Table 2.9. Execution times of sort programs with 256 elements

	Ordered	Random	Inverse
StraightInsertion	0.22	50.74	103.80
BinaryInsertion	1.16	37.66	76.06
StraightSelection	58.18	58.34	73.46
BubbleSort	80.18	128.84	178.66
ShakerSort	0.16	104.44	187.36
ShellSort	0.80	7.08	12.34
HeapSort	2.32	2.22	2.12
QuickSort	0.72	1.22	0.76
NonRecQuickSort	0.72	1.32	0.80
StraightMerge	1.98	2.06	1.98

Table 2.10. Execution times of sort programs with 2048 elements

2.4. Sorting Sequences

2.4.1. Straight Merging

Unfortunately, the sorting algorithms presented in the preceding chapter are inapplicable, if the amount of data to be sorted does not fit into a computer's main store, but if it is, for instance, represented on a peripheral and sequential storage device such as a tape or a disk. In this case we describe the data as a (sequential) file whose characteristic is that at each moment one and only one component is directly accessible. This is a severe restriction compared to the possibilities offered by the array structure, and therefore different sorting techniques have to be used. The most important one is sorting by merging. Merging (or collating) means combining two (or more) ordered sequences into a single, ordered sequence by repeated selection among the currently accessible components. Merging is a much simpler operation than sorting, and it is used as an auxiliary operation in the more complex process of sequential sorting. One way of sorting on the basis of merging, called *straight merging*, is the following:

1. Split the sequence a into two halves, called b and c.
2. Merge b and c by combining single items into ordered pairs.
3. Call the merged sequence a, and repeat steps 1 and 2, this time merging ordered pairs into ordered quadruples.
4. Repeat the previous steps, merging quadruples into octets, and continue doing this, each time doubling the lengths of the merged subsequences, until the entire sequence is ordered.

As an example, consider the sequence

44 55 12 42 94 18 06 67

In step 1, the split results in the sequences

44 55 12 42
94 18 06 67

The merging of single components (which are ordered sequences of length 1), into ordered pairs yields

44 94 ' 18 55 ' 06 12 ' 42 67

Splitting again in the middle and merging ordered pairs yields

06 12 44 94 ' 18 42 55 67

A third split and merge operation finally produces the desired result

06 12 18 42 44 55 67 94

Each operation that treats the entire set of data once is called a *phase*, and the smallest subprocess that by repetition constitutes the sort process is called a pass or a stage. In the above example the sort took three passes, each pass consisting of a splitting phase and a merging phase. In order to perform the sort, three tapes are needed; the process is therefore called a three-tape merge.

Actually, the splitting phases do not contribute to the sort since they do in no way permute the items; in a sense they are unproductive, although they constitute half of all copying operations. They can be eliminated altogether by combining the split and the merge phase. Instead of merging into a single sequence, the output of the merge process is immediately redistributed onto two tapes, which constitute the sources of the subsequent pass. In contrast to the previous two-phase merge sort, this method is called a *single-phase merge* or a *balanced merge*. It is evidently superior because only half as many copying operations are necessary; the price for this advantage is a fourth tape.

We shall develop a merge program in detail and initially let the data be represented as an array which, however, is scanned in strictly sequential fashion. A later version of merge sort will then be based on the sequence structure, allowing a comparison of the two programs and demonstrating the strong dependence of the form of a program on the underlying representation of its data.

A single array may easily be used in place of two sequences, if it is regarded as double-ended. Instead of merging from two source files, we may pick items off the two ends of the array. Thus, the general form of the combined merge-split phase can be illustrated as shown in Fig. 2.12. The destination of the merged items is switched after each ordered pair in the first pass, after each ordered quadruple in the second pass, etc., thus evenly filling the two destination sequences, represented by the two ends of a single array. After each pass, the two arrays interchange their roles, the source becomes the new destination, and vice versa.

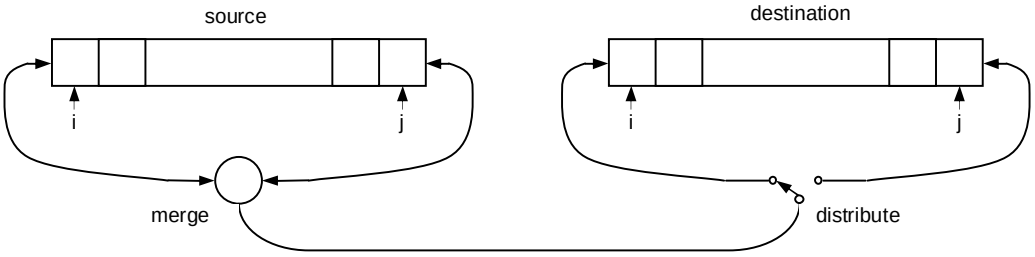

Fig. 2.12. Straight merge sort with two arrays

A further simplification of the program can be achieved by joining the two conceptually distinct arrays into a single array of doubled size. Thus, the data will be represented by

```
a: ARRAY 2*n OF item
```

and we let the indices i and j denote the two source items, whereas k and L designate the two destinations (see Fig. 2.12). The initial data are, of course, the items a1 ... an. Clearly, a Boolean variable up is needed to denote the direction of the data flow; up shall mean that in the current pass components a_0 ... a_{n-1} will be moved up to the variables a_n ... a_{2n}-1, whereas ~up will indicate that a_n ... a_{2n-1} will be transferred down into a_0 ... a_{n-1}. The value of up strictly alternates between consecutive passes. And, finally, a variable p is introduced to denote the length of the subsequences to be merged. Its value is initially 1, and it is doubled before each successive pass. To simplify matters somewhat, we shall assume that n is always a power of 2. Thus, the first version of the straight merge program assumes the following form:

```
PROCEDURE MergeSort;
  VAR i, j, k, L, p: INTEGER;  up: BOOLEAN;
BEGIN up := TRUE; p := 1;
  REPEAT initialize index variables;
    IF up THEN i := 0; j := n-1; k := n; L := 2*n-1
    ELSE k := 0; L := n-1; i := n; j := 2*n-1
    END ;
    merge p-tuples from i- and j-sources to k- and L-destinations;
    up := ~up; p := 2*p
  UNTIL p = n
END MergeSort
```

In the next development step we further refine the statements expressed in italics. Evidently, the merge pass involving n items is itself a sequence of merges of sequences, i.e. of p-tuples. Between every such partial

merge the destination is switched from the lower to the upper end of the destination array, or vice versa, to guarantee equal distribution onto both destinations. If the destination of the merged items is the lower end of the destination array, then the destination index is k, and k is incremented after each move of an item. If they are to be moved to the upper end of the destination array, the destination index is L, and it is decremented after each move. In order to simplify the actual merge statement, we choose the destination to be designated by k at all times, switching the values of the variables k and L after each p-tuple merge, and denote the increment to be used at all times by h, where h is either 1 or -1. These design discussions lead to the following refinement:

```
h := 1; m := n;  (*m = no. of items to be merged*)
REPEAT q := p; r := p; m := m - 2*p;
  merge q items from i-source with r items from j-source.
  destination index is k. increment k by h;
  h := -h; exchange k and L
UNTIL m = 0
```

In the further refinement step the actual merge statement is to be formulated. Here we have to keep in mind that the tail of the one subsequence which is left non-empty after the merge has to be appended to the output sequence by simple copying operations.

```
WHILE (q > 0) & (r > 0) DO
  IF a[i] < a[j] THEN
    move an item from i-source to k-destination; advance i and k;  q := q-1
  ELSE
    move an item from j-source to k-destination; advance j and k;  r := r-1
  END
END ;
copy tail of i-sequence; copy tail of j-sequence
```

After this further refinement of the tail copying operations, the program is laid out in complete detail. Before writing it out in full, we wish to eliminate the restriction that n be a power of 2. Which parts of the algorithm are affected by this relaxation of constraints? We easily convince ourselves that the best way to cope with the more general situation is to adhere to the old method as long as possible. In this example this means that we continue merging p-tuples until the remainders of the source sequences are of length less than p. The one and only part that is influenced are the statements that determine the values of q and r, the lengths of the sequences to be merged. The following four statements replace the three statements

```
q := p;  r := p;  m := m -2*p
```

and, as the reader should convince himself, they represent an effective implementation of the strategy specified above; note that m denotes the total number of items in the two source sequences that remain to be merged:

```
IF m >= p THEN q := p ELSE q := m END ;
m := m-q;
IF m >= p THEN r := p ELSE r := m END ;
m := m-r
```

In addition, in order to guarantee termination of the program, the condition p=n, which controls the outer repetition, must be changed to $p \geq n$. After these modifications, we may now proceed to describe the entire algorithm in terms of a procedure operating on the global array a with 2n elements.

```
PROCEDURE StraightMerge;
  VAR i, j, k, L, t: INTEGER;   (*index range of a is 0 .. 2*n-1 *)
    h, m, p, q, r: INTEGER; up: BOOLEAN;
BEGIN up := TRUE; p := 1;
  REPEAT h := 1; m := n;
    IF up THEN i := 0; j := n-1; k := n; L := 2*n-1
    ELSE k := 0; L := n-1; i := n; j := 2*n-1
    END ;
```

```
    REPEAT (*merge a run from i- and j-sources to k-destination*)
      IF m >= p THEN q := p ELSE q := m END ;
      m := m-q;
      IF m >= p THEN r := p ELSE r := m END ;
      m := m-r;
      WHILE (q > 0) & (r > 0) DO
        IF a[i] < a[j] THEN
          a[k] := a[i]; k := k+h; i := i+1; q := q-1
        ELSE
          a[k] := a[j]; k := k+h; j := j-1; r := r-1
        END
      END ;
      WHILE r > 0 DO
        a[k] := a[j]; k := k+h; j := j-1; r := r-1
      END ;
      WHILE q > 0 DO
        a[k] := a[i]; k := k+h; i := i+1; q := q-1
      END ;
      h := -h; t := k; k := L; L := t
    UNTIL m = 0;
    up := ~up; p := 2*p
  UNTIL p >= n;
  IF ~up THEN
    FOR i := 1 TO n DO a[i] := a[i+n] END
  END
END StraightMerge
```

Analysis of Mergesort. Since each pass doubles p, and since the sort is terminated as soon as p > n, it involves ilog nj passes. Each pass, by definition, copies the entire set of n items exactly once. As a consequence, the total number of moves is exactly

$$M = n \times \log(n)$$

The number C of key comparisons is even less than M since no comparisons are involved in the tail copying operations. However, since the mergesort technique is usually applied in connection with the use of peripheral storage devices, the computational effort involved in the move operations dominates the effort of comparisons often by several orders of magnitude. The detailed analysis of the number of comparisons is therefore of little practical interest.

The merge sort algorithm apparently compares well with even the advanced sorting techniques discussed in the previous chapter. However, the administrative overhead for the manipulation of indices is relatively high, and the decisive disadvantage is the need for storage of 2n items. This is the reason sorting by merging is rarely used on arrays, i.e., on data located in main store. Figures comparing the real time behavior of this Mergesort algorithm appear in the last line of Table 2.9. They compare favorably with Heapsort but unfavorably with Quicksort.

2.4.2. Natural Merging

In straight merging no advantage is gained when the data are initially already partially sorted. The length of all merged subsequences in the k th pass is less than or equal to 2k, independent of whether longer subsequences are already ordered and could as well be merged. In fact, any two ordered subsequences of lengths m and n might be merged directly into a single sequence of m+n items. A mergesort that at any time merges the two longest possible subsequences is called a *natural merge* sort.

An ordered subsequence is often called a string. However, since the word string is even more frequently used to describe sequences of characters, we will follow Knuth in our terminology and use the word *run* instead of string when referring to ordered subsequences. We call a subsequence $a_i \ldots a_j$ such that

$$(a_{i-1} > a_i) \;\&\; (\mathbf{A}k : i \le k < j : a_k \le a_{k+1}) \;\&\; (a_j > a_{j+1})$$

a *maximal run* or, for short, a run. A natural merge sort, therefore, merges (maximal) runs instead of sequences of fixed, predetermined length. Runs have the property that if two sequences of n runs are merged, a single sequence of exactly n runs emerges. Therefore, the total number of runs is halved in each pass, and the number of required moves of items is in the worst case n*log(n), but in the average case it is even less. The expected number of comparisons, however, is much larger because in addition to the comparisons necessary for the selection of items, further comparisons are needed between consecutive items of each file in order to determine the end of each run.

Our next programming exercise develops a natural merge algorithm in the same stepwise fashion that was used to explain the straight merging algorithm. It employs the sequence structure (represented by files, see Sect. 1.8) instead of the array, and it represents an unbalanced, two-phase, three-tape merge sort. We assume that the file variable *c* represents the initial sequence of items. (Naturally, in actual data processing application, the initial data are first copied from the original source to c for reasons of safety.) *a* and *b* are two auxiliary file variables. Each pass consists of a distribution phase that distributes runs equally from c to a and b, and a merge phase that merges runs from a and b to c. This process is illustrated in Fig. 2.13.

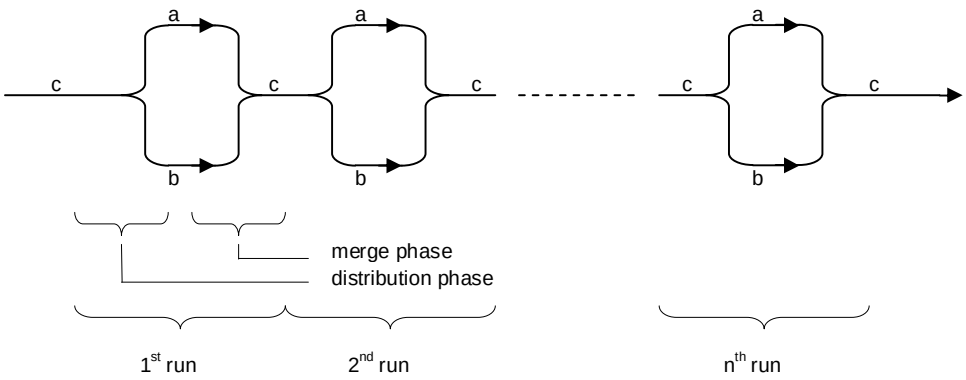

Fig. 2.13. Sort phases and passes

17	31'	05	59'	13	41	43	67'	11	23	29	47'	03	07	71'	02	19	57'	37	61
05	17	31	59'	11	13	23	29	41	43	47	67'	02	03	07	19	57	71'	37	61
05	11	13	17	23	29	31	41	43	47	59	67'	02	03	07	19	37	57	61	71
02	03	05	07	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

Table 2.11. Example of a Natural Mergesort.

As an example, Table 2.11 shows the file c in its original state (line1) and after each pass (lines 2-4) in a natural merge sort involving 20 numbers. Note that only three passes are needed. The sort terminates as soon as the number of runs on c is 1. (We assume that there exists at least one non-empty run on the initial sequence). We therefore let a variable L be used for counting the number of runs merged onto c. By making use of the type *Rider* defined in Sect. 1.8.1, the program can be formulated as follows:

```
VAR L: INTEGER;
  r0, r1, r2: Files.Rider;  (*see 1.8.1*)

REPEAT Files.Set(r0, a, 0); Files.Set(r1, b, 0); Files.Set(r2, c, 0);
  distribute(r2, r0, r1);  (*c to a and b*)
  Files.Set(r0, a, 0); Files.Set(r1, b, 0); Files.Set(r2, c, 0);
  L := 0; merge(r0, r1, r2)  (*a and b into c*)
UNTIL L = 1
```

The two phases clearly emerge as two distinct statements. They are now to be refined, i.e., expressed in more detail. The refined descriptions of *distribute* (from rider r2 to riders r0 and r1) and *merge* (from riders r0 and r1 to rider r2) follow:

```
REPEAT copyrun(r2, r0);
  IF ~r2.eof THEN copyrun(r2, r1) END
UNTIL r2.eof

REPEAT mergerun(r0, r1, r2); INC(L)
UNTIL r1.eof;
IF ~r0.eof THEN copyrun(r0, r2); INC(L) END
```

This method of distribution supposedly results in either equal numbers of runs in both a and b, or in sequence a containing one run more than b. Since corresponding pairs of runs are merged, a leftover run may still be on file a, which simply has to be copied. The statements merge and distribute are formulated in terms of a refined statement *mergerun* and a subordinate procedure *copyrun* with obvious tasks. When attempting to do so, one runs into a serious difficulty: In order to determine the end of a run, two consecutive keys must be compared. However, files are such that only a single element is immediately accessible. We evidently cannot avoid to look ahead, i.e to associate a buffer with every sequence. The buffer is to contain the first element of the file still to be read and constitutes something like a window sliding over the file.

Instead of programming this mechanism explicitly into our program, we prefer to define yet another level of abstraction. It is represented by a new module *Runs*. It can be regarded as an extension of module *Files* of Sect. 1.8, introducing a new type *Rider*, which we may consider as an extension of type *Files.Rider*. This new type will not only accept all operations available on Riders and indicate the end of a file, but also indicate the end of a run and the first element of the remaining part of the file. The new type as well as its operators are presented by the following definition.

```
DEFINITION Runs;
  IMPORT Files, Texts;
  TYPE Rider = RECORD (Files.Rider) first: INTEGER; eor: BOOLEAN END ;

  PROCEDURE OpenRandomSeq(f: Files.File; length, seed: INTEGER);
  PROCEDURE Set (VAR r: Rider; VAR f: Files.File);
  PROCEDURE copy(VAR source, destination: Rider);
  PROCEDURE ListSeq(VAR W: Texts.Writer; f: Files.File);
END Runs.
```

A few additional explanations for the choice of the procedures are necessary. As we shall see, the sorting algorithms discussed here and later are based on copying elements from one file to another. A procedure *copy* therefore takes the place of separate read and write operations.

For convenience of testing the following examples, we also introduce a procedure *ListSeq*, converting a file of integers into a text. Also for convenience an additional procedure is included: *OpenRandomSeq* initializes a file with numbers in random order. These two procedures will serve to test the algorithms to be discussed below. The values of the fields *eof* and *eor* are defined as results of *copy* in analogy to *eof* having been defined as result of a read operation.

```
MODULE Runs;
    IMPORT Files, Texts;
    TYPE Rider* = RECORD (Files.Rider) first: INTEGER; eor: BOOLEAN END ;

    PROCEDURE OpenRandomSeq*( f: Files.File; length, seed: INTEGER);
        VAR i: INTEGER; w: Files.Rider;
    BEGIN Files.Set(w, f, 0);
        FOR i := 0 TO length-1 DO
          Files.WriteInt(w, seed); seed := (31*seed) MOD 997 + 5
        END ;
        Close(f)
    END OpenRandomSeq;

    PROCEDURE Set*(VAR r: Rider; f: Files.File);
    BEGIN Files.Set(r, f, 0); Files.Read (r, r.first); r.eor := r.eof
    END Set;
```

```
    PROCEDURE copy*(VAR src, dest: Rider);
    BEGIN dest.first := src.first;
        Files.Write(dest, dest.first); Files.Read(src, src.first);
        src.eor := src.eof OR (src.first < dest.first)
    END copy;

    PROCEDURE ListSeq*(VAR W: Texts; f: Files.File;);
        VAR x, y, k, n: INTEGER; r: Files.Rider;
    BEGIN k := 0; n := 0; Files.Set(r, f, 0); Files.ReadInt(r, x);
        WHILE ~r.eof DO
          Texts.WriteInt(W, x, 6); INC(k); Files.Read(r, y);
          IF y < x THEN (*run ends*) Texts.Write(W, "|"); INC(n) END ;
          x := y
        END ;
        Texts.Write(W, "$"); Texts.WriteInt(W, k, 5); Texts.WriteInt(W, n, 5);
        Texts.WriteLn(W)
    END ListSeq;

END Runs.
```

We now return to the process of successive refinement of the process of natural merging. Procedure *copyrun* and the statement *merge* are now conveniently expressible as shown below. Note that we refer to the sequences (files) indirectly via the riders attached to them. In passing, we also note that the rider's field *first* represents the *next* key on a sequence being read, and the *last* key of a sequence being written.

```
PROCEDURE copyrun(VAR x, y: Runs.Rider);
BEGIN (*copy from x to y*)
  REPEAT Runs.copy(x, y) UNTIL x.eor
END copyrun

(*merge from r0 and r1 to r2*)
REPEAT
  IF r0.first < r1.first THEN
    Runs.copy(r0, r2);
    IF r0.eor THEN copyrun(r1, r2) END
  ELSE Runs.copy(r1, r2);
    IF r1.eor THEN copyrun(r0, r2) END
  END
UNTIL r0.eor OR r1.eor
```

The comparison and selection process of keys in merging a run terminates as soon as one of the two runs is exhausted. After this, the other run (which is not exhausted yet) has to be transferred to the resulting run by merely copying its tail. This is done by a call of procedure *copyrun*.

This should supposedly terminate the development of the natural merging sort procedure. Regrettably, the program is incorrect, as the very careful reader may have noticed. The program is incorrect in the sense that it does not sort properly in some cases. Consider, for example, the following sequence of input data:

03 02 05 11 07 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67

By distributing consecutive runs alternately to a and b, we obtain

```
a =  03 ' 07 13 19 ' 29 37 43 ' 57 61 71'
b =  02 05 11 ' 17 23 31 ' 41 47 59 ' 67
```

These sequences are readily merged into a single run, whereafter the sort terminates successfully. The example, although it does not lead to an erroneous behaviour of the program, makes us aware that mere distribution of runs to serveral files may result in a number of output runs that is less than the number of input runs. This is because the first item of the i+2nd run may be larger than the last item of the i-th run, thereby causing the two runs to merge automatically into a single run.

Although procedure *distribute* supposedly outputs runs in equal numbers to the two files, the important consequence is that the actual number of resulting runs on *a* and *b* may differ significantly. Our merge procedure, however, only merges pairs of runs and terminates as soon as *b* is read, thereby losing the tail of one of the sequences. Consider the following input data that are sorted (and truncated) in two subsequent passes:

```
17 19 13 57 23 29 11 59 31 37 07 61 41 43 05 67 47 71 02 03
13 17 19 23 29 31 37 41 43 47 57 71 11 59
11 13 17 19 23 29 31 37 41 43 47 57 59 71
```

Table 2.12 Incorrect Result of Mergesort Program.

The example of this programming mistake is typical for many programming situations. The mistake is caused by an oversight of one of the possible consequences of a presumably simple operation. It is also typical in the sense that serval ways of correcting the mistake are open and that one of them has to be chosen. Often there exist two possibilities that differ in a very important, fundamental way:

1. We recognize that the operation of distribution is incorrectly programmed and does not satisfy the requirement that the number of runs differ by at most 1. We stick to the original scheme of operation and correct the faulty procedure accordingly.
2. We recognize that the correction of the faulty part involves far-reaching modifications, and we try to find ways in which other parts of the algorithm may be changed to accommodate the currently incorrect part.

In general, the first path seems to be the safer, cleaner one, the more honest way, providing a fair degree of immunity from later consequences of overlooked, intricate side effects. It is, therefore, the way toward a solution that is generally recommended.

It is to be pointed out, however, that the second possibility should sometimes not be entirely ignored. It is for this reason that we further elaborate on this example and illustrate a fix by modification of the merge procedure rather than the distribution procedure, which is primarily at fault.

This implies that we leave the distribution scheme untouched and renounce the condition that runs be equally distributed. This may result in a less than optimal performance. However, the worst-case performance remains unchanged, and moreover, the case of highly unequal distribution is statistically very unlikely. Efficiency considerations are therefore no serious argument against this solution.

If the condition of equal distribution of runs no longer exists, then the merge procedure has to be changed so that, after reaching the end of one file, the entire tail of the remaining file is copied instead of at most one run. This change is straightforward and is very simple in comparison with any change in the distribution scheme. (The reader is urged to convince himself of the truth of this claim). The revised version of the merge algorithm is shown below in the form of a function procedure:

```
PROCEDURE NaturalMerge(src: Files.File): Files.File;
  VAR L: INTEGER;  (*no. of runs merged*)
        f0, f1, f2: Files.File;
        r0, r1, r2: Runs.Rider;

  PROCEDURE copyrun(VAR x, y: Runs.Rider);
  BEGIN (*from x to y*)
        REPEAT Runs.copy(x, y) UNTIL x.eor
  END copyrun;

BEGIN Runs.Set(r2, src);
  REPEAT f0 := Files.New("test0"); Files.Set(r0, f0, 0);
    f1 := Files.New("test1"); Files.Set (r1, f1, 0);
    (*distribute from r2 to r0 and r1*)
    REPEAT copyrun(r2, r0);
      IF ~r2.eof THEN copyrun(r2, r1) END
    UNTIL r2.eof;
    Runs.Set(r0, f0); Runs.Set(r1, f1);
    f2 := Files.New(""); Files.Set(r2, f2, 0); L := 0;
```

```
    (*merge from r0 and r1 to r2*)
    REPEAT
      REPEAT
        IF r0.first < r1.first THEN
          Runs.copy(r0, r2);
          IF r0.eor THEN copyrun(r1, r2) END
        ELSE Runs.copy(r1, r2);
          IF r1.eor THEN copyrun(r0, r2) END
        END
      UNTIL r0.eor OR r1.eor;
      INC(L)
    UNTIL r0.eof OR r1.eof;
    WHILE ~r0.eof DO copyrun(r0, r2); INC(L) END ;
    WHILE ~r1.eof DO copyrun(r1, r2); INC(L) END ;
    Runs.Set(r2, f2)
  UNTIL L = 1;
  RETURN f2
END NaturalMerge;
```

2.4.3. Balanced Multiway Merging

The effort involved in a sequential sort is proportional to the number of required passes since, by definition, every pass involves the copying of the entire set of data. One way to reduce this number is to distribute runs onto more than two files. Merging *r* runs that are equally distributed on *N* files results in a sequence of r/N runs. A second pass reduces their number to r/N^2, a third pass to r/N^3, and after *k* passes there are r/N^k runs left. The total number of passes required to sort *n* items by N-way merging is therefore $k = \log_N(n)$. Since each pass requires *n* copy operations, the total number of copy operations is in the worst case $M = n \times \log_N(n)$

As the next programming exercise, we will develop a sort program based on multiway merging. In order to further contrast the program from the previous natural two-phase merging procedure, we shall formulate the multiway merge as a single phase, balanced mergesort. This implies that in each pass there are an equal number of input and output files onto which consecutive runs are alternately distributed. Using 2*N* files, the algorithm will therefore be based on N-way merging. Following the previously adopted strategy, we will not bother to detect the automatic merging of two consecutive runs distributed onto the same file. Consequently, we are forced to design the merge program whithout assuming strictly equal numbers of runs on the input files.

In this program we encounter for the first time a natural application of a data structure consisting of arrays of files. As a matter of fact, it is surprising how strongly the following program differs from the previous one because of the change from two-way to multiway merging. The change is primarily a result of the circumstance that the merge process can no longer simply be terminated after one of the input runs is exhausted. Instead, a list of inputs that are still active, i.e., not yet exhausted, must be kept. Another complication stems from the need to switch the groups of input and output files after each pass. Here the indirection of access to files via riders comes in handy. In each pass, data may be copied from the same riders r to the same riders w. At the end of each pass we merely need to reset the input and output files to different riders.

Obviously, file numbers are used to index the array of files. Let us then assume that the initial file is the parameter *src*, and that for the sorting process 2*N* files are available:

```
f, g: ARRAY N OF Files.File;
r, w: ARRAY N OF Runs.Rider
```

The algorithm can now be sketched as follows:

```
PROCEDURE BalancedMerge(src: Files.File): Files.File;
  VAR i, j: INTEGER;
    L: INTEGER; (*no. of runs distributed*)
```

```
    R: Runs.Rider;
BEGIN Runs.Set(R, src);  (*distribute initial runs from R to w[0] ... w[N-1]*)
  j := 0; L := 0;
  position riders w on files g;
  REPEAT
    copy one run from R to w[j];
    INC(j); INC(L);
    IF j = N THEN j := 0 END
  UNTIL R.eof;

  REPEAT (*merge from riders r to riders w*)
    switch files g to riders r;
    L := 0; j := 0;  (*j = index of output file*)
    REPEAT INC(L);
      merge one run from inputs to w[j];
      IF j < N THEN INC(j) ELSE j := 0 END
    UNTIL all inputs exhausted;
  UNTIL L = 1
  (*sorted file is with w[0]*)
END BalancedMerge.
```

Having associated a rider *R* with the source file, we now refine the statement for the initial distribution of runs. Using the definition of *copy*, we replace *copy one run from R to w[j]* by:

```
REPEAT Runs.copy(R, w[j]) UNTIL R.eor
```

Copying a run terminates when either the first item of the next run is encountered or when the end of the entire input file is reached.

In the actual sort algorithm, the following statements remain to be specified in more detail:

1. Position riders w on files g
2. Merge one run from inputs to w_j
3. Switch files g to riders r
4. All inputs exhausted

First, we must accurately identify the current input sequences. Notably, the number of *active* inputs may be less than N. Obviously, there can be at most as many sources as there are runs; the sort terminates as soon as there is one single sequence left. This leaves open the possibility that at the initiation of the last sort pass there are fewer than *N* runs. We therefore introduce a variable, say *k1*, to denote the actual number of inputs used. We incorporate the initialization of *k1* in the statement *switch files* as follows:

```
IF L < N THEN k1 := L ELSE k1 := N END ;
FOR i := 0 TO k1-1 DO Runs.Set(r[i], g[i]) END
```

Naturally, statement (2) is to decrement *k1* whenever an input source ceases. Hence, predicate (4) may easily be expressed by the relation k1 = 0. Statement (2), however, is more difficult to refine; it consists of the repeated selection of the least key among the available sources and its subsequent transport to the destination, i.e., the current output sequence. The process is further complicated by the necessity of determining the end of each run. The end of a run may be reached because (1) the subsequent key is less than the current key or (2) the end of the source is reached. In the latter case the source is eliminated by decrementing *k1*; in the former case the run is closed by excluding the sequence from further selection of items, but only until the creation of the current output run is completed. This makes it obvious that a second variable, say *k2*, is needed to denote the number of sources actually available for the selection of the next item. This value is initially set equal to *k1* and is decremented whenever a run teminates because of condition (1).

Unfortunately, the introduction of *k2* is not sufficient. We need to know not only the number of files, but also which files are still in actual use. An obvious solution is to use an array with Boolean components indicating the availability of the files. We choose, however, a different method that leads to a more efficient selection

procedure which, after all, is the most frequently repeated part of the entire algorithm. Instead of using a Boolean array, a file index map, say *t*, is introduced. This map is used so that t_0 ... t_{k2-1} are the indices of the available sequences. Thus statement (2) can be formulated as follows:

```
k2 := k1;
REPEAT select the minimal key, let t[m] be the sequence number on which it occurs;
  Runs.copy(r[t[m]], w[j]);
  IF r[t[m]].eof THEN eliminate sequence
  ELSIF r[t[m]].eor THEN close run
  END
UNTIL k2 = 0
```

Since the number of sequences will be fairly small for any practical purpose, the selection algorithm to be specified in further detail in the next refinement step may as well be a straightforward linear search. The statement *eliminate sequence* implies a decrease of k1 as well as k2 and also a reassignment of indices in the map *t*. The statement *close run* merely decrements k2 and rearranges components of *t* accordingly. The details are shown in the following procedure, being the last refinement. The statement *switch sequences* is elaborated according to explanations given earlier.

```
PROCEDURE BalancedMerge(src: Files.File): Files.File;
  VAR i, j, m, tx: INTEGER;
    L, k1, k2: INTEGER;
    min, x: INTEGER;
    t: ARRAY N OF INTEGER;  (*index map*)
    R: Runs.Rider;  (*source*)
    f, g:  ARRAY N OF Files.File;
    r, w: ARRAY N OF Runs.Rider;

BEGIN Runs.Set(R, src);
  FOR i := 0 TO N-1 DO g[i] := Files.New(""); Files.Set(w[i], g[i], 0) END ;
  (*distribute initial runs from src to g[0] ... g[N-1]*)
  j := 0; L := 0;
  REPEAT
    REPEAT Runs.copy(R, w[j]) UNTIL R.eor;
    INC(L); INC(j);
    IF j = N THEN j := 0 END
  UNTIL R.eof;

  FOR i := 0 TO N-1 DO t[i] := i END ;
  REPEAT
    IF L < N THEN k1 := L ELSE k1 := N END ;
    FOR i := 0 TO k1-1 DO Runs.Set(r[i], g[i]) END ;  (*set input riders*)
    FOR i := 0 TO k1-1 DO g[i] := Files.New(""); Files.Set(w[i], g[i], 0) END ;  (*set output riders*)
    (*merge from r[0] ... r[N-1] to w[0] ... w[N-1]*)
    FOR i := 0 TO N-1 DO t[i] := i END ;

    L := 0;     (*nof runs merged*)
    j := 0;
    REPEAT (*merge one run from inputs to w[j]*)
      INC(L); k2 := k1;
      REPEAT (*select the minimal key*)
        m := 0; min := r[t[0]].first; i := 1;
        WHILE i < k2 DO
          x := r[t[i]].first;
          IF x < min THEN min := x; m := i END ;
          INC(i)
        END ;
        Runs.copy(r[t[m]], w[j]);
```

```
        IF r[t[m]].eof THEN (*eliminate this sequence*)
          DEC(k1); DEC(k2); t[m] := t[k2]; t[k2] := t[k1]
        ELSIF r[t[m]].eor THEN (*close run*)
          DEC(k2); tx := t[m]; t[m] := t[k2]; t[k2] := tx
        END
      UNTIL k2 = 0;
      INC(j);
      IF j = N THEN j := 0 END
    UNTIL k1 = 0
  UNTIL L = 1;
  RETURN Files.Base(w[t[0]])
END BalancedMerge
```

2.4.4. Polyphase Sort

We have now discussed the necessary techniques and have acquired the proper background to investigate and program yet another sorting algorithm whose performance is superior to the balanced sort. We have seen that balanced merging eliminates the pure copying operations necessary when the distribution and the merging operations are united into a single phase. The question arises whether or not the given sequences could be processed even more efficiently. This is indeed the case; the key to this next improvement lies in abandoning the rigid notion of strict passes, i.e., to use the sequences in a more sophisticated way than by always having N/2 sources and as many destinations and exchanging sources and destinations at the end of each distinct pass. Instead, the notion of a pass becomes diffuse. The method was invented by R.L. Gilstad [2-3] and called *Polyphase Sort.*

It is first illustrated by an example using three sequences. At any time, items are merged from two sources into a third sequence variable. Whenever one of the source sequences is exhausted, it immediately becomes the destination of the merge operations of data from the non-exhausted source and the previous destination sequence.

As we know that *n* runs on each input are transformed into *n* runs on the output, we need to list only the number of runs present on each sequence (instead of specifying actual keys). In Fig. 2.14 we assume that initially the two input sequences *f1* and *f2* contain 13 and 8 runs, respectively. Thus, in the first pass 8 runs are merged from *f1* and *f2* to *f3*, in the second pass the remaining 5 runs are merged from *f3* and *f1* onto *f2*, etc. In the end, *f1* is the sorted sequence.

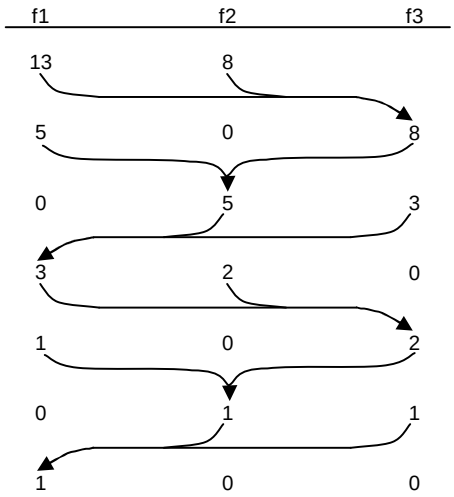

Fig. 2.14. Polyphase mergesort of 21 runs with 3 sequences

A second example shows the Polyphase method with 6 sequences. Let there initially be 16 runs on *f1*, 15 on *f2*, 14 on *f3*, 12 on *f4*, and 8 on *f5*. In the first partial pass, 8 runs are merged onto *f6*; In the end, *f2* contains the sorted set of items (see Fig. 2.15).

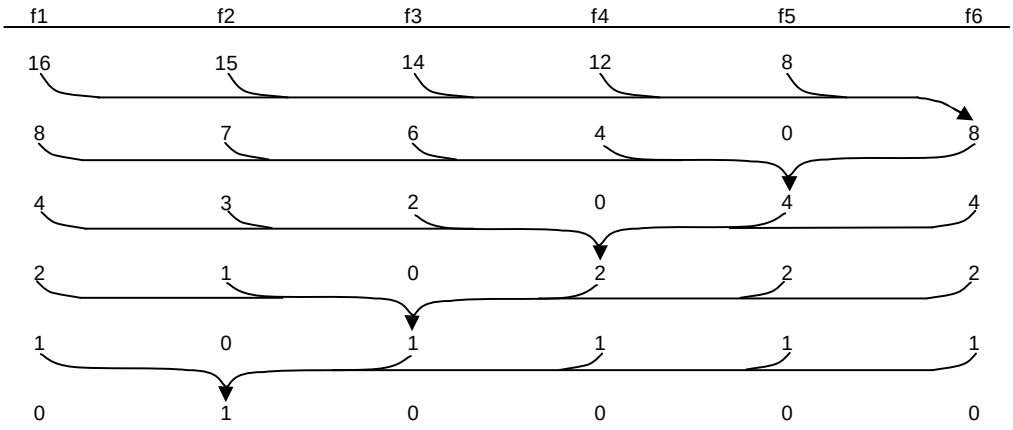

Fig. 2.15. Polyphase mergesort of 65 runs with 6 sequences

Polyphase is more efficient than balanced merge because, given *N* sequences, it always operates with an *N-1*-way merge instead of an *N/2*-way merge. As the number of required passes is approximately $\log_N n$, *n* being the number of items to be sorted and *N* being the degree of the merge operations, Polyphase promises a significant improvement over balanced merging.

Of course, the distribution of initial runs was carefully chosen in the above examples. In order to find out which initial distributions of runs lead to a proper functioning, we work backward, starting with the final distribution (last line in Fig. 2.15). Rewriting the tables of the two examples and rotating each row by one position with respect to the prior row yields Tables 2.13 and 2.14 for six passes and for three and six sequences, respectively.

L	$a_1(L)$	$a_2(L)$	Sum $a_i(L)$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Table 2.13 Perfect distribution of runs on two sequences.

L	$a_1(L)$	$a_2(L)$	$a_3(L)$	$a_4(L)$	$a_5(L)$	Sum $a_i(L)$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Table 2.14 Perfect distribution of runs on five sequences.

From Table 2.13 we can deduce for L > 0 the relations

$$a_2(L+1) = a_1(L)$$
$$a_1(L+1) = a_1(L) + a_2(L)$$

and $a_1(0) = 1$, $a_2(0) = 0$. Defining $f_{i+1} = a_1(i)$, we obtain for $i > 0$

$$f_{i+1} = f_i + f_{i-1},\ f_1 = 1,\ f_0 = 0$$

These are the recursive rules (or recurrence relations) defining the *Fibonacci numbers*:

$$f = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ ...$$

Each Fibonacci number is the sum of its two predecessors. As a consequence, the numbers of initial runs on the two input sequences must be two consecutive Fibonacci numbers in order to make Polyphase work properly with three sequences.

How about the second example (Table 2.14) with six sequences? The formation rules are easily derived as

$$\begin{aligned}
a_5(L+1) &= a_1(L) \\
a_4(L+1) &= a_1(L) + a_5(L) = a_1(L) + a_1(L-1) \\
a_3(L+1) &= a_1(L) + a_4(L) = a_1(L) + a_1(L-1) + a_1(L-2) \\
a_2(L+1) &= a_1(L) + a_3(L) = a_1(L) + a_1(L-1) + a_1(L-2) + a_1(L-3) \\
a_1(L+1) &= a_1(L) + a_2(L) = a_1(L) + a_1(L-1) + a_1(L-2) + a_1(L-3) + a_1(L-4)
\end{aligned}$$

Substituting f_i for $a_1(i)$ yields

$$\begin{aligned}
f_{i+1} &= f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4} \quad \text{for } i > 4 \\
f_4 &= 1 \\
f_i &= 0 \quad \text{for } i < 4
\end{aligned}$$

These numbers are the *Fibonacci numbers of order 4*. In general, the Fibonacci numbers of order p are defined as follows:

$$\begin{aligned}
f_{i+1}(p) &= f_i(p) + f_{i-1}(p) + ... + f_{i-p}(p) \quad \text{for } i > p \\
f_p(p) &= 1 \\
f_i(p) &= 0 \quad \text{for } 0 < i < p
\end{aligned}$$

Note that the ordinary Fibonacci numbers are those of order 1.

We have now seen that the initial numbers of runs for a perfect Polyphase Sort with *N* sequences are the sums of any *N*-1, *N*-2, ... , 1 (see Table 2.15) consecutive Fibonacci numbers of order *N*-2. This apparently implies that this method is only applicable to inputs whose number of runs is the sum of *N*-1 such Fibonacci sums. The important question thus arises: What is to be done when the number of initial runs is not such an ideal sum? The answer is simple (and typical for such situations): we simulate the existence of hypothetical empty runs, such that the sum of real and hypothetical runs is a perfect sum. The empty runs are called *dummy runs*.

But this is not really a satisfactory answer because it immediately raises the further and more difficult question: How do we recognize dummy runs during merging? Before answering this question we must first investigate the prior problem of initial run distribution and decide upon a rule for the distribution of actual and dummy runs onto the *N*-1 tapes.

1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001

Table 2.15 Numbers of runs allowing for perfect distribution.

In order to find an appropriate rule for distribution, however, we must know how actual and dummy runs are merged. Clearly, the selection of a dummy run from sequence *i* means precisely that sequence *i* is ignored during this merge. resulting in a merge from fewer than *N*-1 sources. Merging of a dummy run from all *N*-1 sources implies no actual merge operation, but instead the recording of the resulting dummy run on the output sequence. From this we conclude that dummy runs should be distributed to the *n*-1 sequences as uniformly as possible, since we are interested in active merges from as many sources as possible.

Let us forget dummy runs for a moment and consider the problem of distributing an unknown number of runs onto *N*-1 sequences. It is plain that the Fibonacci numbers of order *N*-2 specifying the desired numbers of runs on each source can be generated while the distribution progresses. Assuming, for example, *N* = 6 and referring to Table 2.14, we start by distributing runs as indicated by the row with index L = 1 (1, 1, 1, 1, 1); if there are more runs available, we proceed to the second row (2, 2, 2, 2, 1); if the source is still not exhausted, the distribution proceeds according to the third row (4, 4, 4, 3, 2), and so on. We shall call the row index *level*. Evidently, the larger the number of runs, the higher is the level of Fibonacci numbers which, incidentally, is equal to the number of merge passes or switchings necessary for the subsequent sort. The distribution algorithm can now be formulated in a first version as follows:

1. Let the distribution goal be the Fibonacci numbers of order *N*-2, level 1.
2. Distribute according to the set goal.
3. If the goal is reached, compute the next level of Fibonacci numbers; the difference between them and those on the former level constitutes the new distribution goal. Return to step 2. If the goal cannot be reached because the source is exhausted, terminate the distribution process.

The rules for calculating the next level of Fibonacci numbers are contained in their definition. We can thus concentrate our attention on step 2, where, with a given goal, the subsequent runs are to be distributed one after the other onto the *N*-1 output sequences. It is here where the dummy runs have to reappear in our considerations.

Let us assume that when raising the level, we record the next goal by the differences d_i for i = 1 ... N-1, where d_i denotes the number of runs to be put onto sequence *i* in this step. We can now assume that we immediately put d_i dummy runs onto sequence *i* and then regard the subsequent distribution as the replacement of dummy runs by actual runs, each time recording a replacement by subtracting 1 from the count d_i. Thus, the d_i indicates the number of dummy runs on sequence *i* when the source becomes empty.

It is not known which algorithm yields the optimal distribution, but the following has proved to be a very good method. It is called *horizontal distribution* (cf. Knuth, Vol 3. p. 270), a term that can be understood by imagining the runs as being piled up in the form of silos, as shown in Fig. 2.16 for *N* = 6, level 5 (cf. Table 2.14). In order to reach an equal distribution of remaining dummy runs as quickly as possible, their replacement by actual runs reduces the size of the piles by picking off dummy runs on horizontal levels proceeding from left to right. In this way, the runs are distributed onto the sequences as indicated by their numbers as shown in Fig. 2.16.

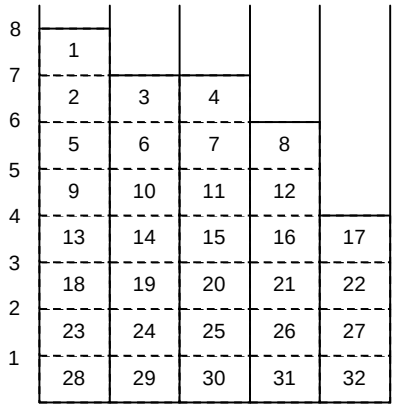

Fig. 2.16. Horizontal distribution of runs

We are now in a position to describe the algorithm in the form of a procedure called *select*, which is activated each time a run has been copied and a new source is selected for the next run. We assume the existence of a variable *j* denoting the index of the current destination sequence. a_i and d_i denote the ideal and dummy distribution numbers for sequence *i*.

```
j, level:    INTEGER;
a, d:   ARRAY N OF INTEGER;
```

These variables are initialized with the following values:

$a_i = 1$, $d_i = 1$ for i = 0 ... N-2
$a_{N-1} = 0$, $d_{N-1} = 0$ dummy
j = 0, level = 0

Note that *select* is to compute the next row of Table 2.14, i.e., the values $a_1(L)$ ... $a_{N-1}(L)$ each time that the level is increased. The next goal, i.e., the differences $d_i = a_i(L) - a_i(L-1)$ are also computed at that time. The indicated algorithm relies on the fact that the resulting d_i decrease with increasing index (descending stair in Fig. 2.16). Note that the exception is the transition from level 0 to level 1; this algorithm must therefore be used starting at level 1. *Select* ends by decrementing d_j by 1; this operation stands for the replacement of a dummy run on sequence *j* by an actual run.

```
PROCEDURE select;
  VAR i, z: INTEGER;
BEGIN
  IF d[j] < d[j+1] THEN INC(j)
  ELSE
    IF d[j] = 0 THEN
      INC(level); z := a[0];
      FOR i := 0 TO N-2 DO
        d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
      END
    END ;
    j := 0
  END ;
  DEC(d[j])
END select
```

Assuming the availability of a routine to copy a run from the source *src* woth rider R onto f_j with rider r_j, we can formulate the initial distribution phase as follows (assuming that the source contains at least one run):

```
REPEAT select; copyrun
UNTIL R.eof
```

Here, however, we must pause for a moment to recall the effect encountered in distributing runs in the previously discussed natural merge algorithm: The fact that two runs consecutively arriving at the same destination may merge into a single run, causes the assumed numbers of runs to be incorrect. By devising the sort algorithm such that its correctness does not depend on the number of runs, this side effect can safely be ignored. In the Polyphase Sort, however, we are particularly concerned about keeping track of the exact number of runs on each file. Consequently, we cannot afford to overlook the effect of such a coincidental merge. An additional complication of the distribution algorithm therefore cannot be avoided. It becomes necessary to retain the keys of the last item of the last run on each sequence. Fortunately, our implementation of *Runs* does exactly this. In the case of output sequences, *f.first* represents the item last written. A next attempt to describe the distribution algorithm could therefore be

```
REPEAT select;
  IF f[j].first <= f0.first THEN continue old run END ;
  copyrun
UNTIL R.eof
```

The obvious mistake here lies in forgetting that *f[j].first* has only obtained a value after copying the first run. A correct solution must therefore first distribute one run onto each of the *N*-1 destination sequences without inspection of *first*. The remaining runs are distributed as follows:

```
WHILE ~R.eof DO
  select;
  IF r[j].first <= R.first THEN
    copyrun;
    IF R.eof THEN INC(d[j]) ELSE copyrun END
  ELSE copyrun
  END
END
```

Now we are finally in a position to tackle the main polyphase merge sort algorithm. Its principal structure is similar to the main part of the *N*-way merge program: An outer loop whose body merges runs until the sources are exhausted, an inner loop whose body merges a single run from each source, and an innermost loop whose body selects the initial key and transmits the involved item to the target file. The principal differences to balanced merging are the following:

1. Instead of *N*, there is only one output sequence in each pass.
2. Instead of switching *N* input and *N* output sequences after each pass, the sequences are rotated. This is achieved by using a sequence index map *t*.
3. The number of input sequences varies from run to run; at the start of each run, it is determined from the counts d_i of dummy runs. If $d_i > 0$ for all *i*, then *N*-1 dummy runs are pseudo-merged into a single dummy run by merely incrementing the count d_N of the output sequence. Otherwise, one run is merged from all sources with $d_i = 0$, and d_i is decremented for all other sequences, indicating that one dummy run was taken off. We denote the number of input sequences involved in a merge by *k*.
4. It is impossible to derive termination of a phase by the end-of status of the *N*-1'st sequence, because more merges might be necessary involving dummy runs from that source. Instead, the theoretically necessary number of runs is determined from the coefficients a_i. The coefficients a_i were computed during the distribution phase; they can now be recomputed backward.

The main part of the Polyphase Sort can now be formulated according to these rules, assuming that all *N*-1 sequences with initial runs are set to be read, and that the tape map is initially set to $t_i = i$.

```
REPEAT (*merge from t[0] ... t[N-2] to t[N-1]*)
  z := a[N-2]; d[N-1] := 0;
  REPEAT k := 0;  (*merge one run*)
    (*determine no. of active input sequences*)
    FOR i := 0 TO N-2 DO
      IF d[i] > 0 THEN DEC(d[i]) ELSE ta[k] := t[i]; INC(k) END
    END ;
    IF k = 0 THEN INC(d[N-1])
    ELSE merge one real run from t[0] ... t[k-1] to t[N-1]
    END ;
    DEC(z)
  UNTIL z = 0;
  Runs.Set(r[t[N-1]], f[t[N-1]]);
  rotate sequences in map t; compute a[i] for next level;
  DEC(level)
UNTIL level = 0
(*sorted output is f[t[0]]*)
```

The actual merge operation is almost identical with that of the *N*-way merge sort, the only difference being that the sequence elimination algorithm is somewhat simpler. The rotation of the sequence index map and the corresponding counts d_i (and the down-level recomputation of the coefficients a_i) is straightforward and can be inspected in detail from Program 2.16, which represents the *Polyphase* algorithm in its entirety.

```
PROCEDURE Polyphase(src: Files.File): Files.File;
  VAR i, j, mx, tn: INTEGER;
    k, dn, z, level: INTEGER;
    x, min: INTEGER;
    a, d:  ARRAY N OF INTEGER;
    t, ta:  ARRAY N OF INTEGER;  (*index maps*)
    R: Runs.Rider;  (*source*)
    f: ARRAY N OF Files.File;
    r: ARRAY N OF Runs.Rider;

  PROCEDURE select;
    VAR i, z: INTEGER;
  BEGIN
    IF d[j] < d[j+1] THEN INC(j)
    ELSE
      IF d[j] = 0 THEN
        INC(level); z := a[0];
        FOR i := 0 TO N-2 DO
          d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
        END
      END ;
      j := 0
    END ;
    DEC(d[j])
  END select;

  PROCEDURE copyrun;  (*from src to f[j]*)
  BEGIN
    REPEAT Runs.copy(R, r[j]) UNTIL R.eor
  END copyrun;

BEGIN Runs.Set(R, src);
  FOR i := 0 TO N-2 DO
    a[i] := 1; d[i] := 1; f[i] := Files.New(""); Files.Set(r[i], f[i], 0)
  END ;
  (*distribute initial runs*)
  level := 1; j := 0; a[N-1] := 0; d[N-1] := 0;
  REPEAT select; copyrun UNTIL R.eof OR (j = N-2);
  WHILE ~R.eof DO
    select;  (*r[j].first = last item written on f[j]*)
    IF r[j].first <= R.first THEN
      copyrun;
      IF R.eof THEN INC(d[j]) ELSE copyrun END
    ELSE copyrun
    END
  END ;

  FOR i := 0 TO N-2 DO t[i] := i; Runs.Set(r[i], f[i]) END ;
  t[N-1] := N-1;
  REPEAT (*merge from t[0] ... t[N-2] to t[N-1]*)
    z := a[N-2]; d[N-1] := 0;
    f[t[N-1]] := Files.New(""); Files.Set(r[t[N-1]], f[t[N-1]], 0);
    REPEAT k := 0;  (*merge one run*)
      FOR i := 0 TO N-2 DO
        IF d[i] > 0 THEN DEC(d[i]) ELSE ta[k] := t[i]; INC(k) END
      END ;
```

```
      IF k = 0 THEN INC(d[N-1])
      ELSE (*merge one real run from t[0] ... t[k-1] to t[N-1]*)
        REPEAT mx := 0; min := r[ta[0]].first; i := 1;
          WHILE i < k DO
            x := r[ta[i]].first;
            IF x < min THEN min := x; mx := i END ;
            INC(i)
          END ;
          Runs.copy(r[ta[mx]], r[t[N-1]]);
          IF r[ta[mx]].eor THEN ta[mx] := ta[k-1]; DEC(k) END
        UNTIL k = 0
      END ;
      DEC(z)
    UNTIL z = 0;
    Runs.Set(r[t[N-1]], f[t[N-1]]);   (*rotate sequences*)
    tn := t[N-1]; dn := d[N-1]; z := a[N-2];
    FOR i := N-1 TO 1 BY -1 DO
      t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
    END ;
    t[0] := tn; d[0] := dn; a[0] := z; DEC(level)
  UNTIL level = 0 ;
  RETURN f[t[0]]
END Polyphase
```

2.4.5. Distribution of Initial Runs

We were led to the sophisticated sequential sorting programs, because the simpler methods operating on arrays rely on the availability of a random access store sufficiently large to hold the entire set of data to be sorted. Often such a store is unavailable; instead, sufficiently large sequential storage devices such as tapes or disks must be used. We know that the sequential sorting methods developed so far need practically no primary store whatsoever, except for the file buffers and, of course, the program itself. However, it is a fact that even small computers include a random access, primary store that is almost always larger than what is needed by the programs developed here. Failing to make optimal use of it cannot be justified.

The solution lies in combining array and sequence sorting techniques. In particular, an adapted array sort may be used in the distribution phase of initial runs with the effect that these runs do already have a length L of approximately the size of the available primary data store. It is plain that in the subsequent merge passes no additional array sorts could improve the performance because the runs involved are steadily growing in length, and thus they always remain larger than the available main store. As a result, we may fortunately concentrate our attention on improving the algorithm that generates initial runs.

Naturally, we immediately concentrate our search on the logarithmic array sorting methods. The most suitable of them is the tree sort or *Heapsort* method (see Sect. 2.2.5). The heap may be regarded as a funnel through which all items must pass, some quicker and some more slowly. The least key is readily picked off the top of the heap, and its replacement is a very efficient process. The action of funnelling a component from the input sequence *src* (rider r0) through a full heap *H* onto an output sequence *dest* (rider r1) may be described simply as follows:

```
Write(r1, H[0]); Read(r0, H[0]); sift(0, n-1)
```

Sift is the process described in Sect. 2.2.5 for sifting the newly inserted component H_0 down into its proper place. Note that H_0 is the least item on the heap. An example is shown in Fig. 2.17. The program eventually becomes considerably more complex for the following reasons:

1. The heap *H* is initially empty and must first be filled.
2. Toward the end, the heap is only partially filled, and it ultimately becomes empty.
3. We must keep track of the beginning of new runs in order to change the output index *j* at the right time.

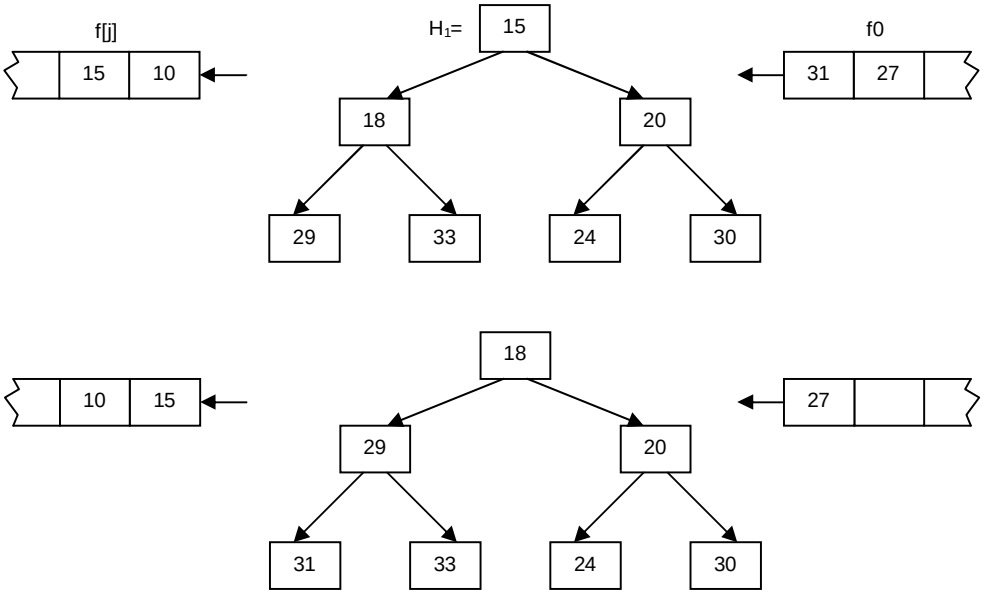

Fig. 2.17. Sifting a key through a heap

Before proceeding, let us formally declare the variables that are evidently involved in the process:

```
VAR L, R, x: INTEGER;
   src, dest: Files.File;
   r, w: Files.Rider;
   H: ARRAY M OF INTEGER;  (*heap*)
```

M is the size of the heap *H*. We use the constant *mh* to denote *M*/2; *L* and R are indices delimiting the heap. The funnelling process can then be divided into five distinct parts.

1. Read the first *mh* keys from src (r) and put them into the upper half of the heap where no ordering among the keys is prescribed.
2. Read another *mh* keys and put them into the lower half of the heap, sifting each item into its appropriate position (build heap).
3. Set *L* to *M* and repeat the following step for all remaining items on *src*: Feed H_0 to the appropriate output sequence. If its key is less or equal to the key of the next item on the input sequence, then this next item belongs to the same run and can be sifted into its proper position. Otherwise, reduce the size of the heap and place the new item into a second, upper heap that is built up to contain the next run. We indicate the borderline between the two heaps with the index *L*. Thus, the lower (current) heap consists of the items H_0 ... H_{L-1}, the upper (next) heap of H_L ... H_{M-1}. If $L = 0$, then switch the output and reset *L* to *M*.
4. Now the source is exhausted. First, set *R* to *M*; then flush the lower part terminating the current run, and at the same time build up the upper part and gradually relocate it into positions H_L ... H_{R-1}.
5. The last run is generated from the remaining items in the heap.

We are now in a position to describe the five stages in detail as a complete program, calling a procedure *switch* whenever the end of a run is detected and some action to alter the index of the output sequence has to be invoked. In Program 2.17 a dummy routine is used instead, and all runs are written onto sequence *dest*.

If we now try to integrate this program with, for instance, *Polyphase Sort*, we encounter a serious difficulty. It arises from the following circumstances: The sort program consists in its initial part of a fairly complicated routine for switching between sequence variables, and relies on the availability of a procedure *copyrun* that delivers exactly one run to the selected destination. The *Heapsort* program, on the other hand, is a complex routine relying on the availability of a closed procedure *select* which simply selects a new destination. There

would be no problem, if in one (or both) of the programs the required procedure would be called at a single place only; but instead, they are called at several places in both programs.

This situation is best reflected by the use of a *coroutine* (thread); it is suitable in those cases in which several processes coexist. The most typical representative is the combination of a process that produces a stream of information in distinct entities and a process that consumes this stream. This producer-consumer relationship can be expressed in terms of two coroutines; one of them may well be the main program itself. The coroutine may be considered as a process that contains one or more breakpoints. If such a breakpoint is encountered, then control returns to the program that had activated the coroutine. Whenever the coroutine is called again, execution is resumed at that breakpoint. In our example, we might consider *Polyphase Sort* as the main program, calling upon *copyrun*, which is formulated as a coroutine. It consists of the main body of Program 2.17 in which each call of *switch* now represents a breakpoint. The test for *end of file* would then have to be replaced systematically by a test of whether or not the coroutine had reached its endpoint.

```
PROCEDURE Distribute(src: Files.File): Files.File;
  CONST M = 16; mh = M DIV 2;  (*heap size*)
  VAR L, R: INTEGER;
    x: INTEGER;
    dest: Files.File;
    r, w: Files.Rider;
    H: ARRAY M OF INTEGER;  (*heap*)

  PROCEDURE sift(L, R: INTEGER);
    VAR i, j, x: INTEGER;
  BEGIN i := L; j := 2*L+1; x := H[i];
    IF (j < R) & (H[j] > H[j+1]) THEN INC(j) END ;
    WHILE (j <= R) & (x > H[j]) DO
      H[i] := H[j]; i := j; j := 2*j+1;
      IF (j < R) & (H[j] > H[j+1]) THEN INC(j) END
    END ;
    H[i] := x
  END sift;

BEGIN Files.Set(r, src, 0); dest := Files.New(""); Files.Set(w, dest, 0);
(*step 1: fill upper half of heap*)
  L := M;
  REPEAT DEC(L); Files.ReadInt(r, H[L]) UNTIL L = mh;
(*step 2: fill lower half of heap*)
  REPEAT DEC(L); Files.ReadInt(r, H[L]); sift(L, M-1) UNTIL L = 0;
(*step 3: pass elements through heap*)
  L := M; Files.ReadInt(r, x);
  WHILE ~r.eof DO
    Files.WriteInt(w, H[0]);
    IF H[0] <= x THEN
      (*x belongs to same run*) H[0] := x; sift(0, L-1)
    ELSE (*start next run*)
      DEC(L); H[0] := H[L]; sift(0, L-1); H[L] := x;
      IF L < mh THEN sift(L, M-1) END ;
      IF L = 0 THEN (*heap full; start new run*) L := M END
    END ;
    Files.ReadInt(r, x)
  END ;
(*step 4: flush lower half of heap*)
  R := M;
  REPEAT DEC(L); Files.WriteInt(w, H[0]);
    H[0] := H[L]; sift(0, L-1); DEC(R); H[L] := H[R];
    IF L < mh THEN sift(L, R-1) END
  UNTIL L = 0;
```

```
(*step 5: flush upper half of heap, start new run*)
  WHILE R > 0 DO
    Files.WriteInt(w, H[0]); H[0] := H[R]; DEC(R); sift(0, R)
  END ;
  RETURN dest
END Distribute
```

Analysis and conclusions. What performance can be expected from a *Polyphase Sort* with initial distribution of runs by a *Heapsort*? We first discuss the improvement to be expected by introducing the heap.

In a sequence with randomly distributed keys the expected average length of runs is 2. What is this length after the sequence has been funnelled through a heap of size *m* ? One is inclined to say *m*, but, fortunately, the actual result of probabilistic analysis is much better, namely *2m* (see Knuth, vol. 3, p. 254). Therefore, the expected improvement factor is *m*.

An estimate of the performance of Polyphase can be gathered from Table 2.15, indicating the maximal number of initial runs that can be sorted in a given number of partial passes (levels) with a given number *N* of sequences. As an example, with six sequences and a heap of size *m* = 100, a file with up to 165'680'100 initial runs can be sorted within 10 partial passes. This is a remarkable performance.

Reviewing again the combination of *Polyphase Sort* and *Heapsort*, one cannot help but be amazed at the complexity of this program. After all, it performs the same easily defined task of permuting a set of items as is done by any of the short programs based on the straight array sorting principles. The moral of the entire chapter may be taken as an exhibition of the following:

1. The intimate connection between algorithm and underlying data structure, and in particular the influence of the latter on the former.
2. The sophistication by which the performance of a program can be improved, even when the available structure for its data (sequence instead of array) is rather ill-suited for the task.

Exercises

2.1. Which of the algorithms given for straight insertion, binary insertion, straight selection, bubble sort, shakersort, shellsort, heapsort, quicksort, and straight mergesort are stable sorting methods?

2.2. Would the algorithm for binary insertion still work correctly if L < R were replaced by L < R in the while clause? Would it still be correct if the statement L := m+1 were simplified to L := m? If not, find sets of values a1 ... an upon which the altered program would fail.

2.3. Program and measure the execution time of the three straight sorting methods on your computer, and find coefficients by which the factors C and M have to be multiplied to yield real time estimates.

2.4. Specifty invariants for the repetitions in the three straight sorting algorithms.

2.5. Consider the following "obvious" version of the procedure Partition and find sets of values a_0 ... a_{n-1} for which this version fails:

```
i := 0; j := n-1; x := a[n DIV 2];
REPEAT
  WHILE a[i] < x DO i := i+1 END ;
  WHILE x < a[j] DO j := j-1 END ;
  w := a[i]; a[i] := a[j]; a[j] := w
UNTIL i > j
```

2.6. Write a procedure that combines the Quicksort and Bubblesort algorithms as follows: Use Quicksort to obtain (unsorted) partitions of length m (1 < m < n); then use Bubblesort to complete the task. Note that the latter may sweep over the entire array of n elements, hence, minimizing the bookkeeping effort. Find that value of m which minimizes the total sort time. Note: Clearly, the optimum value of m will be quite small. It may therefore pay to let the Bubblesort sweep exactly m-1 times over the array instead of including a last pass establishing the fact that no further exchange is necessary.