j. If we arrange for each family to be its own tree, disjoint from all the other "family trees," then we can label each family (equivalence class) by its most senior great-great-. . .grandparent. The detailed topology of the tree doesn't matter at all, as long as we graft each related element onto it *somewhere*.

Therefore, we process each elemental datum "j is equivalent to k" by (i) tracking j up to its highest ancestor, (ii) tracking k up to its highest ancestor, (iii) giving j to k as a new parent, or vice versa (it makes no difference). After processing all the relations, we go through all the elements j and reset their $F(j)$'s to their highest possible ancestors, which then label the equivalence classes.

The following routine, based on Knuth [1], assumes that there are `m` elemental pieces of information, stored in two arrays of length `m`, `lista,listb`, the interpretation being that `lista[j]` and `listb[j]`, `j=1...m`, are the numbers of two elements which (we are thus told) are related.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
void eclass(int nf[], int n, int lista[], int listb[], int m)
```
Given m equivalences between pairs of n individual elements in the form of the input arrays lista[1..m] and listb[1..m], this routine returns in nf[1..n] the number of the equivalence class of each of the n elements, integers between 1 and n (not all such integers used).
```
{
    int l,k,j;

    for (k=1;k<=n;k++) nf[k]=k;              Initialize each element its own class.
    for (l=1;l<=m;l++) {                     For each piece of input information...
        j=lista[l];
        while (nf[j] != j) j=nf[j];          Track first element up to its ancestor.
        k=listb[l];
        while (nf[k] != k) k=nf[k];          Track second element up to its ancestor.
        if (j != k) nf[j]=k;                 If they are not already related, make them
    }                                            so.
    for (j=1;j<=n;j++)                       Final sweep up to highest ancestors.
        while (nf[j] != nf[nf[j]]) nf[j]=nf[nf[j]];
}
```

Alternatively, we may be able to construct a function equiv(j,k) that returns a nonzero (true) value if elements j and k are related, or a zero (false) value if they are not. Then we want to loop over all pairs of elements to get the complete picture. D. Eardley has devised a clever way of doing this while simultaneously sweeping the tree up to high ancestors in a manner that keeps it current and obviates most of the final sweep phase:

```
void eclazz(int nf[], int n, int (*equiv)(int, int))
```
Given a user-supplied boolean function equiv which tells whether a pair of elements, each in the range 1...n, are related, return in nf[1..n] equivalence class numbers for each element.
```
{
    int kk,jj;

    nf[1]=1;
    for (jj=2;jj<=n;jj++) {                  Loop over first element of all pairs.
        nf[jj]=jj;
        for (kk=1;kk<=(jj-1);kk++) {         Loop over second element of all pairs.
            nf[kk]=nf[nf[kk]];               Sweep it up this much.
            if ((*equiv)(jj,kk)) nf[nf[nf[kk]]]=jj;
            Good exercise for the reader to figure out why this much ancestry is necessary!
        }
    }
    for (jj=1;jj<=n;jj++) nf[jj]=nf[nf[jj]];     Only this much sweeping is needed
}                                                    finally.
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.3.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 30.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).