CHAPTER 8

Sorting and Selection

8.0 Introduction

This chapter almost doesn't belong in a book on *numerical* methods: Sorting and selection are bread-and-butter topics in the standard computer science curriculum. However, some review of the techniques for sorting, from the perspective of scientific computing, will prove useful in subsequent chapters. We can develop some standard interfaces for later use, and also illustrate the usefulness of *templates* in object-oriented programming.

In conjunction with numerical work, sorting is frequently necessary when data (either experimental or numerically generated) are being processed. 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. (These kinds of values are generically called *quantiles*.) 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.
- Select the Mth largest value, or estimate arbitrary quantile values, from a data stream in one pass (i.e., without storing the stream for later processing).
- Given a bunch of equivalence relations, organize them into equivalence classes.

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.

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 resource can be 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 Refs. [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. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).[1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd 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.

sort.h

```
template<class T>
void piksrt(NRvector<T> &arr)
```

Sort an array `arr[0..n-1]` into ascending numerical order, by straight insertion. `arr` is replaced on output by its sorted rearrangement.

```
{
    Int i,j,n=arr.size();
    T a;
    for (j=1;j<n;j++) {
        a=arr[j];
```

Pick out each element in turn.

```
        i=j;
        while (i > 0 && arr[i-1] > a) {     Look for the place to insert it.
            arr[i]=arr[i-1];
            i--;
        }
        arr[i]=a;                           Insert it.
    }
}
```

Notice the use of a template in order to make the routine general for any type of `NRvector`, including both `VecInt` and `VecDoub`. The only thing required of the elements of type `T` in the vector is that they have an assignment operator and a > relation. (We will generally assume that the relations <, >, and == all exist.) If you try to sort a vector of elements without these properties, the compiler will complain, so you can't go wrong.

It is a matter of taste whether to template on the element type, as above, or on the vector itself, as

```
template<class T>
void piksrt(T &arr)
```

This would seem more general, since it will work for any type `T` that has a subscript operator `[]`, not just `NRvectors`. However, it also requires that `T` have some method for getting at the type of its elements, necessary for the declaration of the variable `a`. If `T` follows the conventions of STL containers, then that declaration can be written

```
T::value_type a;
```

but if it doesn't, then you can find yourself lost at C.

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`:

sort.h

```
template<class T, class U>
void piksr2(NRvector<T> &arr, NRvector<U> &brr)
```

Sort an array `arr[0..n-1]` into ascending numerical order, by straight insertion, while making the corresponding rearrangement of the array `brr[0..n-1]`.

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

Note that the types of `arr` and `brr` are separately templated, so they don't have to be the same.

Don't generalize this technique to the rearrangement of a larger number of arrays by sorting on one of them. Instead see §8.4.

8.1.1 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 $n_0 \ldots n_{15}$, is this: First sort, by straight insertion, each of the 8 groups of 2 (n_0, n_8), (n_1, n_9), $\ldots$, (n_7, n_{15}). Next, sort each of the 4 groups of 4 (n_0, n_4, n_8, n_{12}), $\ldots$, $(n_3, n_7, n_{11}, n_{15})$. Next sort the 2 groups of 8 records, beginning with $(n_0, n_2, n_4, n_6, n_8, n_{10}, n_{12}, n_{14})$. 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 $\ldots, 8, 4, 2, 1$ is in fact not a good choice, especially for N a power of 2. A much better choice is the sequence

$$(3^k - 1)/2, \ldots, 40, 13, 4, 1 \qquad (8.1.1)$$

which can be generated by the recurrence

$$i_0 = 1, \qquad i_{k+1} = 3i_k + 1, \quad k = 0, 1, \ldots \qquad (8.1.2)$$

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. 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.

sort.h

```
template<class T>
void shell(NRvector<T> &a, Int m=-1)
```

Sort an array `a[0..n-1]` into ascending numerical order by Shell's method (diminishing increment sort). `a` is replaced on output by its sorted rearrangement. Normally, the optional argument `m` should be omitted, but if it is set to a positive value, then only the first `m` elements of `a` are sorted.

```
{
    Int i,j,inc,n=a.size();
    T v;
    if (m>0) n = MIN(m,n);                  Use optional argument.
    inc=1;                                  Determine the starting increment.
    do {
        inc *= 3;
        inc++;
    } while (inc <= n);
    do {                                    Loop over the partial sorts.
        inc /= 3;
        for (i=inc;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. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1.[1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd 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 question of the best strategy when an element is equal to the partitioning element is subtle; see Sedgewick [1] for a discussion. (Answer: You should stop and do an exchange.)

For speed of execution, we don't 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 seven places, this is just as efficient as doing the sorts immediately, and saves on the overhead. However, on modern machines with a cache hierarchy, 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]:

sort.h

```
template<class T>
void sort(NRvector<T> &arr, Int m=-1)
```

Sort an array `arr[0..n-1]` into ascending numerical order using the Quicksort algorithm. `arr` is replaced on output by its sorted rearrangement. Normally, the optional argument `m` should be omitted, but if it is set to a positive value, then only the first `m` elements of `arr` are sorted.

```
{
    static const Int M=7, NSTACK=64;
```

Here `M` is the size of subarrays sorted by straight insertion and `NSTACK` is the required auxiliary storage.

```
    Int i,ir,j,k,jstack=-1,l=0,n=arr.size();
    T a;
    VecInt istack(NSTACK);
    if (m>0) n = MIN(m,n);                  Use optional argument.
    ir=n-1;
    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];
                }
                arr[i+1]=a;
            }
            if (jstack < 0) break;
            ir=istack[jstack--];            Pop stack and begin a new round of partitioning.
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;                  Choose median of left, center, and right elements as partitioning element a. Also rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
            SWAP(arr[k],arr[l+1]);
            if (arr[l] > arr[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) throw("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;
            }
        }
    }
}
```

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

sort.h

```
template<class T, class U>
void sort2(NRvector<T> &arr, NRvector<U> &brr)
```

Sort an array `arr[0..n-1]` into ascending order using Quicksort, while making the corresponding rearrangement of the array `brr[0..n-1]`.

```
{
    const Int M=7,NSTACK=64;
    Int i,ir,j,k,jstack=-1,l=0,n=arr.size();
    T a;
    U b;
    VecInt istack(NSTACK);
    ir=n-1;
    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 < 0) break;
            ir=istack[jstack--];                Pop stack and begin a new round of partitioning.
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;                      Choose median of left, center, and right elements as partitioning element a. Also rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
            SWAP(arr[k],arr[l+1]);
            SWAP(brr[k],brr[l+1]);
            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) throw("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;
            }
        }
    }
}
```

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

CITED REFERENCES AND FURTHER READING:

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

8.3 Heapsort

Heapsort is slower than Quicksort by a constant factor. It is so beautiful that we sometimes use it anyway, just for the sheer joy of it. (However, we don't recommend that you do this if your employer is paying for efficient code.) Heapsort is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ algorithm, not only on average, but also for the worst case order of input data. In fact, its worst case is only 20 % or so worse than its average running time.

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

A set of N numbers a_j, $j = 0, \ldots, N-1$, is said to form a "heap" if it satisfies the relation

$$a_{(j-1)/2} \geq a_j \quad \text{for} \quad 0 \leq (j-1)/2 < j < N \tag{8.3.1}$$

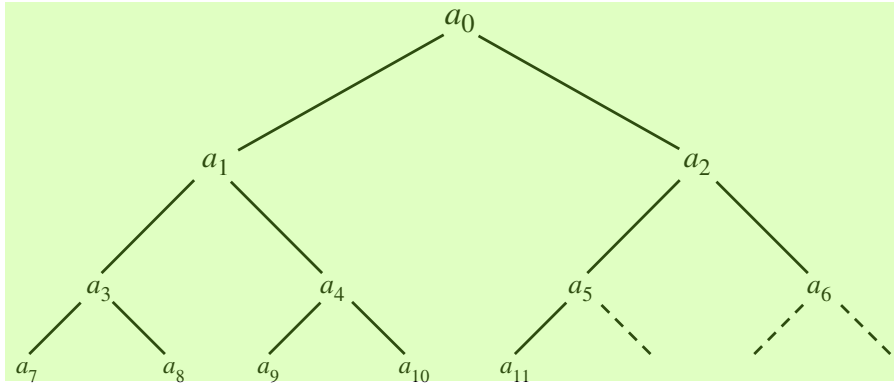

Figure 8.3.1. Ordering implied by a "heap," here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only "laterally."

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_0; the two "underling" nodes being a_1 and a_2; *their* four underling nodes being a_3 through a_6; 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. 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.

sort.h

```
namespace hpsort_util {
    template<class T>
    void sift_down(NRvector<T> &ra, const Int l, const Int r)
    Carry out the sift-down on element ra(l) to maintain the heap structure. l and r determine
    the "left" and "right" range of the sift-down.
    {
        Int j,jold;
        T a;
        a=ra[l];
        jold=l;
        j=2*l+1;
        while (j <= r) {
            if (j < r && ra[j] < ra[j+1]) j++;      Compare to the better underling.
            if (a >= ra[j]) break;                  Found a's level. Terminate the sift-
            ra[jold]=ra[j];                             down. Otherwise, demote a and
            jold=j;                                     continue.
            j=2*j+1;
        }
        ra[jold]=a;                                 Put a into its slot.
    }
}

template<class T>
void hpsort(NRvector<T> &ra)
Sort an array ra[0..n-1] into ascending numerical order using the Heapsort algorithm. ra is
replaced on output by its sorted rearrangement.
{
    Int i,n=ra.size();
    for (i=n/2-1; i>=0; i--)
        The index i, which here determines the "left" range of the sift-down, i.e., the element
        to be sifted down, is decremented from n/2-1 down to 0 during the "hiring" (heap
        creation) phase.
        hpsort_util::sift_down(ra,i,n-1);
    for (i=n-1; i>0; i--) {
        Here the "right" range of the sift-down is decremented from n-2 down to 0 during the
        "retirement-and-promotion" (heap selection) phase.
        SWAP(ra[0],ra[i]);                  Clear a space at the end of the array, and retire
        hpsort_util::sift_down(ra,0,i-1);   the top of the heap into it.
    }
}
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3.[1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd 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, $i = 0, \ldots, N-1$, can be a daunting task. Instead, one can construct an *index table* I_j, $j = 0, \ldots, N-1$, such that the smallest K_i has $i = I_0$, the second smallest has $i = I_1$, and so on up to the largest K_i with $i = I_{N-1}$. In other words, the array

$$K_{I_j} \quad j = 0, 1, \ldots, N-1 \tag{8.4.1}$$

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 0 to $N-1$, 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.

The concept of an index table maps particularly nicely into an object, say `Indexx`. The constructor takes a vector `arr` as its argument; it stores an index table to `arr`, leaving `arr` unmodified. Subsequently, the method `sort` can be invoked to rearrange `arr`, or any other vector, into the sorted order of `arr`. `Indexx` is not a templated class, since the stored index table does not depend on the type of vector that is indexed. However, it does need a templated constructor.

sort.h

```
struct Indexx {
    Int n;
    VecInt indx;

    template<class T> Indexx(const NRvector<T> &arr) {
    Constructor. Calls index and stores an index to the array arr[0..n-1].
        index(&arr[0],arr.size());
    }
    Indexx() {}                          Empty constructor. See text.

    template<class T> void sort(NRvector<T> &brr) {
    Sort an array brr[0..n-1] into the order defined by the stored index. brr is replaced on
    output by its sorted rearrangement.
        if (brr.size() != n) throw("bad size in Index sort");
        NRvector<T> tmp(brr);
        for (Int j=0;j<n;j++) brr[j] = tmp[indx[j]];
    }

    template<class T> inline const T & el(NRvector<T> &brr, Int j) const {
    This function, and the next, return the element of brr that would be in sorted position j
    according to the stored index. The vector brr is not changed.
        return brr[indx[j]];
    }
    template<class T> inline T & el(NRvector<T> &brr, Int j) {
    Same, but return an l-value.
        return brr[indx[j]];
    }

    template<class T> void index(const T *arr, Int nn);
    This does the actual work of indexing. Normally not called directly by the user, but see
    text for exceptions.

    void rank(VecInt_O &irank) {
    Returns a rank table, whose jth element is the rank of arr[j], where arr is the vector
    originally indexed. The smallest arr[j] has rank 0.
        irank.resize(n);
```

```
        for (Int j=0;j<n;j++) irank[indx[j]] = j;
    }

};

template<class T>
void Indexx::index(const T *arr, Int nn)
```

Indexes an array `arr[0..nn-1]`, i.e., resizes and sets `indx[0..nn-1]` such that `arr[indx[j]]` is in ascending order for $j = 0, 1, \ldots, nn-1$. Also sets member value `n`. The input array `arr` is not changed.

```
{
    const Int M=7,NSTACK=64;
    Int i,indxt,ir,j,k,jstack=-1,l=0;
    T a;
    VecInt istack(NSTACK);
    n = nn;
    indx.resize(n);
    ir=n-1;
    for (j=0;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];
            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) throw("NSTACK too small in index.");
            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;
            }
        }
    }
}
```

A typical use of `Indexx` might be to rearrange three vectors (not necessarily of the same type) into the sorted order defined by one of them:

```
Indexx arrindex(arr);
arrindex.sort(arr);
arrindex.sort(brr);
arrindex.sort(crr);
```

The generalization to any other number of arrays is obvious.

The `Indexx` object also provides a method `el` (for "element") to access any vector in `arr`-sorted order without actually modifying that vector (or, for that matter, `arr`). In other words, after we index `arr`, say by

```
Indexx arrindex(arr);
```

we can address an element in `brr` that corresponds to the `j`th element of a *virtually sorted* `arr` as simply `arrindex.el(brr,j)`. Neither `arr` nor `brr` are altered from their original state. `el` is provided in two versions, so that it can be both an l-value (on the left-hand side of an assignment) and an r-value (in an expression).

As an aside, the reason that the internal workhorse `index` uses a pointer, not a vector, for its argument is so that it can be used (purists would say misused) in other situations, such as indexing one row in a matrix. That is also the reason for providing an additional, empty, constructor. If you want to index `nn` consecutive elements sitting around somewhere, pointed to by `ptr`, you write

```
Indexx myhack;
myhack.index(ptr,nn);
```

A *rank table* is different from an index table. A rank table's jth entry gives the rank of the jth element of the original array of keys, ranging from 0 (if that element was the smallest) to $N-1$ (if that element was the largest). One can easily construct a rank table from an index table. Indeed, you might already have noticed the method `rank` in `Indexx` that returns just such a table, stored as a vector.

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? (In this convention, used throughout this section, k takes on values $k = 0, 1, \ldots, N-1$, so $k = 0$ is the smallest array element and $k = N-1$ the largest.) The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements

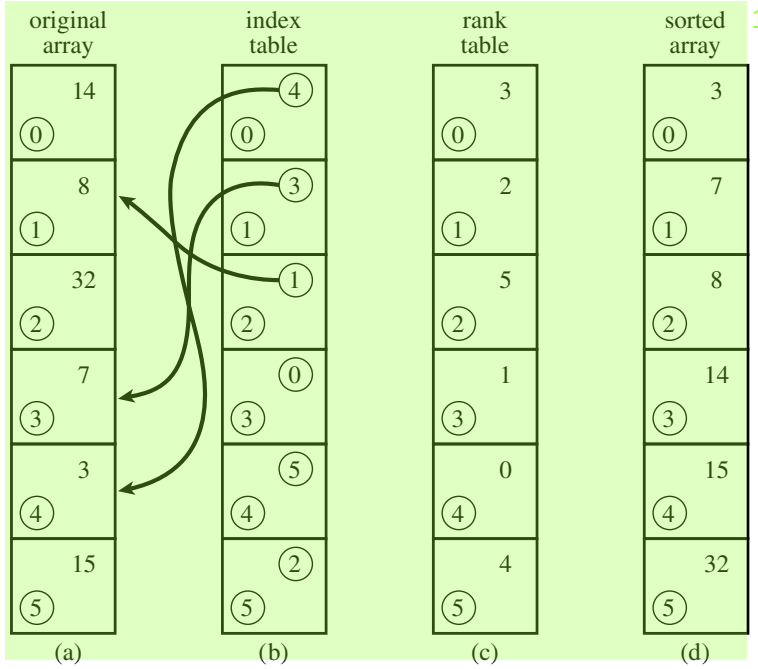

Figure 8.4.1. (a) An unsorted array of six numbers. (b) Index table whose entries are pointers to the elements of (a) in ascending order. (c) Rank table whose entries are the ranks of the corresponding elements of (a). (d) Sorted array of the elements in (a).

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 an array is very long, so that making a scratch copy of it is taxing on memory, one turns to *in-place* algorithms without side effects, which are slower but leave the original array undisturbed.

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 (quantile $p = 1/2$) or the top and bottom quartile elements (quantile $p = 1/4, 3/4$). When N is odd, the exact definition of the median is that it 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 $k = N/2 - 1$ and $k = N/2$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you embrace such formality, you must perform two separate selections to find these elements. (If you do the first selection by a partition method, see below, you can do the second by a single pass through $N/2$ elements in the right partition, looking for the smallest element.) For $N > 100$ we usually use $k = N/2$ as the median element, formalists be damned.

A variant on selection for large data sets is *single-pass selection*, where we have a stream of input values, each of which we get to see only once. We want to be able to report at any time, say after N values, the kth smallest (or largest) value seen so far, or, equivalently, the quantile value for some p. We will describe two approaches: If we care only about the smallest (or largest) M values, for fixed M, so that $0 \le k < M$, then there are good algorithms that require only M storage. On the other hand, if we can tolerate an approximate answer, then there are efficient

algorithms that can report at any time a good *estimate* of the p-quantile value for any p, $0 < p < 1$. That is to say, we will get not the exact kth smallest element, $k = pN$, of the N that have gone by, but something very close to it — and without requiring N storage or having to know p in advance.

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.

sort.h

```
template<class T>
T select(const Int k, NRvector<T> &arr)
Given k in [0..n-1] returns an array value from arr[0..n-1] such that k array values are
less than or equal to the one returned. The input array will be rearranged to have this value in
location arr[k], with all smaller elements moved to arr[0..k-1] (in arbitrary order) and all
larger elements in arr[k+1..n-1] (also in arbitrary order).
{
    Int i,ir,j,l,mid,n=arr.size();
    T a;
    l=0;
    ir=n-1;
    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.
        }
    }
}
```

If you don't want your array `arr` to be rearranged, then you will want to make

a scratch copy before calling `select`, e.g.,

```
VecDoub brr(arr);
```

The reason for not doing this internally in `select` is because you may wish to call `select` with a variety of different values `k`, and it would be wasteful to copy the vector anew each time; instead, just let `brr` keep getting rearranged.

8.5.1 Tracking the M Largest in a Single Pass

Of course `select` should not 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 efficient ways to code the case where k is bounded by some fixed M, modest in comparison to N, so that memory of order M is not burdensome. Indeed, N may not even be known: You may have a stream of incoming data values and be called upon at any time to provide a list of the M largest values seen so far.

A good approach to this case is to use the method of Heapsort (§8.3), maintaining a heap of the M largest values. The advantage of the heap structure, as opposed to a linear array of length M, is that at most $\log M$, rather than M, operations are required every time a new data value is processed.

The object `Heapselect` has a constructor, by which you specify M, an "`add`" method that assimilates a new data value, and a "`report`" method for getting the kth largest seen so far. Note that the initial cost of a report is $O(M \log M)$, because we need to sort the heap; but you can then get all values of k at no extra cost, until you do the next `add`. A special case is that getting the $M-1$st largest is always cheap, since it is at the top of the heap; so if you have a single favorite value of k, it is best to choose M with $M-1=k$.

sort.h

```
struct Heapselect {
Object for tracking the m largest values seen thus far in a stream of values.
    Int m,n,srtd;
    VecDoub heap;

    Heapselect(Int mm) : m(mm), n(0), srtd(0), heap(mm,1.e99) {}
    Constructor. The argument is the number of largest values to track.

    void add(Doub val) {
    Assimilate a new value from the stream.
        Int j,k;
        if (n<m) {                          Heap not yet filled.
            heap[n++] = val;
            if (n==m) sort(heap);           Create initial heap by overkill!
        } else {
            if (val > heap[0]) {            Put it on the heap?
                heap[0]=val;
                for (j=0;;) {               Sift down.
                    k=(j << 1) + 1;
                    if (k > m-1) break;
                    if (k != (m-1) && heap[k] > heap[k+1]) k++;
                    if (heap[j] <= heap[k]) break;
                    SWAP(heap[k],heap[j]);
                    j=k;
                }
            }
            n++;
        }
        srtd = 0;                           Mark heap as "unsorted".
    }
```

```
    Doub report(Int k) {
        Return the kth largest value seen so far. k=0 returns the largest value seen, k=1 the second
        largest, ... , k=m-1 the last position tracked. Also, k must be less than the number of
        previous values assimilated.
        Int mm = MIN(n,m);
        if (k > mm-1) throw("Heapselect k too big");
        if (k == m-1) return heap[0];          Always free, since top of heap.
        if (! srtd) { sort(heap); srtd = 1; } Otherwise, need to sort the heap.
        return heap[mm-1-k];
    }
};
```

8.5.2 Single-Pass Estimation of Arbitrary Quantiles

The data values fly by in a stream. You get to look at each value only once, and do a constant-time process on it (meaning that you can't take longer and longer to process later and later data values). Also, you have only a fixed amount of storage memory. From time to time you want to know the median value (or 95th percentile value, or arbitrary p-quantile value) of the data that you have seen thus far. How do you do this?

Evidently, with the conditions stated, you'll have to tolerate an approximate answer, since an exact answer must require unbounded storage and (perhaps) unlimited processing. If you think that "binning" is somehow the answer, you are right. But it is not immediately obvious how to choose the bins, since you have to see a potentially unlimited amount of data before you can tell for sure how its values are distributed.

Chambers et al. [2] have given a robust, and extremely fast, algorithm, which they call *IQ agent*, that adaptively adjusts a set of bins so that they converge to the data values of specified quantile p-values. The general idea (see Figure 8.5.1) is to accumulate incoming data into batches, then to update a stored, piecewise linear, cumulative distribution function (cdf) by adding a batch's cdf and then interpolating back to a fixed set of p-values. Arbitrary requested quantile values ("incremental quantiles," or "IQs," hence the algorithm's name) can be obtained at any time by linear interpolation on the stored cdf. Batching allows the program to be very efficient, with an (amortized) cost of only a small number of operations per new data value. The batching is done transparently to the user.

Similar to `Heapselect`, the `IQagent` object has `add` and `report` methods, the latter now taking a value for p as its argument. In the implementation below, we use a batch size of `nbuf=1000` but do an early update step with a partial batch whenever a quantile is requested. With these parameters, you should therefore request quantile information no more frequently than after every few `nbuf` data values, at which point you can request as many different values of p as you want before continuing. The alternative is to remove the call to `update` from `report`, in which case you'll get fast, but constant, answers, changing only after each regular batch update.

`IQagent` uses internally a general purpose set of 251 p-values that includes integer percentile points from 10 to 90, and a logarithmically spaced set of smaller and larger values spanning 10^{-6} to $1 - 10^{-6}$. Other p-values that you request are obtained by interpolation. Of course you cannot get meaningful tail quantiles for small values of p until at least several times $1/p$ data values have been processed. Before that, the program will simply report the smallest value previously seen (or largest value previously seen, for $p \to 1$).

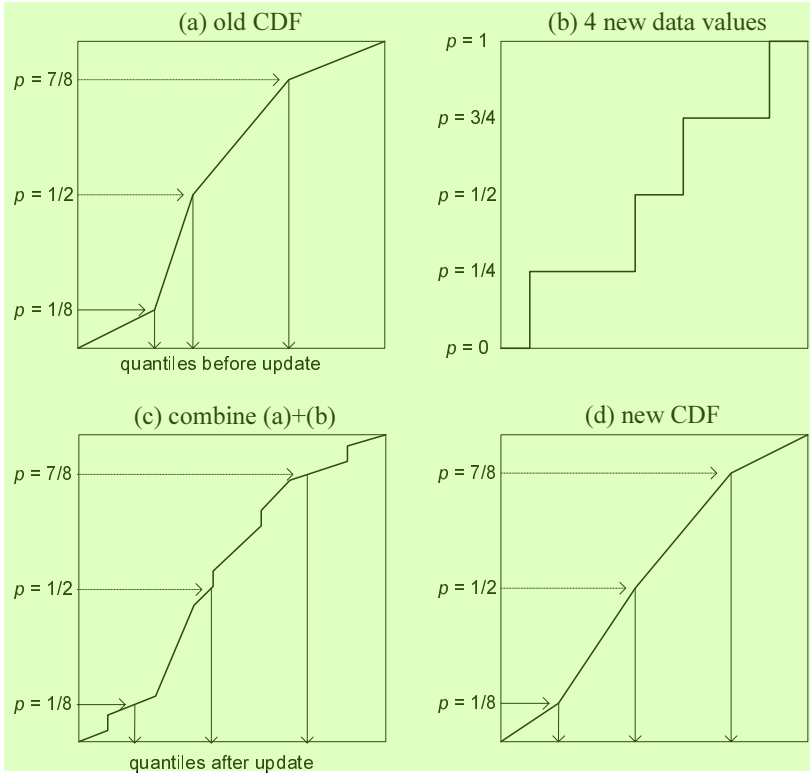

Figure 8.5.1. Algorithm for updating a piecewise linear cumulative distribution function (cdf). (a) The cdf is represented by quantile values at a fixed set of p-values (here, just 3). (b) A batch of new data values (here, just 4) define a stepwise constant cdf. (c) The two cdfs are summed. New data steps are small in proportion to the new batch size versus number of data values previously processed. (d) The new cdf representation is obtained by interpolating the fixed p-values to (c).

iqagent.h

```
struct IQagent {
```
Object for estimating arbitrary quantile values from a continuing stream of data values.
```
    static const Int nbuf = 1000;
```
Batch size. You may $\times 10$ if you expect $> 10^6$ data values.
```
    Int nq, nt, nd;
    VecDoub pval,dbuf,qile;
    Doub q0, qm;

    IQagent() : nq(251), nt(0), nd(0), pval(nq), dbuf(nbuf),
    qile(nq,0.), q0(1.e99), qm(-1.e99) {
```
Constructor. No arguments.
```
        for (Int j=85;j<=165;j++) pval[j] = (j-75.)/100.;
```
Set general purpose array of p-values ranging from 10^{-6} to $1-10^{-6}$. You can change this if you want:
```
        for (Int j=84;j>=0;j--) {
            pval[j] = 0.87191909*pval[j+1];
            pval[250-j] = 1.-pval[j];
        }
    }

    void add(Doub datum) {
```
Assimilate a new value from the stream.
```
        dbuf[nd++] = datum;
        if (datum < q0) {q0 = datum;}
        if (datum > qm) {qm = datum;}
```

```
    if (nd == nbuf) update();                Time for a batch update.
}

void update() {
Batch update, as shown in Figure 8.5.1. This function is called by add or report and
should not be called directly by the user.
    Int jd=0,jq=1,iq;
    Doub target, told=0., tnew=0., qold, qnew;
    VecDoub newqile(nq);                     Will be new quantiles after update.
    sort(dbuf,nd);
    qold = qnew = qile[0] = newqile[0] = q0;     Set lowest and highest to min
    qile[nq-1] = newqile[nq-1] = qm;                 and max values seen so far,
    pval[0] = min(0.5/(nt+nd),0.5*pval[1]);          and set compatible p-values.
    pval[nq-1] = max(1.-0.5/(nt+nd),0.5*(1.+pval[nq-2]));
    for (iq=1;iq<nq-1;iq++) {                Main loop over target p-values for inter-
        target = (nt+nd)*pval[iq];               polation.
        if (tnew < target) for (;;) {
            Here's the guts: We locate a succession of abscissa-ordinate pairs (qnew,tnew)
            that are the discontinuities of value or slope in Figure 8.5.1(c), breaking to
            perform an interpolation as we cross each target.
            if (jq < nq && (jd >= nd || qile[jq] < dbuf[jd])) {
                Found slope discontinuity from old CDF.
                qnew = qile[jq];
                tnew = jd + nt*pval[jq++];
                if (tnew >= target) break;
            } else {                         Found value discontinuity from batch data
                qnew = dbuf[jd];                 CDF.
                tnew = told;
                if (qile[jq]>qile[jq-1]) tnew += nt*(pval[jq]-pval[jq-1])
                    *(qnew-qold)/(qile[jq]-qile[jq-1]);
                jd++;
                if (tnew >= target) break;
                told = tnew++;
                qold = qnew;
                if (tnew >= target) break;
            }
            told = tnew;
            qold = qnew;
        }                                    Break to here and perform the new interpolation.
        if (tnew == told) newqile[iq] = 0.5*(qold+qnew);
        else newqile[iq] = qold + (qnew-qold)*(target-told)/(tnew-told);
        told = tnew;
        qold = qnew;
    }
    qile = newqile;
    nt += nd;
    nd = 0;
}

Doub report(Doub p) {
Return estimated p-quantile for the data seen so far. (E.g., p = 0.5 for median.)
    Doub q;
    if (nd > 0) update();                    You may want to remove this line. See text.
    Int jl=0,jh=nq-1,j;
    while (jh-jl>1) {                        Locate place in table by bisection.
        j = (jh+jl)>>1;
        if (p > pval[j]) jl=j;
        else jh=j;
    }
    j = jl;                                  Interpolate.
    q =  qile[j] + (qile[j+1]-qile[j])*(p-pval[j])/(pval[j+1]-pval[j]);
    return MAX(qile[0],MIN(qile[nq-1],q));
}
};
```

How accurate is the IQ agent algorithm, as compared, say, to storing all N data values in an array A and then reporting the "exact" quantile $A_{\lfloor pN \rfloor}$? There are several sources of error, all of which you can control by modifying parameters in `IQagent`. (We think that the default parameters will work just fine for almost all users.) First, there is interpolation error: The desired cdf is represented by a piecewise linear function between `nq=251` stored values. For typical distributions, this limits the accuracy to three or four significant figures. We find it hard to believe that anyone needs to know a median, e.g., more accurately than this, but if you do, then you can increase the density of p-values in the regions of interest.

Second, there are statistical errors. One way to characterize these is to ask what value j has A_j closest to the quantile reported by IQ agent, and then how small is $|j - pN|$ as a fraction of $[Np(1-p)]^{1/2}$, the accuracy inherent in your finite sample size N. If this fraction is $\lesssim 1$, then the estimate is "good enough," meaning that no method can do substantially better at estimating the population quantiles given your sample.

With the default parameters, and for reasonably behaved distributions, `IQagent` passes this test for $N \lesssim 10^6$. For larger N, the statistical error becomes significant (though still generally smaller than the interpolation error, above). You can, however, decrease it by increasing the batch size, `nbuf`. Larger is always better, if you have the memory and can tolerate the logarithmic increase in the cost per point of the sort.

Although the accuracy of `IQagent` is not guaranteed by a provable bound, the algorithm is fast, robust, and highly recommended. For other approaches to incremental quantile estimation, including some that do give provable bounds (but have other issues), see [3,4] and references cited therein.

8.5.3 Other Uses for Incremental Quantile Estimation

Incremental quantile estimation provides a useful way to histogram data into variable-size bins that each contain the same number of points, without knowing in advance the bin boundaries: First, throw N data values at an `IQagent` object. Next, choose a number of bins m, and define

$$p_i \equiv \frac{i}{m}, \qquad i = 0, \ldots, m \tag{8.5.1}$$

Finally, if q_i is the quantile value at p_i, plot the ith bin from q_i to q_{i+1} with a height

$$h_i = N \frac{p_{i+1} - p_i}{q_{i+1} - q_i}, \qquad i = 0, \ldots, m-1 \tag{8.5.2}$$

A different application concerns the monitoring of quantile values for changes. For example, you might be producing widgets with a parameter T whose tolerance is $T \pm \delta T$, and you want an early warning if the observed values of T at the 5th and 95th percentiles start to drift.

The `IQagent` object is easily modified for such applications. Simply change the line `nt += nd` to `nt = my_constant`, where `my_constant` is the number of past widgets that you want to average over. (More precisely, the number corresponding to one e-fold of weight decrease in an exponentially decreasing average over all past production.) Now, the stored cdf combines with a new batch of data with a constant, not an increasing, weight, and you can look for changes over time in any desired quantiles.

8.5.4 In-Place Selection

In-place, nondestructive, selection is conceptually simple, but it requires a lot of bookkeeping, and it is correspondingly slow. 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

$$M \sim 2^{\sqrt{\log_2 N}} \tag{8.5.3}$$

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.

Further discussion, and code, is in a Webnote [5].

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), pp. 126ff.[1]

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

Chambers, J.M., James, D.A., Lambert, D., and Vander Wiel, S. 2006, "Monitoring Networked Applications with Incremental Quantiles," *Statistical Science*, vol. 21.[2]

Tieney, L. 1983, "A Space-efficient Recursive Procedure for Estimating a Quantile of an Unknown Distribution," *SIAM Journal on Scientific and Statistical Computing*, vol. 4, pp. 706-711.[3]

Liechty, J.C., Lin, D.K.J, and McDermott, J.P. 2003, "Single-Pass Low-Storage Arbitrary Quantile Estimation for Massive Datasets," *Statistics and Computing*, vol. 13, pp. 91–100.[4]

Numerical Recipes Software 2007, "Code Listing for Selip," *Numerical Recipes Webnote No. 11*, at `http://www.nr.com/webnotes?11` [5]

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 $0, \ldots, N-1$. You are given pairwise information about whether the 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 $F(j) = j$.[1] 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; and (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=0...m-1`, are the numbers of two elements that (we are thus told) are related.

eclass.h

```
void eclass(VecInt_O &nf, VecInt_I &lista, VecInt_I &listb)
```

Given `m` equivalences between pairs of `n` individual elements in the form of the input arrays `lista[0..m-1]` and `listb[0..m-1]`, this routine returns in `nf[0..n-1]` the number of the equivalence class of each of the `n` elements, integers between `0` and `n-1` (not all such integers used).

```
{
    Int l,k,j,n=nf.size(),m=lista.size();
    for (k=0;k<n;k++) nf[k]=k;                  Initialize each element its own class.
    for (l=0;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=0;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 boolean function `equiv(j,k)` that returns a value `true` if elements `j` and `k` are related, or `false` 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:

eclass.h

```
void eclazz(VecInt_O &nf, Bool equiv(const Int, const Int))
```

Given a user-supplied boolean function `equiv` that tells whether a pair of elements, each in the range `0...n-1`, are related, return in `nf[0..n-1]` equivalence class numbers for each element.

```
{
    Int kk,jj,n=nf.size();
    nf[0]=0;
    for (jj=1;jj<n;jj++) {                      Loop over first element of all pairs.
```

```
        nf[jj]=jj;
        for (kk=0;kk<jj;kk++) {              Loop over second element of all pairs.
            nf[kk]=nf[nf[kk]];               Sweep it up this much.
            if (equiv(jj+1,kk+1)) nf[nf[nf[kk]]]=jj;
            Good exercise for the reader to figure out why this much ancestry is necessary!
        }
    }
    for (jj=0;jj<n;jj++) nf[jj]=nf[nf[jj]];        Only this much sweeping is needed
}                                                      finally.
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1997, *Fundamental Algorithms*, 3rd ed., vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.3.3.[1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapter 30.