Sorting Algorithms: Internal and External

To make introduction into the area of sorting algorithms, the most appropriate are "elementary" methods. They provide an easy way to learn terminology and basic mechanism for sorting algorithms giving an adequate background for more sophisticated sorts. Furthermore, in a great many applications of sorting it's better to use these simple methods then the more powerful general-purpose methods. Finally, some of these simple methods can be used to improve the efficiency of more powerful ones.

The best known sorting methods are selection, insertion and bubble sorting algorithms.

Selection sorting works according to the prescript:

- first find the smallest element in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, etc.

The following program is a full implementation of this process. For each i from 1 to N-1, it exchanges the minimum element in array with an element in i's position:

```
procedure selection;
var i,j,min, t: integer ;
begin
    for i:=1 to N -1 do
    begin
    min :=i ;
    for j :=i+1 to N do
    if a [j]<a [min] then min :=j ;
    t:=a [min]; a [min]:=a [i]:=t
    end;
end;
```

Insertion and bubble sorting methods are almost as simple as selection sorting is, but they are more flexible.

Insertion sorting:

```
procedure insertion;
    var i,j,v:integer;
begin
    for i:=2 to N do
    begin
    v:=a[i]; j:=i;
    while a[j-1]>v do
    begin a[j]:=a[j-1]; j:=j-1 end;
    a[j]:=v
    end
end;
```

Bubble sorting:

```
    procedure bubble;
        var i,j,t: integer;
    begin
        for i:=N downto 1 do
        for j:=2 to i do
        if a[j-1]>a[j] then
        begin t:=a[j-1]; a[j-1]:=a[j]; a[j]:=t  end
```

end;

Performance characteristics of these methods:

- selection sort uses about $N^2/2$ comparisons and N exchanges;
- insertion sort uses $N^2/4$ comparisons and $N^2/8$ exchanges on the average, twice as many in the worst case;
- bubble sort uses $N^2/2$ comparisons and $N^2/2$ exchanges on the average and in the worst case.

Sorting files with large records

It is possible to arrange things so that any sorting method uses only N exchanges of full records, by having the algorithm operate indirectly on the file. The additional array of indices may be considered and the rearrangement may be done afterwards.

```
procedure insertion;
  var i, j, v: integer;
  begin
       for i:=l to N do p [i ]:=i;
       for i:=2 to N do
         begin
         v:=p [i ]; j:=i;
       while a[p[j-1]]>a[vl do
         begin p [j]:=p [j- 1]; j:=j - 1 end;
    p [j]:=v
    end
end;
```

Specifically, if the array ***a[1…N]*** consists of large records, then we prefer to manipulate a "pointer array" ***p[1…N]*** accessing the original array only for comparisons. If we define ***p [i] = i*** initially, then the algorithms above need only be modified to refer to a ***p[i]*** rather than ***a[i]***. This produces an algorithm that will "sort" the index array so that ***p[1]*** is the index of the smallest element in ***a***, ***p[2]*** is the index of the second smallest element in ***a***, etc. and the cost of moving large records around excessively is avoided. The code above shows how insertion sort might be modified to work in this way.

We are dealing with two files. Let's consider the situation when there isn't enough room for another copy of the file. Then the rearrangements may be made in-site:

```
procedure insitu;
     var i, j, k, t: integer;
     begin
     for i:=l to N do
       if p[i]<>i then
          begin
          t:=a[i]; k:=i;
          repeat
            j:=k; a[j]:=a[p[j]];
            k:=p[j]; p[j]:=j;
          until k=i;
          a [j]:=t
          end;
     end;
```

The viability of this technique for particular applications of course depends on the relative size of records and keys in the file to be sorted. Certainly one would not go to such trouble for a file consisting of small records, because of the extra space required for the index array and the extra time required for the indirect comparisons. But for files consisting of large records, it is almost always desirable to use an indirect sort, and in many applications it may not be necessary to move the data at all. Of course, for files with very large records, plain selection sort is the method to use.

Because of the availability of this indirect approach, the conclusions we draw and those which follow when comparing methods to sort files of integers are likely to apply to more general situations.

Shellsort

Insertion sort is slow because it exchanges only adjacent elements. For example, if the smallest element happens to be at the end of the array, ***N*** steps are needed to get it where it belongs. ***Shellsort*** is a simple extension of insertion sort which gains speed by allowing exchanges of elements that are far apart.

The idea is to rearrange the file to give it the property that taking every h^{th} element (starting anywhere) yields a sorted file. Such a file is said to be ***h-sorted***. Put another way, an h-sorted file is h independent sorted files, interleaved together. By h-sorting for some large values of h, we can move elements in the array long distances and thus make it easier to h-sort for smaller values of h. Using such a procedure for any sequence of values of h which ends in I will produce a sorted file: this is Shellsort.

One way to implement ***Shellsort*** would be, for each h, to use insertion sort independently on each of the h subfiles. But it turns out to be much easier than that: If we replace every occurrence of "1" by "h" (and "2" by "h+l") in insertion sort, the resulting program h-sorts the file, as follows.

```
  procedure shellsort;
      label 0;
      var i, j, h, v: integer;
      begin
      h:=l; repeat h:=3*h+1 until h>N;
      repeat
      h:=h div 3;
      for i:=h+l to N do
      begin
        v:=a [i ]; j:=i;
        while a[j-h]>v do
        begin
              a[j]:=a[j-h]; j:=j-h;
              if j< = h then goto 0
        end;
0:  a [j]:=v
      end
      until h=l;
      end;
```

This program uses the increment sequence 1093, 364, 121, 40, 13, 4, 1. Other increment sequences might do about as well as this in practice, but some care must be exercised. The increment sequence in this program is easy to use and leads to an efficient sort. Many other increment sequences may lead to a more efficient sort, as well as to a bad sorting sequences.

The description is imprecise because of no one is able to analyse this algorithm in a theoretical way. There is no analytical comparison to other methods too.

Nevertheless for the sequence given:

Statement: *Shellsort* never does more than $N^{3/2}$ comparisons.

Distribution Counting

A very special situation for which there is a simple sorting algorithm is the following: *"sort a file of N records whose keys are distinct integers between 1 and N."*

This problem can be solved using a temporary array t with the statement:

for i:=l ***to*** N ***do*** t[a[i]]:=a [i].

A more realistic problem in the same spirit is:

"sort a file of N records whose keys are integers between 0 and M - 1."

If *M* is not too large, an algorithm called *distribution counting* can be used to solve this problem. The idea is to count the number of keys with each value and then use the counts to move the records into position on a second pass through the file, as in the following code:

```
for j:=0 to M-1 do count[j]:=0;
for i:=l to N do
  count [a [i ] ]:=count [a [i j ]+l;
for j:=l to M- 1 do
  count [j]:=count [j- 1]+count [j];
for i:=N downto 1 do
    begin
    b [count [a [i ]]]:=a [i];
    count [a [i ] ]:=count [a [i ] ]-1
    end;
  for i:=l to N do a [i ]:=b [i ];
```

This method will work very well for the type of files postulated. Furthermore, it can be extended to produce a much more powerful method - **radix sorting**.

Quicksort algorithm

The ***quick-sort*** algorithm, introduced in 1962 by C.A.R.Hoare, belongs to the class of so-called *divide-and-conquer* algorithms, similar as the merge-sort too. It is popular because of it's not difficult to implement, it's a good "general-purpose" sort (it works in a variety of situations), and it consumes fewer resources than any other sorting algorithm in many situations.

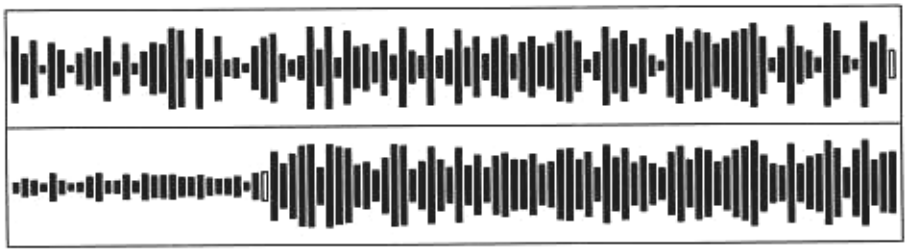
```
procedure quicksort(l, r: integer);
var i: integer;
begin
        if r>l then
        begin
        i:=partition (l, r)
        quicksort(l, i-1);
        quicksort(i+1, r);
        end
end;
```

It works by partitioning a file into two parts, then sorting the parts independently. The exact position of the partition depends on the file, and the algorithm has the following recursive structure:

If the partition method can be made more precise, (for example, choosing right-most element as a partition element), and the recursive call of this procedure may be eliminated, then the implementation of the algorithm looks like this:

```
procedure quicksort (l,r: integer);
var v, t, i, j: integer;
begin
      if r>l then
      begin
      v:=a[r]; i:=l-1; j:=r;
      repeat
      repeat i:=i+1 until a[i]=>v;
      repeat j:=j-1 until a[j]=<v;
      t:=a[i]; a[i]:=a[j]; a[j]:=t;
      until j=<i;
      a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
      quicksort (l,i-1);
      quicksort (i+1,r)
      end
end;
```

An example of a partitioning of a larger file (choosing the right-most element):



The crux of the method is the partition procedure, which must rearrange the array to make the following conditions hold:

- the element a[i] is in its final place in the array for some i;
- all the elements left to the a[i] are less then or equal to it;
- all the elements right to the a[i] are greater then or equal to it.

The first improvement: Removing Recursion

The recursion may be removed by using an ***explicit pushdown*** stack, which is containing "work to be done" in the form of subfiles to be sorted. Any time we need a subfile to process, we pop the stack. When we partition, we create two subfiles to be processed which can be pushed on the stack. This leads to the following nonrecursive implementation:

```
procedure quicksort;
    var t, i, 1, r: integer;
    begin
    l:=1; r:=N; stackinit;
    push (l); push (r);
```

```
repeat
  if r>l then
      begin
      i:=partition (l, r);
      if (i-l)>(r-i)
then begin push(l); push(i-1); l:=i+l end
else begin push(i+l); push(r); r:=i-I end;
end
else
begin r:=pop; l:=pop end;
until stackempty;
end;
```

This program differs from the description above in two important ways:

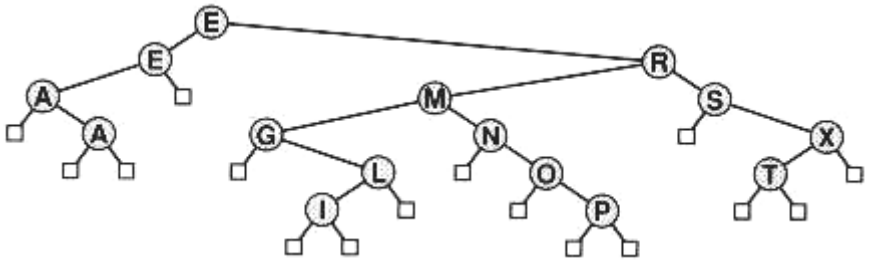
- first, the two subfiles are not put on the stack in some arbitrary order, but their sizes are checked and the larger of the two is put on the stack first;
- second, the smaller of the two subfiles is not put on the stack at all; the values of the parameters are simply reset.

For Quicksort, the combination of end-recursion removal and the policy of processing the smaller of the two subfiles first turns out to ensure that the stack need contain room for only about ***lgN*** entries, since each entry on the stack after the top one must represent a subfile less than half the size of the previous entry.

This is in sharp contrast to the size of the stack in the worst case in the recursive implementation, which could be as large as ***N*** (for example, when the file is already sorted). This is a subtle but real difficulty with a recursive implementation of Quicksort: there's always an underlying stack, and a degenerate case on a large file could cause the program to terminate abnormally because of lack of memory, behavior obviously undesirable for a library sorting routine.

The simple use of an explicit stack in the program above leads to a far more efficient program than the direct recursive implementation, but there is still overhead that could be removed. The problem is that, if *both* subfiles have only one element, an entry with ***r=l*** is put on the stack only to be immediately taken off and discarded. It is straightforward to change the program so that it puts no such files on the stack.

Tree diagram of the partitioning process for "a sorting example":



The second improvement: Small Subfiles

The recursive program is to call itself for many small files, so it should use as good a method as possible when small files are encountered. One obvious way to do this is to change the sorting procedure:

instead of "**if** *r > l* **then**" to call "**if** *r - l <= M* **then** *insertion (l, r)*"

The parameter M depends upon the implementation. The value choosen for M need not be the best possible: the algorithm works the same for the value about 5 - 25. The improvement in the running time is on the order of 20% for most applications.

The third improvement: Partitioning

There are several possibilities to use a better partitioning element. The ***safest*** choice to avoid the worst case would be a random element from the array for a partitioning element. Then the worst case will happen with negligibly small probability. This is a simple example of a "*probabilistic algorithm*" one which uses randomness to achieve good performance almost always, regardless of the arrangement of the input.

Another useful improvement is to take three elements from the file, then use the median of the three for the partitioning element. If the three elements chosen are from the *left, middle,* and *right* of the array, then the use of sentinels can be avoided as follows: sort the three elements, then exchange the one in the middle with *a[r-1],* and then run the partitioning algorithm on *a[l+1 … r-2]*. This improvement is called the *median-of'-three* partitioning method.

The median-of-three method helps Quicksort in three ways. First, it makes the worst case much more unlikely to occur in any actual sort. In order for the sort to take N2 time, two out of the three elements examined must be among the largest or among the smallest elements in the file, and this must happen consistently through most of the partitions. Second, it eliminates the need for a sentinel key for partitioning, since this function is served by the three elements examined before partitioning. Third, it actually reduces the total average running time of the algorithm by about 5%.

The combination of a nonrecursive implementation of the median-of-three method with a cutoff for small subfiles can improve the running time of Quicksort over the naive recursive implementation by 25% to 30%.

Further algorithmic improvements are possible (for example, the median of five or more elements could be used), but the amount of time gained will be marginal. More significant time savings can be realized (with less effort) by coding the inner loops (or the whole program) in assembly or machine language. Neither path is recommended except for experts with serious sorting applications.

Order Statistics

Sorting programs are often used for applications in which a full sort is not necessary. For example, suppose one wanted to find the median of a set of numbers. One way to proceed would be to sort the numbers and look at the middle one, but we can do better.

The operation of finding the median is a special case of the operation of *selection:*
find the k^{th} smallest of a set of numbers.

Since an algorithm cannot guarantee that a particular item is the k^{th} smallest without having examined and identified the $k - 1$ elements which are smaller and the $N - k$ elements which are larger, most selection algorithms can return all of the k smallest elements of a file without a great deal of extra calculation.

Selection has many applications in the processing of experimental and other data. The use of the median and other *order statistics* to divide a file up into smaller groups is very common. Often only a small part of a large file is to be saved for further processing; in such cases, a program which

can select, say, the top ten percent of the elements of the file might be more appropriate than a full sort.

An algorithm which can be directly adapted to selection (if *k* is very small) - *selection sort* will work very well, requiring time proportional *to Nk:*

first find the smallest element, then find the second smallest by finding the smallest of the remaining items, etc.

For slightly larger k, priority queues can be immediately adapted to run in time proportional to *N log k.*

An interesting method which adapts well to all values of *k* and runs in linear time on the average can be formulated from the partitioning procedure. The partitioning method rearranges an array *a[1..N]* and returns an integer *i* such that *a[1], . . ., a [i-1]* are less than or equal to *a[i]* and *a[i+1], . . ., a[N]* are greater than or equal to *a[i]*. If the k^{th} smal lest element in the file is sought and we have *k=i,* then we're done. Otherwise, if $k < i$ then we need to look for the k^{th} smallest element in the left subfile, and if $k > i$ then we need to look for the $(k-i)^{th}$ smallest element in the right subfile. Adjusting this argument to apply to finding the k^{th} smallest element in an array *a[l.. r]* leads immediately to the following recursive formulation.

```
procedure select(l.r,k: integer);
var i: integer;
      begin
      if r>l then
        begin
        i:=partition(l,r);
        if i>l+k-1 then select(l,i-l,k);
        if i<l+k-1 then select (i+l, r, k - i);
        end
end;
```

This procedure rearranges the array so that *a[1] … a[k - l]* are less than or equal to *a[k]* and *a[k+1], …, a [r]* are greater than or equal to a *[k]*. For example, the call *select(1, N, (N+1)**div** 2)* partitions the array on its median value.

Since the *select* procedure always ends with only one call on itself, it is not really recursive in that no stack is needed to remove the recursion: when the time comes for the recursive call, we can simply reset the parameters and go back to the beginning, since there is nothing more to do. Also, we can eliminate the simple calculations involving *k*, as in the following implementation.

```
procedure select(k: integer);
  var v, t, i, j, l, r: integer;
  begin
  l:=1; r:=N;
  while r>l do
  begin
       v:=a[r]; i:=l-1; j:=r;
       repeat
       repeat i:=i+l until a[i]>=v;
       repeat j:=j-1 until a[j]<=v;
       t:=a[i]; a[i]:=a[j]; a[j]:=t;
       until j< =i;
```

```
        a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
        if i>=k then r:=i-1;
        if i<=k then l:=i+l;
      end;
  end
```

We can (*very roughly*) argue that, on a very large file, each partition should roughly split the array in half, so the whole process should require about

$$N + N/2 + N/4 + N/8 + \ldots = 2N$$

Comparisons, and this rough argument is not too far from the truth.

Property: *Quicksort-based selection is linear--time on the average.*

A significantly more complex analysis leads to the result that the average number of comparisons is about

$$2N + 2k \ln(N/k) + 2(N - k) \ln(N/(N - k),$$

which is linear for any allowed value of *k*. For $k = N/2$ (finding the median), this evaluates to about $(2 + 2 \ln 2)N$ comparisons.

Radix Sorting

The "keys" used to define the order of the records for files for many sorting applications can be very complicated. For many applications, however, it is possible to take advantage of the fact that the keys can be thought of as numbers from some ***restricted range***. Sorting methods which take advantage of the digital properties of these numbers are called *radix sorts.* These methods do not just compare keys: they process and compare *pieces of keys*.

Radix-sorting algorithms treat the keys as numbers represented in a ***base-M*** number system, for different values of M *(the radix)*, and work with individual digits of the numbers. Of course, with most computers it's more convenient to work with ***M = 2*** (or some power of 2).

Anything that's represented inside a digital computer can be treated as a binary number, so many sorting applications can be recast to make feasible the use of radix sorts operating on keys which are binary numbers. Unfortunately, Pascal and many other languages intentionally make it difficult to write a program that depends on the binary representation of numbers.

Fortunately, it's not too difficult to use arithmetic operations to simulate the operations needed, and so here we'll be able to write (inefficient) Pascal programs to describe the algorithms that can be easily translaled to efficient programs in programming languages that support bit operations on binary numbers.

Bits

Given a (key represented as a) binary number, the fundamental operation needed for radix sorts is extracting a contiguous set of bits from the number. Suppose we are to process keys which we know to be integers between 0 and 1000.

We may assume that these are represented by ten-bit binary numbers. In machine language, bits are extracted from binary numbers by using bitwise "*and*" operations and shifts. For example, the leading two bits of a ten-bit number are extracted by shifting right eight bit positions, then doing a bitwise "*and*" with the mask 0000000011. In Pascal, these operations can be simulated with **div** and **mod.** For example, two bits of a ten-bitnumber *x* are given by (*x* **div** 256) **mod** 4.

In general, "*shift x right k bit positions*" can be simulated by computing *x* **div** 2^k and "*zero all but the j rightmost bits of x*" can be simulated by computing *x* **mod** 2^j. In our description of the radix-sort algorithms, we'll assume the existence of a

function *bits (x, k, j: integer): integer*

which combines these operations to return the *j* bits which appear *k* bits from the right in *x* by computing (*x* **div** 2^k) **mod** 2^j. This function can be made efficient by precomputing (or defining as constants) the powers of 2. Many Pascal implementations have extensions to the language which allow these operations to be specified somewhat more directly.

Armed with this basic tool, we'll consider two types of radix sorts which differ in the order in which they examine the bits of the keys. We assume that the keys are not short, so that it is worthwhile to go to the effort of extracting their bits. If the keys are short, then the *distribution-counting* method can be used. Recall that this method can sort *N* keys known to be integers between *0* and *M - 1* in linear time, using one auxiliary table of size *M* for counts and another of size *N* for rearranging records. Thus, if we can afford a table of size 2^b, then b-bit keys can easily be sorted in linear time. Radix sorting comes into play if the keys are sufficiently long (say *b = 32*) that this is not possible.

The first basic method for radix sorting that we'll consider examines the bits in the keys from left to right. It is based on the fact that the outcome of "comparisons" between two keys depends only on the value of the bits in the first position at which they differ (reading from left to right). Thus, all keys with leading bit 0 appear before all keys with leading bit 1 in the sorted file; among the keys with leading bit 1, all keys with second bit 0 appear before all keys with second bit 1, and so forth. The left-to-right radix sort, which is called *radix exchange sort***,** sorts by systematically dividing up the keys in this way.

The second basic method is called *straight radix sort, and* examines the bits in the keys from *right to left*. It is based on an interesting principle that reduces a sort on b-bit keys to b sorts on 1-bit keys. This can be combined with distribution counting to produce a sort that runs in linear time under quite generous assumptions.

Radix Exchange Sort

Suppose we can rearrange the records of a file so that all those whose keys begin with a 0 bit come *before* all those whose keys begin with a 1 bit. This immediately defines a recursive sorting method: if the two subfiles are sorted independently, then the whole file is sorted:

```
procedure radixexchange (l, r, b: integer);
   var t, i, j: integer;
   begin
   if (r>l) and (b>=0) then
      begin
      i:=l; j:=r;
      repeat
        while (bits(a[i], b, 1)=0) and (i<j) do i:=i+l;
        while (bits (a[j], b, 1)=1) and (i <j) do j:=j - 1;
        t:=a[i]; a[i]:=aU]; all]:=t;
      until j=i;
      if bits (a[r], b, 1)=0 then j:=j+l;
      radixexchange (l, j-1, b - 1);
      radixexchange (j, r, b - 1)
```

```
    end
end;
```

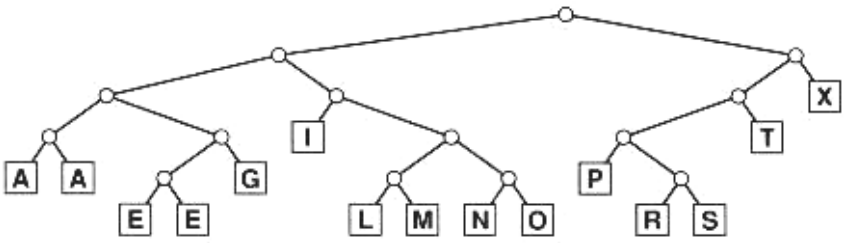
For simplicity, assume that *a[1 … N]* contains positive integers less than 2^{32} (so that they can be represented as 31-bit binary numbers). Then *radixexchange (1, N, 30)* will sort the array. The variable *b* keeps track of the bit being examined, ranging from *30* (leftmost) down to *0* (rightmost).

One serious potential problem for radix sort is that degenerate partitions (*partitions with all keys having the same value for the bit being used*) can happen frequently. This situation arises commonly in real files when small numbers (with many leading zeros) are being sorted. It also occurs for characters: for example, suppose that *32-bit keys are made up from four characters* by encoding each in a standard eight-bit code and then putting them together. Then degenerate partitions are likely to occur at the beginning of each character position, since for example, lower-case letters all begin with the same bits in most character codes. Many other similar effects are obviously of concern when sorting encoded data.

It can be seen from the analysis of the algorithm, that once a key is distinguished from all the other keys by its left bits, no further bits are examined. This is a distinct advantage in some situations, a disadvantage in others. When the keys are truly random bits, each key should differ from the others after about *lgN* bits, which could be many fewer than the number of bits in the keys.

On the other hand, notice that all the bits of equal keys are examined. Radix sorting simply does not work well on files which contain many equal keys. Radix-exchange sort is actually slightly faster than Quicksort if the keys to be sorted are comprised of truly random bits, but Quicksort adapts better to less random situations.

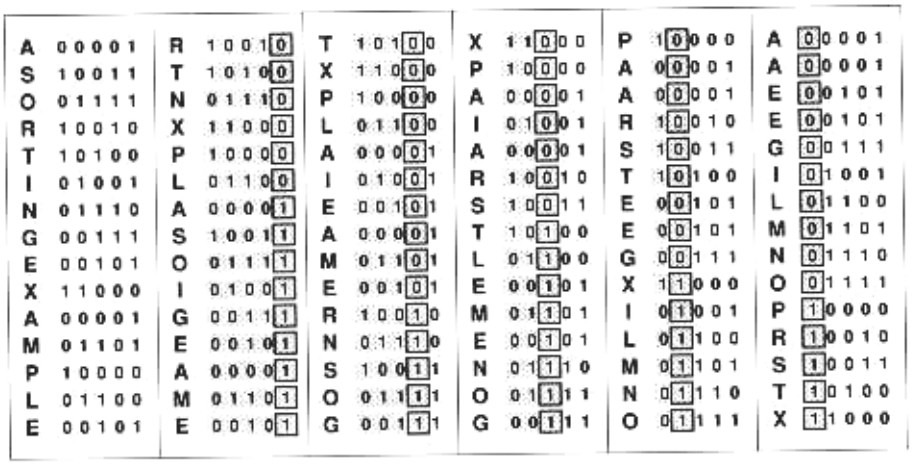
Figure below gives the tree that represents the partitioning process for radix exchange sort, and this tree may be compared with tree for Quicksort (above):



The basic recursive implementation given above can be improved by removing recursion and treating small subfiles differently.

Straight Radix Sort

An alternative radix-sorting method is to examine the bits from right to left. This is the method used by old computer-card-sorting machines: a deck of cards was run through the machine 80 times, once for each column, proceeding from right to left. Figure below shows how a right-to-left bit-by-bit radix sort works on the file of sample keys:



It's not easy to be convinced that the method works; in fact it doesn't work at all unless the one-bit partitioning process is ***stable***. Once stability has been identified as being important, a trivial proof that the method works can be found: after putting keys with i^{th} bit 0 before those with i^{th} bit I (in a stable manner), we know that any two keys appear in proper order (on the basis of the bits so far examined) in the file either because their i^{th} bits are different, in which case partitioning puts them in the proper order, or because their i^{th} bits are the same, in which case they're in proper order because of stability.

The requirement of stability means, for example, that the partitioning method used in the radix-exchange sort can't be used for this right-to-left sort. The partitioning is like sorting a file with only two values, and the distribution counting sort is entirely appropriate for this. If *M = 2* in the distribution counting program and replace *a[i] by bits(a[i], k, 1)*, then that program becomes a method for sorting the elements of the array *a* on the bit *k* positions from the right and putting the result in a temporary array *t*. But there's no reason to use *M = 2*, in fact, we should make *M* as large as possible, realizing that we need a table of *M* counts. This corresponds to using *m* bits at a time during the sort, with $M = 2^m$. Thus, straight radix sort becomes little more than a generalization of distribution-counting sort:

```
procedure straightradix;
  var i, j, pass: integer;
  count: array [0..M] of integer;
  begin
  for pass:=0 to (w div m) - 1 do
     begin
     for j:=0 to M-1 do count [j]:=0;
     for i:=l to N do
       count[bits(a[i], pass*m, m)]:=count[bits(a[i], pass*m, m)]+1;
     for j:=l to M-1 do
       count [j]:=count [j-1] + count [j];
     for i:=N downto 1 do
        begin
        b[count[bits(a[i], pass*m, m)]]:=a[i];
        count[bits(a[i], pass*m, m)]:=count[bits(a [i], pass*m, m)] -1;
        end;
     for i:=l to N do a[i]:=b[i];
     end;
  end;
```

For clarity, this procedure uses two calls on *bits* to increment and decrement count when one would suffice. Also, the correspondence $M = 2^m$ has been preserved in the variable names.

The procedure above works properly only if *w* is a multiple of *m*. Normally, this is not a particularly restrictive assumption for radix sort: it simply corresponds to dividing the keys to be sorted into an integral number of equal-size pieces. When *m=w* we have distribution counting sort; when *m=l* we have straight radix sort, the right-to-left bit-by-bit radix sort described in the example above.

Performance Characteristics of Radix Sorts

The running times of both basic radix sorts for sorting *N* records with *b*-bit keys are essentially *Nb*. On the one hand, one can think of this running time as being essentially the same as *N log n*, since if the numbers are all different, *b* must be at least *log N*. On the other hand, both methods usually use many fewer than *Nb* operations: the left-to-right method because it can stop once differences between keys have been found, and the right-to-left method because it can process many bits at once.

Property 1: Radix-exchange sort uses on the average about *N lg N* bit com*parisons.*

Property 2: Both radix sorts use less than *Nb* bit comparisons to sort *N* b -bit keys.

Property 3: Straight radix sort can sort *N* records with *b*-bit keys in *b/m* passes, using extra space for 2^m counters (and a buffer for rearranging the file).

A Linear Sort

The straight radix sort implementation makes *b/m* passes through the file. By making *m* large, we get a very efficient sorting method, as long as we have $M = 2^m$ words of memory available. A reasonable choice is to make *m* about one quarter the word-size (*b/4*), so that the radix sort is four distribution counting passes.

The keys are treated as base-M numbers, and each (base-M) digit of each key is examined, but there are only four digits per key. (This corresponds directly to the architectural organization of many computers: one typical organization has 32-bit words, each consisting of four 8-bit bytes. The *bits* procedure then winds up extracting particular bytes from words in this case, which obviously can be done very efficiently on such computers.) Now, each distribution-counting pass is linear, and since there are only four of them, the entire sort is linear, certainly the best performance we could hope for in a sort.

In fact, it turns out that we can get by with only two distribution counting passes. (Even a careful reader is likely to have difficulty telling right from left by this time, so some effort may be necessary to understand this method.) We do this by taking advantage of the fact that the file will be *almost* sorted if only the leading *b/2* bits of the b-bit keys are used.

The sort can be completed efficiently by using insertion sort on the whole file afterwards. This method is obviously a trivial modification to the implementation above: to do a right-to-left sort using the leading half of the keys, we simply start the outer loop *at pass=b* **div** (2*m) rather than pass=l. Then a conventional insertion sort can be used on the nearly ordered file that results.

Using two distribution counting passes (with *m* about one-fourth the word size) and then using insertion sort to finish the job will yield a sorting method that is likely to run faster than any of the others we've seen for large files whose keys are random bits. Its main disadvantage is that it

requires an extra array of the same size as the array being sorted. It is possible to eliminate the extra array using linked-list techniques, but extra space proportional to *N* (for the links) is still required.

A linear sort is obviously desirable for many applications, but there are reasons why it is not the panacea that it might seem:

- First, its efficiency really does depend on the keys being random bits, randomly ordered. If this condition is not satisfied, severely degraded performance is likely.
- Second, it requires extra space proportional to the size of the array being sorted.
- Third, the "inner loop" of the program actually contains quite a few instructions, so even though it's linear, it won't be as much faster than Quicksort (say) as one might expect, except for quite large files (at which point the extra array becomes a real liability).

The choice between Quicksort and radix sort is a difficult one that is likely to depend not only on features of the application, such as key, record, and file size, but also on features of the programming and machine environment that relate to the efficiency of access and use of individual bits. Again, such tradeoffs need to be studied by an expert and this type of study is likely to be worthwhile only for serious sorting applications.

Mergesort

The ***merge sort*** algorithm is closely following the divide-and-conquer paradigm. Intuitively, it operates as follows:

- ***Divide***: Divide the ***n***-element sequence to be sorted into two subsequences of (***n/2***) elements each.
- ***Conquer:*** Sort the two subsequences recursively using mergesort.
- ***Combine:*** Merge the two sorted subsequences to produce the sorted answer.

Suppose we have two sorted arrays *a[1…N], b[1…M]*, and we need to merge them into one. The mergesort algorithm is based on a ***merge*** procedure, which can be presented as follows:

```
procedure merge (a, b);
var a,b,c: array [1..M + N)] of integer;
begin
i:=1; j:=1;
a[M+1]:=maxint; b[N+1]:=maxint;
for k:=1 to M+N do
        if a[i]<b[j]
        then begin c[k]:=a[i]; i:=i+1 end
        else begin c[k]:=b[j]; j:=j+1 end;
end;
```

The above procedure works with arrays as an input data. If the input data are lists, then the output can be a list too:

```
type link=^node;
node=record key: integer; next: link end;
var t,z:link; N: integer;
function merge(a,b:link):link;
var c:link;
begin c:=z;
        repeat
```

```
        if a^.key=<b^.key
        then begin c^.next:=a; c:=a; a:=a^.next end
        else begin c^.next:=b; c:=b; b:=b^.next end
        until c^.key=maxint;
        merge:=z^.next; z^.next:=z;
end;
```

To perform the ***mergesort***, the algorithm for arrays can be as follows:

```
procedure mergesort(l,r: integer);
var i,j,k,m: integer;
begin if r-l>0 then
        begin m:=(r+l) div 2;
        mergesort(l,m); mergesort(m+1,r);
        for i:=m downto l do b[i]:=a[i];
        for j:=m+1 to r do b[r+m+1-j]:=a[j];
        for k:=l to r do if b[i]<b[j]
        then begin a[k]:=b[i]; i:=i+1 end
        else begin a[k]:=b[j]; j:=j-1 end;
        end;
end;
```

or for lists:

```
function mergesort(c:link):link;
var a,b:link;
begin if c^.next=z then mergesort:=c else
        begin
        a:=c; b:=c^.next; b:=b^.next; b:=b^.next;
        while b<>z do begin c:=c^.next; b:=b^.next; b:=b^.next end;
        b:=c^.next; c^.next:=z;
        mergesort:=merge(mergesort(a),mergesort(b));
        end;
end;
```

Property 1: Mergesort requires about *N lg N* comparisons to sort any file of *N* elements.

Property 2: Mergesort is stable.

Property 3: Mergesort is insensible to the initial order of its input.

External Sorting

Many important sorting applications involve processing very large files, much too large to fit into the primary memory of any computer. Methods appropriate for such applications are called ***external*** methods, since they involve a large amount of processing external to the central processing unit.

There are two major factors which make external algorithms quite different:

- First, the cost of accessing an item is orders of magnitude greater than any bookkeeping or calculating costs.

- Second, over and above with this higher cost, there are severe restrictions on access, depending on the external storage medium used: for example, items on a magnetic tape can be accessed only in a sequential manner.

The wide variety of external storage device types and costs makes the development of external sorting methods very dependent on current technology. These methods can be complicated, and many parameters affect their performance: that a clever method might go unappreciated or unused because of a simple change in the technology is a definite possibility in external sorting. For this reason, only general methods will be considered rather than specific implementations.

For external sorting, the "systems" aspect of the problem is certainly as important as the "algorithms" aspect. Both areas must be carefully considered if an effective external sort is to be developed. The primary costs in external sorting are for input-output. A good exercise for someone planning to implement an efficient program to sort a very large file is first to implement an ***efficient program to copy a large file***, then (if that was too easy) implement an efficient program to reverse the order of the elements in a large file. The systems problems that arise in trying to solve these problems efficiently are similar to those that arise in external sorts.

Permuting a large external file in any non-trivial way is about as difficult as sorting it, even though no key comparisons, etc. are required. In external sorting, we are concerned mainly with limiting the number of times each piece of data is moved between the external storage medium and the primary memory, and being sure that such transfers are done as efficiently as allowed by the available hardware.

External sorting methods have been developed which are suitable for the punched cards and paper tape of the past, the magnetic tapes and disks of the present, and emerging technologies such as bubble memories and videodisks. The essential differences among the various devices are the relative size and speed of available storage and the types of data access restrictions.

We'll concentrate on basic methods for sorting on magnetic tape and disk because these devices are in widespread use and illustrate the two fundamentally different modes of access that characterize many external storage systems. Often, modern computer systems have a "storage hierarchy" of several progressively slower, cheaper, and larger memories. Many of the algorithms can be adapted to run well in such an environment, but we'll deal exclusively with "two-level" memory hierarchies consisting of main memory and disk or tape.

Sort-Merge

Most external sorting methods use the following general strategy: make a first pass through the file to be sorted, breaking it up into blocks about the size of the internal memory, and *sort* these blocks. Then *merge* the sorted blocks together by making several passes through the file, creating successively larger sorted blocks until the whole file is sorted. The data is most often accessed in a sequential manner, a property which makes this method appropriate for most external devices. Algorithms for external sorting strive to reduce the number of passes through the file and to reduce the cost of a single pass to be as close to the cost of a copy as possible.

Since most of the cost of an external sorting method is for input-output, we can get a rough measure of the cost of a sort-merge by counting the number of times each word in the file is read or written (the number of passes over all the data). For many applications, the methods involve on the order of ten or fewer such passes. Note that this implies that we're interested in methods that can eliminate even a single pass. Also, the running time of the whole external sort can be easily estimated from the running time of something like the "reverse file copy" exercise.

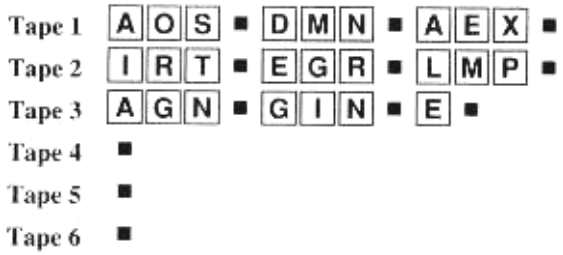
Balanced Multiway Merging

To begin, we'll trace through the various steps of the simplest sort-merge procedure for a small example. Suppose that we have records with the keys

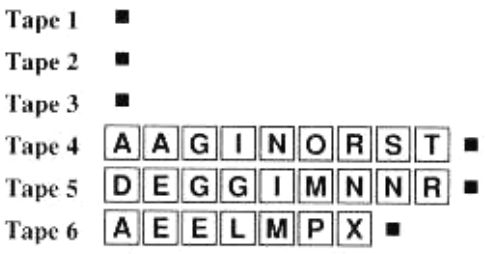
A S 0 R T I N G A N D M E R G I N G E X A M P L E

on an input tape; these are to be sorted and put onto an output tape. Using a "tape" simply means that we're restricted to reading the records sequentially: the second record can't be read until the first is read, and so on. Assume further that we have only enough room for three records in our computer memory but that we have plenty of tapes available.

The first step is to read in the file three records at a time, sort them to make three-record blocks, and output the sorted blocks. Thus, first we read in A S 0 and output the block A 0 S, next we read in R T I and output the block I R T, and so forth. Now, in order for these blocks to be merged together, they must be on different tapes. If we want to do a three-way merge, then we would use three tapes, ending up after the sorting pass with the configuration shown in figure:



Now we're ready to merge the sorted blocks of size three. We read the first record off each input tape (there's just enough room in the memory) and output the one with the smallest key. Then the next record from the same tape as the record just output is read in and, again, the record in memory with the smallest key is output. When the end of a three-word block in the input is encountered, that tape is ignored until the blocks from the other two tapes have been processed and nine records have been output. Then the process is repeated to merge the second three-word block on each tape into a nine-word block (which is output on a different tape, to get ready for the next merge). By continuing in this way, we get three long blocks configured as shown in figure next:



Now one more three-way merge completes the sort. If we had a much longer file with many blocks of size 9 on each tape, then we would finish the second pass with blocks of size 27 on tapes 1, 2, and 3, then a third pass would produce blocks of size 81 on tapes 4, 5, and 6, and so forth. We need six tapes to sort an arbitrarily large file: three for the input and three for the output of each three-way merge. (Actually, we could get by with just four tapes: the output could be put on just one tape, and then the blocks from that tape distributed to the three input tapes in between merging passes.)

This method is called the *balanced multiway erge:* it is a reasonable algorithm for external sorting and a good starting point for the implementation of an external sort. The more sophisticated algorithms below can make the sort run a little faster, but not much. (However, when execution times are measured in hours, as is not uncommon in external sorting, even a small percentage decrease in running time can be quite significant.)

Suppose that we have N words to be manipulated by the sort and an internal memory of size M. Then the "sort" pass produces about *N/M* sorted blocks. (This estimate assumes one-word records: for larger records, the number of sorted blocks is computed by multiplying further by the record size.) If we do P-way merges on each subsequent pass, then the number of subsequent passes is about $log_p (N/M)$, since each pass reduces the number of sorted blocks by a factor of *P*.

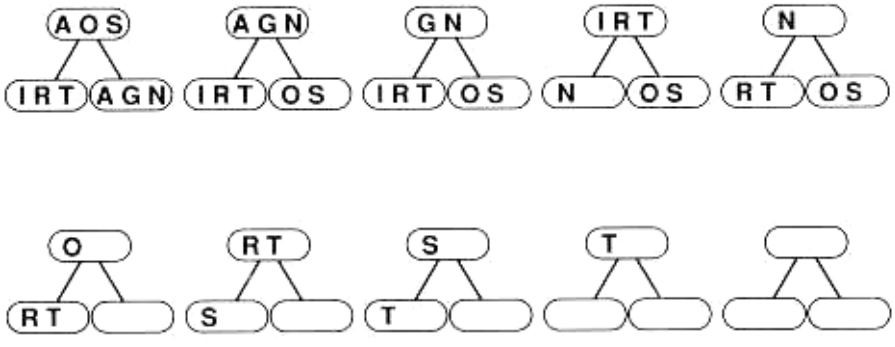
For example, the formula above says that using a four-way merge to sort a 200 million-word file on a computer with a million words of memory should take a total of about five passes. A very rough estimate of the running time can be found by multiplying by five the running time for the reverse file copy implementation suggested above.

Replacement Selection

It turns out that the details of the implementation can be developed in an elegant and efficient way using priority queues. First, we'll see that priority queues provide a natural way to implement a multiway merge. More important, it turns out that we can use priority queues for the initial sorting pass in such a way that they can produce sorted blocks much longer than could fit into internal memory.

The basic operation needed to do P-way merging is repeatedly to output the smallest of the smallest elements not yet output from each of the P blocks to be merged. That smallest element should be replaced with the next element from the block from which it came. The *replace* operation on a priority queue of size P is exactly what is needed. Specifically, to do a P-way merge we begin by filling up a priority queue of size P with the smallest element from each of the P inputs. Then we output the smallest element and replace it in the priority queue with the next element from its block.

The process of merging A 0 S with I R T and A G N (the first merge from our example above), using a heap of size three in the merging process is shown in figure:



The "keys" in these heaps are the smallest (first) key in each node. For clarity, we show entire blocks in the nodes of the heap; of course, an actual implementation would be an indirect heap of pointers into the blocks. First, the A is output so that the O (the next key in its block) becomes the "key" of the root. This violates the heap condition, so that node is exchanged with the node containing A, G, and N. Then that A is output and replaced with the next key in its block, the G. This does not violate the heap condition, so no further change is necessary.

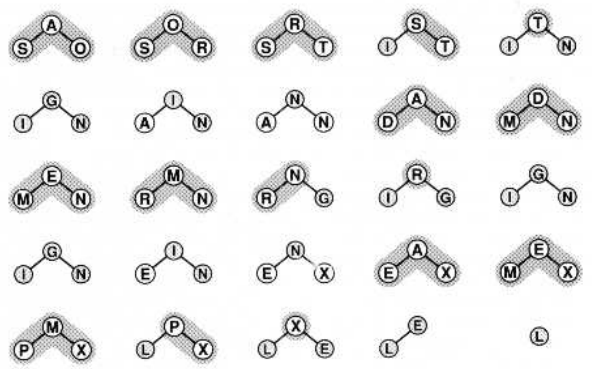
Continuing in this way, we produce the sorted file (read the smallest key in the root node of the trees in figure above to see the keys in the order in which they appear in the first heap position and are output). When a block is exhausted, a sentinel is put on the heap and considered to be larger than all the other keys. When the heap consists of all sentinels, the merge is completed. This way of using priority queues is sometimes called ***replacement selection***.

Thus to do a P-way merge, we can use replacement selection on a priority queue of size P to find each element to be output in log P steps. This performance difference has no particular practical relevance, since a brute-force implementation can find each element to output in P steps and P is normally so small that this cost is dwarfed by the cost of actually outputting the element. The real importance of replacement selection is the way that it can be used in the first part of the sort-merge process: to form the initial sorted blocks which provide the basis for the merging passes.

The idea is to pass the (unordered) input through a large priority queue, always writing out the smallest element on the priority queue as above, and always replacing it with the next element from the input, with one additional ***proviso***:

> if the new element is smaller than the last one output, then, since it could not possibly become part of the current sorted block, it should be marked as a member of the next block and treated as greater than all elements in the current block.

When L marked element makes it to the top of the priority queue, the old block is ended and a new block started. Our example file clearly demonstrates the value of replacement selection. With an internal memory capable of holding only three records, we can produce sorted blocks of size 5, 3, 6, 4, 5, and 2, as illustrated in figure:



As before, the order in which the keys occupy the first position in the heap is the order in which they are output. The shading indicates which keys in the heap belong to which different blocks: an element marked the same way as the element at the root belongs to the current sorted block and the others belong to the next sorted block. The heap condition (first key less than the second and third) is maintained throughout, with elements in the next sorted block considered to be greater than elements in the current sorted block. The first run ends with I N G in the heap, since these keys all arrived with larger keys at the root (so they couldn't be included in the first run), the second run ends with A N D in the heap, etc.

The practical effect of this is to save one merging pass: rather than starting with sorted runs about the size of the internal memory and then taking a merging pass to produce runs about twice the size of the internal memory, we can start right off with runs about twice the size of the internal memory, by using replacement selection with a priority queue of size M. If there is some order in the keys, then the runs will be much, much longer. For example, if no key has more than M larger keys before it in the file, the file will be completely sorted by the replacement selection pass, and no merging will be necessary! This is the most important practical reason to use the method.

In summary, the replacement selection technique can be used for both the "sort" and the "merge" steps of a balanced multiway merge.

Property: A *file of N records can be sorted using an internal memory capable of holding M records and (P + 1) tapes in about* $1 + \log_P (N/2M)$ *passes.*

We first use replacement selection with a priority queue of size *M* to produce initial runs of size about *2M* (in a random situation) or more (if the file is partially ordered), then use replacement selection with a priority queue of size *P* for about $\log_P (N/2M)$ (or fewer) merge passes.

Practical Considerations

To finish implementing the sorting method outlined above, it is necessary to implement the input-output functions which actually transfer data between the processor and the external devices. These functions are obviously the key to good performance for the external sort, and they just as obviously require careful consideration of some systems (as opposed to algorithm) issues. (Readers not concerned with computers at the "systems" level may wish to skim the next few paragraphs.)

A major goal in the implementation should be to overlap reading, writing, and computing as much as possible. Most large computer systems have independent processing units for controlling the large-scale input/output (I/O) devices which make this overlapping possible. The efficiency to be achieved by an external sorting method depends on the number of such devices available.

For each file being read or written, the standard systems programming technique called *double-buffering* can be used to maximize the overlap of I/O with computing. The idea is to maintain two "buffers," one for use by the main processor, one for use by the I/O device (or the processor which controls the I/O device). For input, the processor uses one buffer while the input device is filling the other. When the processor has finished using its buffer, it waits until the input device has filled its buffer, and then the buffers switch roles:

the processor uses the new data in the just-filled buffer while the input device refills the buffer with the data already used by the processor.

The same technique works for output, with the roles of the processor and the device reversed. Usually the I/O time is far greater than the processing time and so the effect of double-buffering is to overlap the computation time entirely; thus the buffers should be as large as possible.

A difficulty with double-buffering is that it really uses only about half the available memory space. This can lead to inefficiency if many buffers are involved, as is the case in P-way merging when P is not small. This problem can be dealt with using a technique called ***forecasting***, which requires the use of only one extra buffer (not P) during the merging process.

Forecasting works as follows. Certainly the best way to overlap input with computation during the replacement selection process is to overlap the input of the buffer that needs to be filled next with the processing part of the algorithm. And it is easy to determine which buffer this is: the next input buffer to be emptied is the one whose *last* item is smallest.

For example, when merging A 0 S with I R T and A G N we know that the third buffer will be the first to empty, then the first. A simple way to overlap processing with input for multiway merging is therefore to keep one extra buffer which is filled by the input device according to this rule. When the processor encounters an empty buffer, it waits until the input buffer is filled (if it hasn't been filled already), then switches to begin using that buffer and directs the input device to begin filling the buffer just emptied according to the forecasting rule.

The most important decision to be made in the implementation of the multiway merge is the choice of the value of P, the "order" of the merge. For tape sorting, when only sequential access is allowed, this choice is easy:

> P must be one less than the number of tape units available, since the multiway merge uses P input tapes and one output tape.

Obviously, there should be at least two input tapes, so it doesn't make sense to try to do tape sorting with less than three tapes.

For disk sorting, when access to arbitrary positions is allowed but is somewhat more expensive than sequential access, it is also reasonable to choose P to be one less than the number of disks available, to avoid the higher cost of nonsequential access that would be involved, for example, if two different input files were on the same disk. Another alternative commonly used is to pick P large enough that the sort will be complete in two merging phases: it is usually unreasonable to try to do the sort in one pass, but a two-pass sort can often be done with a reasonably small P.

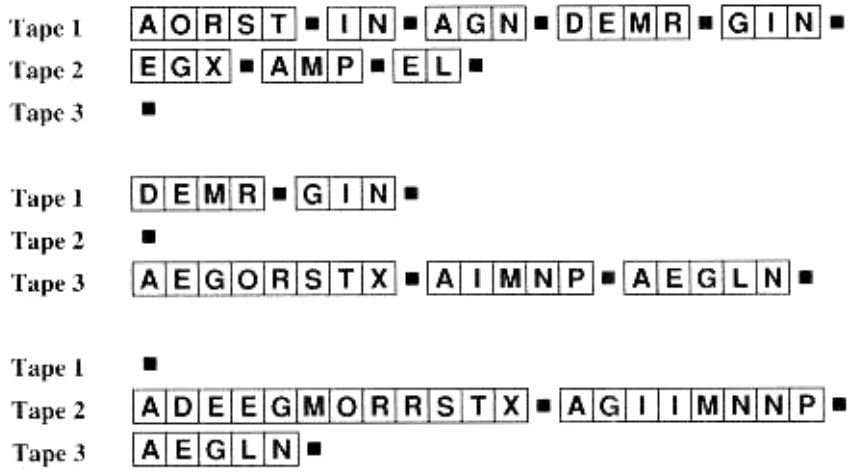
Since replacement selection produces about *N/2M* runs and each merging pass divides the number of runs by P, this means that P should be chosen to be the smallest integer with $P^2 > N/2M$. For our example of sorting a 200-million-word file on a computer with a one-million-word memory, this implies that P = 11 would be a safe choice to ensure a two-pass sort. The best choice between these two alternatives of the lowest reasonable value of P and the highest reasonable value of P is very dependent on many systems parameters: both alternatives (and some in between) should be considered.

Polyphase Merging

One problem with balanced multiway merging for tape sorting is that it requires either an excessive number of tape units or excessive copying. For P-way merging either we must use 2P tapes (P for input and P for output) or we must copy almost all of the file from a single output tape to P input tapes between merging passes, which effectively doubles the number of passes to be about $2 \log_P (N/2M)$. Several clever tape-sorting algorithms have been invented which eliminate virtually all of this copying by changing the way in which the small sorted blocks are merged together. The most prominent of these methods is called *polyphase merging.*

The basic idea behind polyphase merging is to distribute the sorted blocks produced by replacement selection somewhat unevenly among the available tape units (leaving one empty) and then to apply a "merge-until-empty" strategy, at which point one of the output tapes and the input tape switch roles.

For example, suppose that we have just three tapes, and we start out with the initial configuration of sorted blocks on the tapes shown at the top of figure:



(This comes from applying replacement selection to our example file with an internal memory that can only hold two records.) Tape 3 is initially empty, the output tape for the first merges. Now, after three two-way merges from tapes I and 2 to tape 3, the second tape becomes empty. Then, after two two-way merges from tapes I and 3 to tape 2, the first tape becomes empty. The sort is completed in two more steps. First, a two-way merge from tapes 2 and 3 to tape I leaves one file on tape 2, one file on tape 1. Then a two-way merge from tapes I and 2 to tape 3 leaves the entire sorted file on tape 3.

This merge-until-empty strategy can be extended to work for an arbitrary number of tapes. Table below shows how six tapes might be used to sort 497 initial runs:

Tape 1	*61*	0	*31*	*15*	7	*3*	1	0	1
Tape 2	**0**	*61*	*30*	*14*	*6*	*2*	0	1	0
Tape 3	*120*	*59*	*28*	*12*	*4*	0	*2*	1	0
Tape 4	*116*	*55*	*24*	*8*	0	*4*	*2*	1	0
Tape 5	*108*	*47*	*16*	0	*8*	*4*	*2*	1	0
Tape 6	*92*	*31*	0	*16*	*8*	*4*	*2*	1	0

If we start out as indicated in the first column of table, with Tape 2 being the output tape, Tape I having 61 initial runs, Tape 3 having 120 initial runs, etc. as indicated in the first column of table, then after running a five-way "merge until empty," we have Tape I empty, Tape 2 with 61 (long) runs, Tape 3 with 59 runs, etc., as shown in the second column of table. At this point, we can rewind Tape 2 and make it an input tape, and rewind Tape I and make it the output tape. Continuing in this way, we eventually get the whole sorted file onto Tape 1. The merge is broken up into many ***phases*** which don't involve all the data, but no direct copying is involved.

The main difficulty in implementing a polyphase merge is to determine how to distribute the initial runs. It is not difficult to see how to build the table above by working backwards:

take the largest number in each column, make it zero, and add it to each of the other numbers to get the previous column.

This corresponds to defining the highest-order merge for the previous column which could give the present column. This technique works for any number of tapes (at least three):

the numbers which arise are "generalized Fibonacci numbers" which have many interesting properties.

Of course, the number of initial runs may not be known in advance, and it probably won't be exactly a generalized Fibonacci number. Thus a number of "dummy" runs must be added to make the number of initial runs exactly what is needed for the table.

The analysis of polyphase merging is complicated and interesting, and yields surprising results. For example, it turns out that the very best method for distributing dummy runs among the tapes involves using extra phases and more dummy runs than would seem to be needed. The reason for this is that some runs are used in merges much more often than others.

Many other factors must be taken into consideration in implementing a most efficient tape-sorting method. A major factor which we have not considered at all is the time that it takes to rewind a tape. This subject has been studied extensively, and many fascinating methods have been defined. However, as mentioned above, the savings achievable over the simple multiway balanced merge are quite limited. Even polyphase merging is better than balanced merging only for small P, and then not substantially. For $P > 8$, balanced merging is likely to run faster than polyphase, and for smaller P the effect of polyphase is basically to save two tapes (a balanced merge with two extra tapes will run faster).

An Easier Way

Many modern computer systems provide a large *virtual memory* capability which should not be overlooked in implementing a method for sorting very large files. In a good virtual-memory system, the programmer can address a very large amount of data, leaving to the system the responsibility of making sure that the addressed data is transferred from external to internal storage when needed. This strategy relies on the fact that many programs have a relatively small "*locality of reference*":

each reference to memory is likely to be to an area of memory that is relatively close to other recently referenced areas.

This implies that transfers from external to internal storage are needed infrequently. An internal sorting method with a small locality of reference can work very well on a virtual-memory system. (For example, Quicksort has two "localities": most references are near one of the two partitioning pointers).

But a method such as radix sorting, which has no locality of reference whatsoever, would be disastrous on a virtual memory system, and even Quicksort could cause problems, depending on how well the available virtual memory system is implemented.

On the other hand, the strategy of using a simple internal sorting method for sorting disk files deserves serious consideration in a ood virtual-memory environment.