UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 1999

Sorting*

1 Sorting: Basic concepts

At least at one time, most CPU time and I/O bandwidth was spent sorting (these days, I suspect more may be spent rendering `.gif` files). As a result, sorting has been the subject of extensive study and writing. We will hardly scratch the surface here.

The purpose of any sort is to permute some set of items that we'll call *records* so that they are sorted according to some ordering relation. In general, the ordering relation looks at only part of each record, the *key*. The records may be sorted according to more than one key, in which case we refer to the *primary key* and to *secondary keys*. This distinction is actually realized in the ordering function: record A comes before B iff either A's primary key comes before B's, or their primary keys are equal and A's secondary key comes before B's. One can extend this definition in an obvious way to hierarchy of multiple keys. For the purposes of these Notes, I'll usually assume that records are of some type `Record` and that there is an ordering relation on the records we are sorting. I'll write `before(`A, B`)` to mean that the key of A comes before that of B in whatever order we are using.

Although conceptually we move around the records we are sorting so as to put them in order, in fact these records may be rather large. Therefore, it is often preferable to keep around pointers to the records and exchange those instead. If necessary, the real data can be physically re-arranged as a last step. In Java, this is very easy of course, since "large" data items are always referred to by pointers.

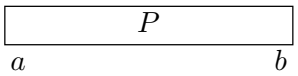
Stability. A sort is called *stable* if it preserves the order of records that have equal keys. Any sort can be made stable by (in effect) adding the original record position as a final secondary key, so that the list of keys (Bob, Mary, Bob, Robert) becomes something like (Bob.1, Mary.2, Bob.3, Robert.4).

Inversions. For some analyses, we need to have an idea of *how out-of-order* a given sequence of keys is. One useful measure is the number of *inversions* in the sequence – in a sequence of keys $k_0, \ldots, k_{N-1}$, this is the number of pairs of integers, (i, j), such that $i < j$ and $k_i > k_j$. When the keys are already in order, the number of inversions is 0, and when they are in reverse order, so that *every* pair of keys is in the wrong order, the number of inversions is $N(N-1)/2$, which is the number of pairs of keys. When all keys are originally within some distance D of their correct positions in the sorted permutation, we can establish a pessimistic upper bound of DN inversions in the original permutation.

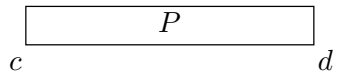
Internal vs. external sorting. A sort that is carried out entirely in primary memory is known as an *internal* sort. Those that involve auxiliary disks (or, in the old days especially, tapes) to hold intermediate results are called *external* sorts. The sources of input and output are irrelevant to this classification (one can have internal sorts on data that comes from an external file; it's just the intermediate files that matter).

2 A Little Notation

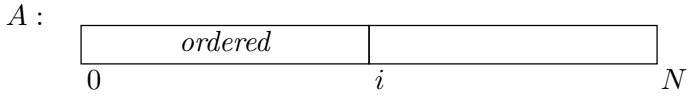
Many of the algorithms in these notes deal with (or can be thought of as dealing with) arrays. In describing or commenting them, we sometimes need to make assertions about the contents of these arrays. For this purpose, I am going to use a notation used by David Gries to make descriptive comments about my arrays. The notation



denotes a section of an array whose elements are indexed from a to b and that satisfies property P. It also asserts that $a \leq b + 1$; if $a > b$, then the segment is empty. I can also write

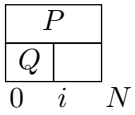


to describe an array segment in which items $c + 1$ to $d - 1$ satisfy P, and that $c < d$. By putting these segments together, I can describe an entire array. For example,



is true if the array A has N elements, elements 0 through $i - 1$ are ordered, and $0 \leq i \leq N$. A notation such as

denotes a 1-element array segment whose index is j and whose (single) value satisfies P. Finally, I'll occasionally need to have simultaneous conditions on nested pieces of an array. For example,



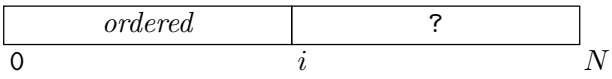
refers to an array segment in which items 0 to $N-1$ satisfy P, items 0 to $i-1$ satisfy Q, $0 \le N$, and $0 \le i \le N$.

3 Insertion sorting

One very simple sort – and quite adequate for small applications, really – is the *straight insertion sort.* The name comes from the fact that at each stage, we insert an as-yet-unprocessed record into a (sorted) list of the records processed so far, as illustrated in Figure 1. The algorithm is as follows (when the sequence of data is stored as an array).

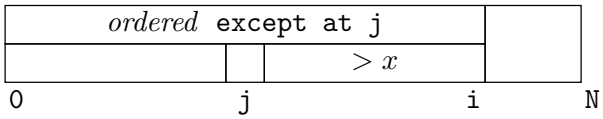
```
/** Permute the elements of A to be in ascending order. */
static void insertionSort(Record[] A) {
   int N = A.length;
   for (int i = 1; i < N; i += 1) {

      /*    A:                                                    */
```



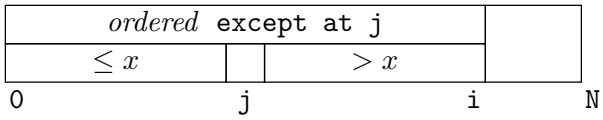

```
      Record x = A[i];
      int j;
      for (j = i; j > 0 && before(x, A[j-1]); j -= 1) {

         /*   A:                                                  */
```




```
         A[j] = A[j-1];
      }


      /*   A:                                                     */
```




```
      A[j] = x;
   }
}
```

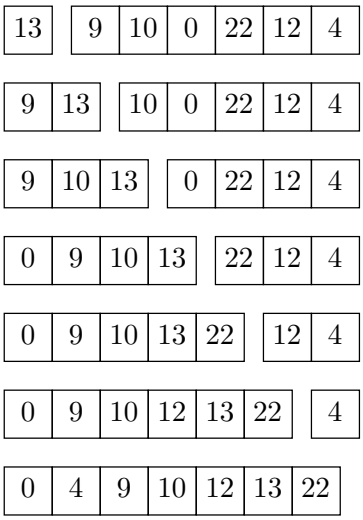


Figure 1: Example of insertion sort, showing the array before each call of `insertElement`. The gap at each point separates the portion of the array known to be sorted from the unprocessed portion.

A common way to measure the time required to do a sort is to count the comparisons of keys (here, the calls to `before`) The total (worst-case) time required by `insertionSort` is $\sum_{0<i<N} C_{IL}(i)$, where $C_{IL}(m)$ is the cost of the inner (`j`) loop when `i`$= m$, and N is the size of `A`. Examination of the inner loop shows that the number of comparisons required is equal to the number of records numbered 0 to `i-1` whose keys larger than that of `x`, plus one if there is at least one smaller key. Since `A[0..i-1]` is sorted, it contains no inversions, and therefore, the number of elements after `X` in the sorted part of `A` happens to be equal to the number of inversions in the sequence `A[0],..., A[i]` (since `X` is `A[i]`). When `X` is inserted correctly, there will be no inversions in the resulting sequence. It is fairly easy to work out from that point that the running time of `insertionSort`, measured in key comparisons, is bounded by $I + N - 1$, where I is the total number of inversions in the original argument to `insertionSort`. Thus, the more sorted an array is to begin with, the faster `insertionSort` runs.

4 Shell's sort

The problem with insertion sort can be seen by examining the worst case – where the array is initially in reverse order. The keys are a great distance from their final resting places, and must be moved one slot at a time until they get there. If keys could be moved great distances in little time, it might speed things up a bit. This is the idea behind Shell's sort[1]. We choose a diminishing sequence of strides, $s_0 > s_1 > \ldots > s_{m-1}$, typically choosing $s_{m-1} = 1$. Then,

[1]Also known as "shellsort." Knuth's reference: Donald L. Shell, in the *Communications of the ACM* **2** (July, 1959), pp. 30–32.

for each j, we divide the N records into the s_j interleaved sequences

$$R_0, R_{s_j}, R_{2s_j}, \ldots,$$
$$R_1, R_{s_j+1}, R_{2s_j+1}, \ldots$$
$$\ldots$$
$$R_{s_j-1}, R_{2s_j-1}, \ldots$$

and sort each of these using insertion sort. Figure 2 illustrates the process with a vector in reverse order (requiring a total of 49 comparisons as compared with 120 comparisons for straight insertion sort).

A good sequence of s_j turns out to be $s_j = \lfloor 2^{m-j} - 1 \rfloor$, where $m = \lfloor \lg N \rfloor$. With this sequence, it can be shown that the number of comparisons required is $O(N^{1.5})$, which is considerably better than $O(N^2)$. Intuitively, the advantages of such a sequence – in which the successive s_j are relatively prime – is that on each pass, each position of the vector participates in a sort with a new set of other positions. The sorts get "jumbled" and get more of a chance to improve the number of inversions for later passes.

5 Distribution counting

When the range of keys is restricted, there are a number of optimizations possible. In Column #1 of his book *Programming Pearls*[2], Jon Bentley gives a simple algorithm for the problem of sorting N distinct keys, all of which are in a range of integers so limited that the programmer can build a vector of bits indexed by those integers. In the following program, I use a Java `BitSet`, which is abstractly a set of non-negative integers (implemented as a packed array of 1-bit quantities):

```
/** Permute the elements of KEYS, which must be distinct,
 *  into ascending order. */
static void distributionSort1(int[] keys) {
   int N = keys.length;
   int L = min(keys), U = max(keys);

   java.util.BitSet b = new java.util.BitSet();
   for (int i = 0; i < N; i += 1)
      b.set(keys[i] - L);
   for (int i = L, k = 0; i <= U; i += 1)
      if (b.get(i-L)) {
         keys[k] = i; k += 1;
      }
}
```

[2] Addison-Wesley, 1986. By the way, that column makes very nice "consciousness-raising" column on the subject of appropriately-engineered solutions. I highly recommend both this book and his *More Programming Pearls,* Addison-Wesley, 1988.

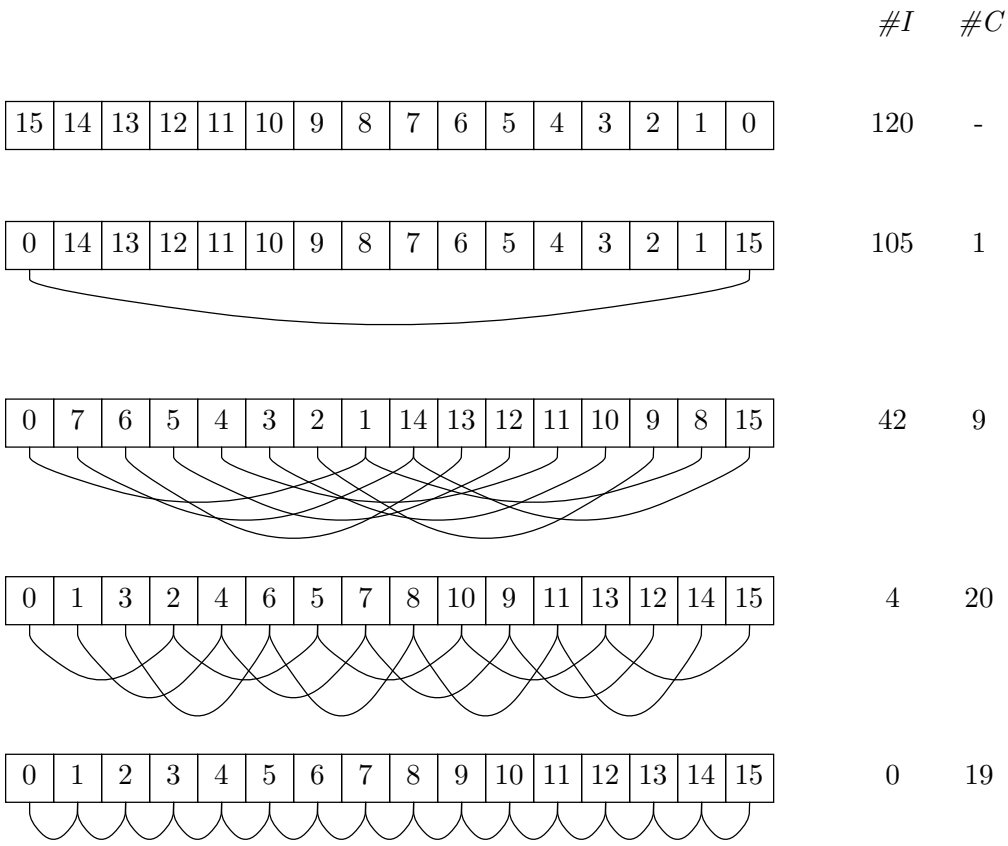


Figure 2: An illustration of Shell's sort, starting with a vector in reverse order. The increments are 15, 7, 3, and 1. The column marked $\#I$ gives the number of inversions remaining in the array, and the column marked $\#C$ gives the number of key comparisons required to obtain each line from its predecessor. The arcs underneath the arrays indicate which subsequences of elements are processed at each stage.

Here, functions `min` and `max` return the minimum and maximum values in an array. Their values are arbitrary if the arrays are empty.

Let's consider a more general technique we can apply even when there are multiple records with the same key. Assume that the keys of the records to be sorted are in some reasonably small range of integers. Then the program shown in Figure 4 sorts N records stably, moving them from an input array (`A`) to a different output array (`B`). It computes the correct final position in `B` for each record in `A`. To do so, it uses the fact that the position of any record in `B` is supposed to be the number the records that either have smaller keys than it has, or that have the same key, but appear before it in `A`. Figure 3 contains an example of the program in operation.

6 Selection sort

In insertion sort, we determine an item's final position piecemeal. Another way to proceed is to place each record in its final position in one move by selecting the smallest (or largest) key at each step. The simplest implementation of this idea is *straight selection sorting,* as follows.

```
static void selectionSort(Record[] A)
{
   int N = A.length;
   for (int i = 0; i < N-1; i += 1) {

      /*  A:                                    */
                ordered | ≥ items 0..i-1
                0       i              N
      int m, j;
      for (j = i+1, m = i; j < N; j += 1)
         if (before(A[j], A[m]) m = j;
      /*  Now A[m] is the smallest element in A[i..N-1] */
      swap(A, i, m);
   }
}
```

Here, `swap(A,i,m)` is assumed to swap elements `i` and `m` of `A`. This sort is not stable; the swapping of records prevents stability. On the other hand, the program can be modified to produce its output in a separate output array, and then it is relatively easy to maintain stability [how?].

It should be clear that the algorithm above is insensitive to the data. Unlike insertion sort, it *always* takes the same number of key comparisons – $N(N-1)/2$. Thus, in this naive form, although it is very simple, it suffers in comparison to insertion sort (at least on a sequential machine).

On the other hand, we have seen another kind of selection sort before – heapsort is a form of selection sort that (in effect) keeps around information about the results of comparisons from each previous pass, thus speeding up the minimum selection considerably.

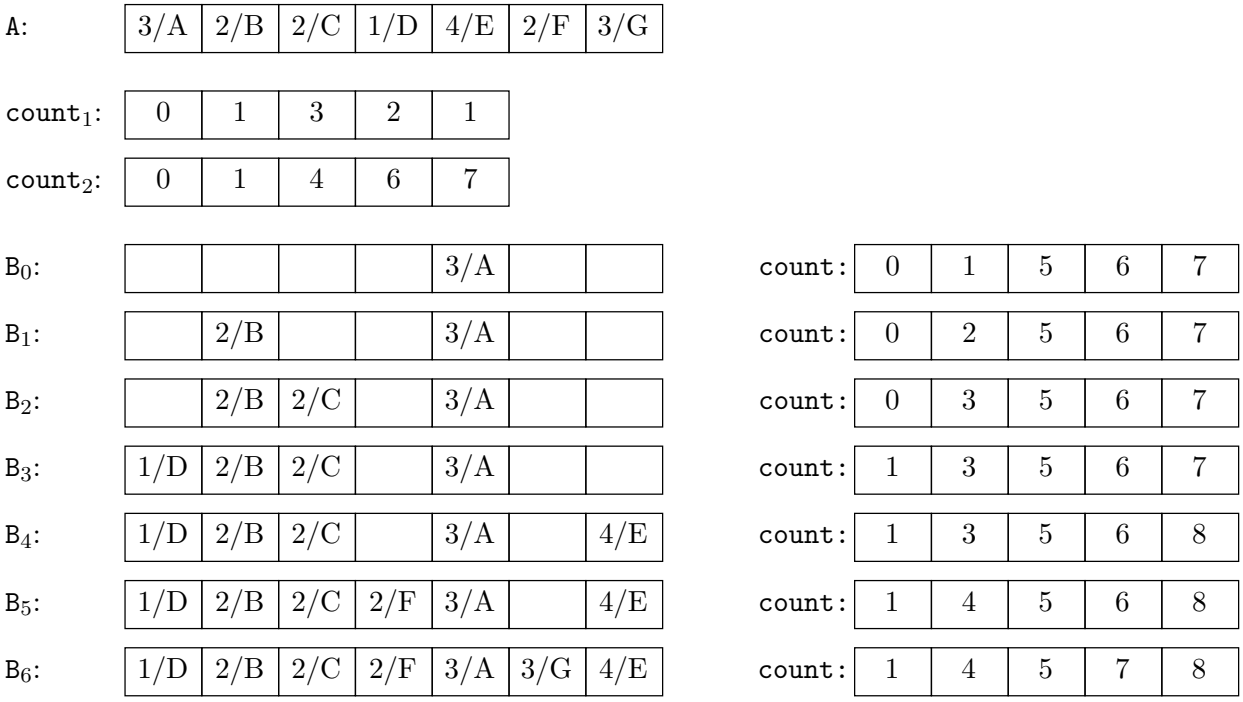


Figure 3: Illustration of distribution sorting. The values to be sorted are shown in the array marked `A`. The keys are the numbers to the left of the slashes. The data are sorted into the array `B`, shown at various points in the algorithm. The labels at the left refer to points in the program in Figure 4. Each point B_k indicates the situation at the end of the last loop where $i = k$. The role of array `count` changes. First (at count_1) `count[k-1]` contains the number of instances of key `(k-1)-1`. Next (at count_2), it contains the number of instances of keys less than `k-1`. In the B_i lines, `count[`$k-1$`]` indicates the position (in `B`) at which to put the next instance of key k. (It's `k-1` in these places, rather than `k`, because 1 is the smallest key.

```
/** Assuming that A and B are not the same array and are of
 *  the same size, sort the elements of A stably into B.
 */
void distributionSort2(Record[] A, Record[] B)
{
   int N = A.length;

   int L = min(A), U = max(A);

   /* count[i-L] will contain the number of items <i */
   // NOTE: count[U-L+1] is not terribly useful, but is
   // included to avoid having to test for for i == U in
   // the first i loop below.
   int[] count = new int[U-L+2];

   // Clear count: Not really needed in Java, but a good habit
   // to get into for other languages (e.g., C, C++).
   for (int j = L; j <= U+1; j += 1)
      count[j-L] = 0;

   for (int i = 0; i < N; i += 1)
      count[key(A[i]) - L + 1] += 1;
   /* Now count[i-L] == # of records whose key is equal to i-1 */
   /* See Figure 3, point count1. */

   for (int j = L+1; j <= U; j += 1)
       count[j-L] += count[j-L-1];
   /* Now count[k] == # of records whose key is less than k,
    * for all k, L <= k <= U.
    * See Figure 3, point count2. */

   for (i = 0; i < N; i += 1) {
      /* Now count[k-L] == # of records whose key is less than k,
       * or whose key is k and have already been moved to B. */
      B[count[key(A[i])-L]] = A[i];
      count[key(A[i])-L] += 1;
      /* See Figure 3, points B0–B6. */
   }
}
```

Figure 4: Distribution Sorting. This program assumes that `key(R)` is an integer.

7 Exchange sorting: Quicksort

One of the most popular methods for internal sorting was developed by C. A. R. Hoare[3]. Evidently much taken with the technique, he named it "quicksort." The name is actually quite appropriate. The basic algorithm is as follows.

```
static final int K = ...;

void quickSort(Record A[])
{
    quickSort(A,0,A.length-1);
    insertionSort(A);
}

/* Permute A[L..U] so that all records are $<$ K away from their */
/* correct positions in sorted order.  Assumes K > 0. */
void quickSort(Record[] A, int L, int U)
{
   if (U-L+1 > K) {
      Choose Record T = A[p], where p ∈ L..U;
 P:   Set i and permute A[L..U] to establish the partitioning
          condition:

                                                              ;
          ┌──────────────────┬─┬──────────────────┐
          │ key ≤ key(T)     │T│ key ≥ key(T)     │
          └──────────────────┴─┴──────────────────┘
          0                  i                   N
      quickSort(A, L, i-1); quickSort(A, i+1, U);
   }
}
```

Here, `K` is a constant value that can be adjusted to tune the speed of the sort. Once the approximate sort gets all records within a distance `K-1` of their final locations, the final insertion sort proceeds in $O(KN)$ time. If `T` can be chosen so that its key is near the median key for the records in `A`, then we can compute roughly that the time in key comparisons required for performing `quicksort` on N records is approximated by $C(N)$, defined as follows.

$$\begin{aligned} C(K) &= 0 \\ C(N) &= N - 1 + 2C(\lfloor N/2 \rfloor) \end{aligned}$$

This assumes that we can partition an N-element array in $N-1$ comparisons, which we'll see to be possible. We can get a sense for the solution by considering the case $N = 2^m K$:

$$\begin{aligned} C(N) &= 2^m K + 2C(2^{m-1}K) \\ &= 2^m K - 1 + 2^m K - 2 + 4C(2^{m-2}K) \end{aligned}$$

[3]Knuth's reference: *Computing Journal* **5** (1962), pp. 10–15.

$$\begin{aligned} &= \underbrace{2^m K + \ldots + 2^m K}_{m} - 1 - 2 - 4 - \ldots - 2^{m-1} + C(K) \\ &= m2^m K - 2^m + 1 \\ &\in \Theta(m2^m K) = \Theta(N \lg N) \end{aligned}$$

(since $\lg(2^m K) = m \lg K$).

Unfortunately, in the worst case – where the partition `T` has the largest or smallest key, quicksort is essentially a straight selection sort, with running time $\Theta(N^2)$. Thus, we must be careful in the choice of the partitioning element. One technique is to choose a random record's key for `T`. This is certainly likely to avoid the bad cases. A common choice for `T` is the *median* of `A[L]`, `A[(L+U)/2]`, and `A[U]`, which is also unlikely to fail.

Partitioning. This leaves the small loose end of how to partition the array at each stage (step P in the program above). There are many ways to do this. Here is one due to Nico Lomuto – not the fastest, but simple.

```
P:
swap(A, L, p);
i = L;
for (int j = L+1; j < U; j += 1) {

   /*  A[L..U]:                                    */
                  | T | <T | ≥T |  |
                    L     i     j  U
   if (before(A[j],T)) {
      i += 1;
      swap(A, j, i);
   }
}

/*  A[L..U]:                          */
                | T |  <T  |  ≥T  |
                  L       i      U
swap(A, L, i);
```

Some authors go to the trouble of developing non-recursive versions of quicksort, evidently under the impression that they are thereby vastly improving its performance. This view of the cost of recursion is widely held, so I suppose I can't be surprised. However, a quick test using a C version indicated about a 3% improvement using his iterative version. This is hardly worth obscuring one's code to obtain.

8 Merge sorting

Quicksort was a kind of divide-and-conquer algorithm[4] that we might call "try to divide-and-conquer," since it is not guaranteed to succeed in dividing the data evenly. An older technique, known as merge sorting, is a form of divide-and-conquer that does guarantee that the data are divided evenly.

At a high level, it goes as follows.

```
/** Sort items A[L..U]. */
static void mergeSort(Record[] A, int L, int U)
{
    if (L >= U)
        return;
    mergeSort(A, L, (L+U)/2);
    mergeSort(A, (L+U)/2+1, U);
    merge(A, L, (L+U)/2, A, (L+U)/2+1, U, A, L);
}
```

The `merge` program has the following specification

```
/** Assuming V0[L0..U0] and V1[L1..U1] are each sorted in */
/*  ascending order by keys, set V2[L2..U2] to the sorted contents */
/*  of V0[L0..U0], V1[L1..U1].  (U2 = L2+U0+U1-L0-L1+1). */
void merge(Record[] V0, int L0, int U0, Record[] V1, int L1, int U1,
           Record[] V2, int L2)
```

Since `V0` and `V1` are in ascending order already, it is easy to do this in $\Theta(N)$ time, where $N = U2 - L2 + 1$, the combined size of the two arrays. It progresses through the arrays from left to right. That makes it well-suited for computers with small memories and lots to sort. The arrays can be on secondary storage devices that are restricted to *sequential access* – i.e., requiring that one read or write the arrays in increasing (or decreasing) order of index[5].

The real work is done by the merging process, of course. The pattern of these merges is rather interesting. For simplicity, consider the case where N is a power of two. If you trace the execution of `mergeSort`, you'll see the following pattern of calls on `merge`.

[4] The term *divide-and-conquer* is used to describe algorithms that divide a problem into some number of smaller problems, and then combine the answers to those into a single result.

[5] A familiar movie cliché of decades past was spinning tape units to indicate that some piece of machinery was a computer (also operators flipping console switches – something one almost *never* really did during normal operation). When those images came from footage of real computers, the computer was most likely sorting.

Call #	V0	V1
0.	A[0]	A[1]
1.	A[2]	A[3]
2.	A[0..1]	A[2..3]
3.	A[4]	A[5]
4.	A[6]	A[7]
5.	A[4..5]	A[6..7]
6.	A[0..3]	A[4..7]
7.	A[8]	A[9]
	etc.	

We can exploit this pattern to good advantage when trying to do merge sorting on linked lists of elements, where the process of dividing the list in half is not as easy as it is for arrays. Assume that records are linked together into `List`s. The program below shows how to perform a merge sort on these lists; Figure 5 illustrates the process. The program maintains a "binomial comb" of sorted sublists, `comb[0 .. M-1]`, such that the list in `comb[i]` is either null or has length 2^i.

```
/** Permute the Records in List A to be sorted by key,
 *  returning the resulting list.  The list A may be
 *  destructively modified. */
static List mergeSort(List A)
{
   int M = a number such that 2^(M-1) ≥ length of A;
   List comb = new Record[M];

   for (int i = 0; i < M; i += 1)
       comb[i] = null;

   for (List L = A, next; L != null; L = next) {
        next = L.tail;
        L.tail = null;
        addToComb(comb, L);
        L = next;
    }

    List r;  r = null;
    for (int i = 0; i < M; i += 1)
       r = mergeLists(r, comb[i]);

    return r;
}
```

At each point, the comb contains sorted lists that are to be merged. We first build up the

comb one new item at a time, and then take a final pass through it, merging all its lists. To add one element to the comb, we have

```
/** Assuming that each C[i] is a sorted list whose length is either 0
 *  or 2^i elements, adds the singleton list P to the items in
 *  C so as to maintain this same condition. */
static void addToComb(List C[], List p)
{
   int i;
   for (i = 0; C[i] != null; i += 1) {
      p = mergeLists(p, C[i]);
      C[i] = null;
   }
   C[i] = p;
}
```

I leave to you the `mergeLists` procedure, which takes two sorted lists and interleaves their elements into a single sorted list.

8.1 Complexity

The optimistic time estimate for quicksort applies in the worst case to merge sorting, because merge sorts really do divide the data in half with each step (and merging of two lists or arrays takes linear time). Thus, merge sorting is a $\Theta(N \lg N)$ algorithm, with N the number of records. Unlike quicksort or insertion sort, merge sorting as I have described it is generally insensitive to the ordering of the data. This changes somewhat when we consider external sorting, but $O(N \lg N)$ comparisons remains an upper bound.

9 Speed of comparison-based sorting

I've presented a number of algorithms and have claimed that the best of them require $\Theta(N \lg N)$ comparisons in the worst case. There are several obvious questions to ask about this bound. First, how do "comparisons" translate into "instructions"? Second, can we do better than $N \lg N$?

The point of the first question is that I have been a bit dishonest to suggest that a comparison is a constant-time operation. For example, when comparing strings, the size of the strings matters in the time required for comparison in the worst case. Of course, on the average, one expects not to have to look too far into a string to determine a difference. Still, this means that to correctly translate comparisons into instructions, we should throw in another factor of the length of the key. Suppose that the N records in our set all have distinct keys. This means that the keys themselves have to be $\Omega(\lg N)$ long. Assuming keys are no longer than necessary, and assuming that comparison time goes up proportionally to the size of a key (in the worst case), this means that sorting *really* takes $\Theta(N(\lg N)^2)$ time (assuming that the time required to move one of these records is at worst proportional to the size of the key).

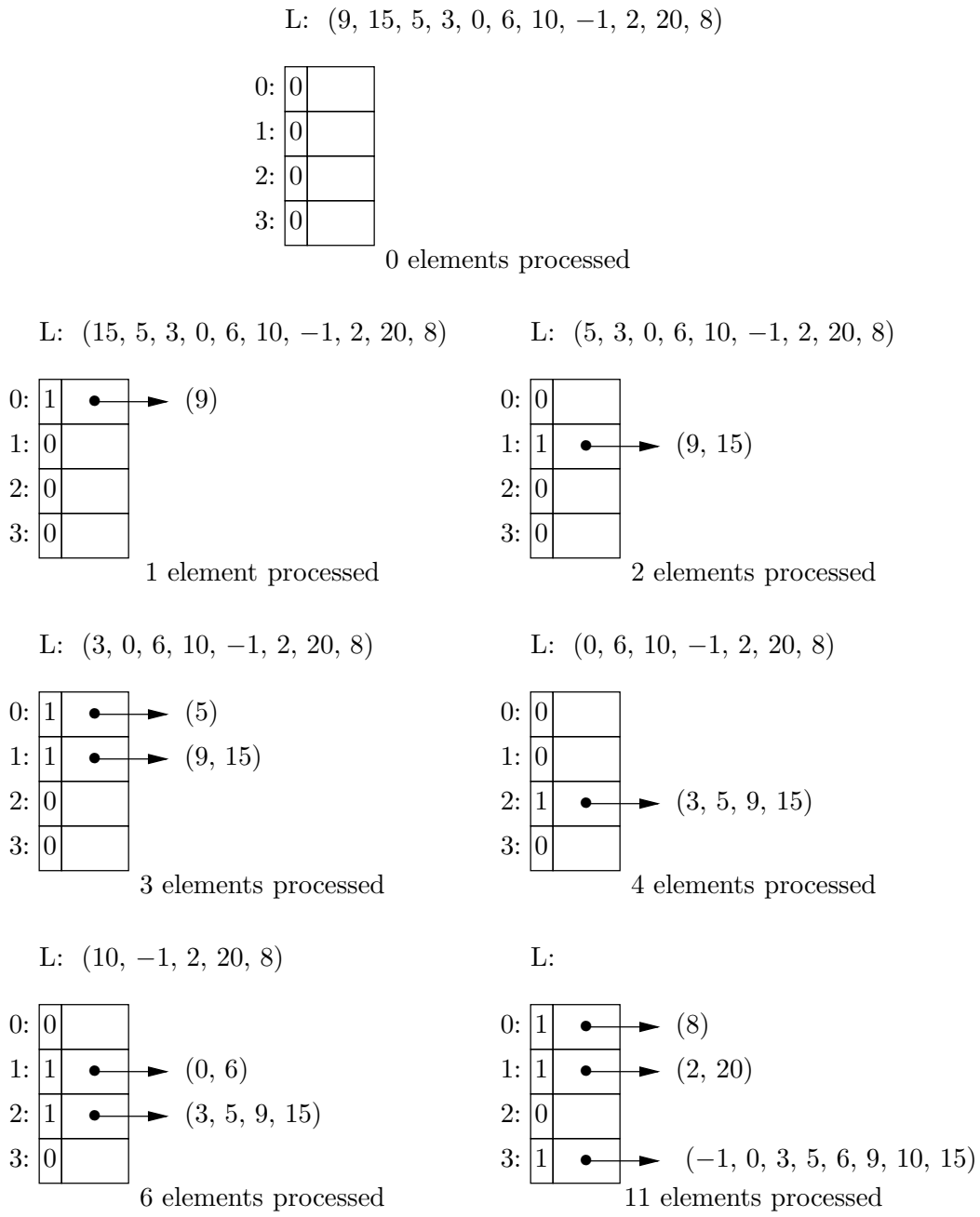


Figure 5: Merge sorting of lists, showing the state of the "comb" after various numbers of items from the list `L` have been processed. The final step is to merge the lists remaining in the comb after all 11 elements from the original list have been added to it. The 0s and 1s in the small boxes are decorations to illustrate the pattern of merges that occurs. Each empty box has a 0 and each non-empty box has a 1. If you read the contents of the four boxes as a single binary number, units bit on top, it equals the number of elements processed.

As to the question about whether it is possible to do better than $\Theta(N \lg N)$, the answer is that if the only information we can obtain about keys is how they compare to each other, then we cannot do better than $\Theta(N \lg N)$. That is, $\Theta(N \lg N)$ comparisons is a lower bound on the worst case of all possible sorting algorithms that use comparisons.

The proof of this assertion is instructive. A sorting program can be thought of as first performing a sequence of comparisons, and then deciding how to permute its inputs, based *only* on the information garnered by the comparisons. The two operations actually get mixed, of course, but we can ignore that fact here. In order for the program to "know" enough to permute two different inputs differently, these inputs must cause different sequences of comparison results. Thus, we can represent this idealized sorting process as a tree in which the leaf nodes are permutations and the internal nodes are comparisons, with each left child containing the comparisons and permutations that are performed when the comparison turns out true and the right child containing those that are performed when the comparison turns out false. Figure 6 illustrates this for the case $N = 3$. The height of this tree corresponds to the number of comparisons performed. Since the number of possible permutations (and thus leaves) is $N!$, and the minimal height of a binary tree with M leaves is $\lceil \lg M \rceil$, the minimal height of the comparison tree for N records is roughly $\lg(N!)$. Now

$$\begin{aligned} \lg N! &= \lg N + \lg(N-1) + \ldots + 1 \\ &\leq \lg N + \lg N + \ldots + \lg N \\ &= N \lg N \\ &\in O(N \lg N) \end{aligned}$$

and also (taking N to be even)

$$\begin{aligned} \lg N! &\geq \lg N + \lg(N-1) + \ldots + \lg(N/2) \\ &\geq (N/2+1)\lg(N/2) \\ &\in \Omega(N \lg N) \end{aligned}$$

so that

$$\lg N! \in \Theta(N \lg N).$$

Thus *any* sorting algorithm that uses only (true/false) key comparisons to get information about the order of its input's keys requires $\Theta(N \lg N)$ comparisons in the worst case to sort N keys.

10 Radix sorting

Suppose that we are *not* restricted to simply comparing keys. Can we improve on our $O(N \lg N)$ bounds? Interestingly enough, we can, sort of. This is possible by means of the sorting analogue of the search trie that we've seen previously: *radix sort*.

Most keys are actually sequences of fixed-size pieces (characters or bytes, in particular) with a lexicographic ordering relation – that is, the key $k_0 k_1 \cdots k_{n-1}$ is less than $k'_0 k'_1 \cdots k'_{n-1}$ if $k_0 < k'_0$ or $k_0 = k'_0$ and $k_1 \cdots k_{n-1}$ is less than $k'_1 \cdots k'_{n-1}$ (we can always treat the keys

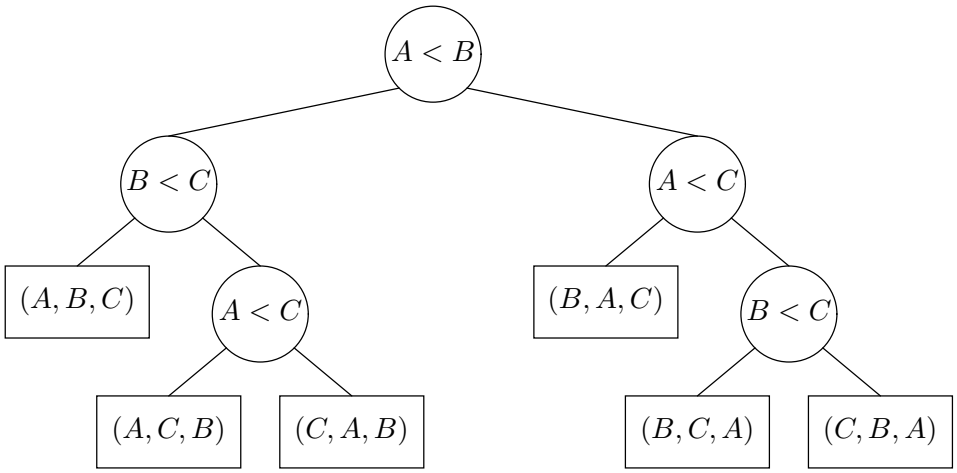


Figure 6: A comparison tree for $N = 3$. The three values being sorted are A, B, and C. Each internal node indicates a test. The left children indicate what happens when the test is successful (true), and the right children indicate what happens if it is unsuccessful. The leaf nodes (rectangular) indicate the ordering of the three values that is uniquely determined by the comparison results that lead down to them. We assume here that A, B, and C are distinct. This tree is optimal, demonstrating that three comparisons are needed in the worst case to sort three items.

as having equal length by choosing a suitable padding character for the shorter string). Just as in a search trie we used successive characters in a set of keys to distribute the strings amongst subtrees, we can use successive characters of keys to sort them. There are basically two varieties of algorithm – from least significant to most significant digit (LSD-first) and from most significant to least significant digit (MSD-first). I use "digit" here as a generic term encompassing not only decimal digits, but also alphabetic characters, or whatever is appropriate to the data one is sorting.

Let's start with the LSD-first algorithm. The idea here is to first use the least significant character to order all records, then the second-least significant, and so forth. At each stage, we perform a *stable* sort, so that if the k most significant characters of two records are identical, they will remain sorted by the remaining, least significant, characters. Because characters have a limited range of values, it is easy to sort them in linear time (using, for example, `distributionSort2`, or, if the records are kept in a linked list, by keeping an array of list headers, one for each possible character value). Figure 7 illustrates the process.

LSD-first radix sort is precisely the algorithm used by card-sorters. These machines had a series of bins and could be programmed (using plugboards) to drop cards from a feeder into bins depending on what was punched in a particular column. By repeating the process for each column, one ended up with a sorted deck of cards.

Each distribution of a record to a bin takes (about) constant time (assuming we use pointers to avoid moving large amounts of data around). Thus, the total time is proportional to the total amount of key data – which is the total number of bytes in all keys. In other words, radix sorting is $O(B)$ where B is the total number of bytes of key data. If keys are K bytes long, then $B = NK$, where N is the number of records. Since merge sorting, heap

Initial: set, cat, cad, con, bat, can, be, let, bet

			bet
			let
			bat
		can	cat
be	cad	con	set
‘␣’	‘d’	‘n’	‘t’

After first pass: be, cad, con, can, set, cat, bat, let, bet

bat	bet	
cat	let	
can	set	
cad	be	con
‘a’	‘e’	‘o’

After second pass: cad, can, cat, bat, be, set, let, bet, con

	con		
bet	cat		
be	can		
bat	cad	let	set
‘b’	‘c’	‘l’	‘s’

After final pass: bat, be, bet, cad, can, cat, con, let, set

Figure 7: An example of a LSD-first radix sort. Each pass sorts by one character, starting with the last. Sorting consists of distributing the records to bins indexed by characters, and then concatenating the bins' contents together. Only non-empty bins are shown.

sorting, etc., require $O(N \lg N)$ comparisons, each requiring in the worst case K time, we get a total time of $O(NK \lg N) = O(B \lg N)$ time for these sorts. Even if we assume constant comparison time, if keys are no longer than they have to be (in order to provide N different keys we must have $K \geq \log_C N$, where C is the number of possible characters), then radix sorting is also $O(N \lg N)$.

Thus, relaxing the constraint on what we can do to keys yields a fast sorting procedure, at least in principle. As usual, the Devil is in the details. If the keys are considerably longer than $\log_C N$, as they very often are, the passes made on the last characters will typically be largely wasted. One possible improvement, which Knuth credits to M. D. Maclaren, is to use LSD-first radix sort on the first two characters, and then finish with an insertion sort (on the theory that things will almost be in order after the radix sort). We must fudge the definition of "character" for this purpose, allowing characters to grow slightly with N. For example, when $N = 100000$, Maclaren's optimal procedure is to sort on the first and second 10-bit segments of the key (on an 8-bit machine, this is the first 2.25 characters). Of course, this technique can, in principle, make no guarantees of $O(B)$ performance.

We may turn instead to an MSD-first radix sort. The idea here is simple enough. We sort the input by the first (most-significant) character into C (or fewer) subsequences, one for each starting character (that is, the first character of all the keys in any given subsequence is the same). Next, we sort each of the subsequences that has more than one key individually by its second character, yielding another group of subsequences in which all keys in any given subsequence agree in their first two characters. This process continues until all subsequences are of length 1. At each stage, we order the subsequences, so that one subsequence precedes another if all its strings precede all those in the other. When we are done, we simply write out all the subsequences in the proper order.

The tricky part is keeping track of all the subsequences so that they can be output in the proper order at the end and so that we can quickly find the next subsequence of length greater than one. Here is a sketch of one technique for sorting an array; it is illustrated in Figure 8.

```
static final int ALPHA = size of alphabet of digits;

/** Sort A[L..U] stably, ignoring the first k characters in each key.  */
static void MSDradixSort(Record[] A, int L, int U, int k) {
   int countLess = new int[ALPHA+1];

   Sort A[L..U] stably by the kth character of each key, and for each
      digit, c, set countLess[c] to the number of records in A
      whose kth character comes before c in alphabetical order.

   for (int i = 0; i <= ALPHA; i += 1)
      if (countLess[i+1] - countLess[i] > 1)
         MSDradixSort(A, L + countLess[i],
                      L + countLess[i+1] - 1, k+1);
}
```

A	posn
⋆ set, cat, cad, con, bat, can, be, let, bet	0
⋆ bat, be, bet / cat, cad, con, can / let / set	1
bat / ⋆ be, bet / cat, cad, con, can / let / set	2
bat / be / bet / ⋆ cat, cad, con, can / let / set	1
bat / be / bet / ⋆ cat, cad, can / con / let / set	2
bat / be / bet / cad / can / cat / con / let / set	

Figure 8: An example of an MSD radix sort on the same data as in Figure 7. The first line shows the initial contents of A and the last shows the final contents. Partially-sorted segments that agree in their initial characters are separated by single slash (/) characters. The ⋆ character indicates the segment of the array that is about to be sorted and the `posn` column shows which character position is about to be used for the sort.

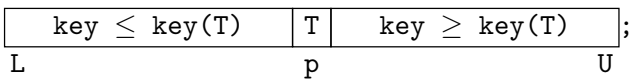
---

11 Selection

Consider the problem of finding the *median* value in an array – a value in the array with as many array elements less than it as greater than it. A brute-force method of finding such an element is to sort the array and choose the middle element (or *a* middle element, if the array has an even number of elements). However, we can do substantially better.

The general problem is *selection* – given a (generally unsorted) sequence of elements and a number k, find the k^{th} value in the sorted sequence of elements. Finding a median, maximum, or minimum value is a special case of this general problem. Perhaps the easiest efficient method is the following simple adaptation of Hoare's quicksort algorithm.

```
/** Assuming 0<=k<N, return a record of A whose key is kth smallest
 *  (k=0 gives the smallest, k=1, the next smallest, etc.). A may
 *  be permuted by the algorithm. */
Record select(Record[] A, int L, int U, int k) {
    Record T = some member of A[L..U];
    Permute A[L..U] and find p to establish the partitioning
      condition:
```




```
    if (p-L == k)
        return T;
    if (p-L < k)
        return select(A, p+1, U, k - p + L - 1);
    else
        return select(A, L, p-1, k);
}
```

The key observation here is that when the array is partitioned as for quicksort, the value `T` is the $(p-L)$st smallest element; the $p-L$ smallest record keys will be in `A[L..p-1]`; and the larger record keys will be in `A[p+1..U]`. Hence, if $k < p-L$, the k^{th} smallest key is in the left part of `A` and if $k > p$, it must be the $(k-p+L-1)$st largest key in the right half.

Optimistically, assuming that each partition divides the array in half, the recurrence governing cost here (measured in number of comparisons) is

$$\begin{aligned} C(1) &= 0 \\ C(N) &= N + C(\lceil N/2 \rceil). \end{aligned}$$

where $N = U - L + 1$. The N comes from the cost of partitioning, and the $C(\lceil N/2 \rceil)$ from the recursive call. This differs from the quicksort and mergesort recurrences by the fact that the multiplier of $C(\cdots)$ is 1 rather than 2. For $N = 2^m$ we get

$$\begin{aligned} C(N) &= 2^m + C(2^{m-1}) \\ &= 2^m + 2^{m-1} + C(2^{m-2}) \\ &= 2^{m+1} - 1 = 2N - 1 \\ &\in \Theta(N) \end{aligned}$$

This algorithm is only probabilistically good, just as was quicksort. There are selection algorithms that *guarantee* linear bounds, but we'll leave them for CS170.