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:

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

float select(unsigned long k, unsigned long n, float arr[])
```

Returns the k th 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 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];
```

```
            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.
```

```
            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.