CS321 Lecture: External Sorting 3/21/88 - revised 11/22/99

Materials: Transparencies of basic merge sort, balanced 2-way merge sort, Natural merge, stable natural merge, merge with internal run generation, merge with replacement selection during run generation. Transparency of Knuth "snowplow" analogy, polyphase analysis.

Introduction

A. We have seen that the algorithms we use for searching tables stored on disk are quite different from those used for searching tables stored in main memory, because the disk access time dominates the processing time.

B. For much the same reason, we use different algorithms for sorting information stored on disk than for sorting information in main memory.

1. We call an algorithm that sorts data contained in main memory an INTERNAL SORTING algorithm, while one that sorts data on disk is called an EXTERNAL SORTING algorithm.

2. In the simplest case - if all the data fits in main memory - we can simply read the data from disk into main memory, sort it using an internal sort, and then write it back out.

3. The more interesting case - and the one we consider here - arises when the file to be sorted does not all fit in main memory.

4. Historically, external sorting algorithms were developed in the context of systems that used magnetic tapes for file storage, and the literature still uses the term "tape", even though files are most often kept on some form of disk. It turns out, though, that the storage medium being used doesn't really matter because the algorithms we will consider all read/write data sequentially.

I. A Survey of External Sorting Methods

A. Most external sorting algorithms are variants of a basic algorithm known as EXTERNAL MERGE sort. Note that there is also an internal version of merge sort that we have considered. External merging reads data one record at a time from each of two or more files, and writes records to one or more output files. As was the case with internal merging, external merging is O(n log n) for time, but O(n) for extra space, and (if done carefully) it is stable.

B. First, though, we need to review some definitions:

1. A RUN is a sequence of records that are in the correct relative order.

2. A STEPDOWN normally occurs at the boundary between runs. Instead of the key value increasing from one record to the next, it decreases.

Example: In the following file: B D E C F A G H

- we have three runs (B D E, C F, A G H)

- we have two stepdowns (E C, F A)

3. Observe that an unsorted file can have up to n runs, and up to n-1 stepdowns. In general (unless the file is exactly backwards) there will be a lesser number than this of runs and stepdowns, due to pre-existing order in the file.

4. Observe that a sorted file consists of one run, and has no stepdowns.

C. We begin with a variant of external merge sort that one would not use directly, but which serves as the foundation on which all the other variants build.

1. In the simplest merge sort algorithm, we start out by regarding the file as composed of n runs, each of length 1. (We ignore any runs which may already be present in the file.) On each pass, we merge pairs of runs to produce runs of double length.

   a. After pass 1, we have n/2 runs of length 2.

   b. After pass 2, we have n/4 runs of length 4.

   c. The total number of passes will be ceil(log n). [Where ceil is the ceiling function - smallest integer greater than or equal to.] After the last pass, we have 1 run of length n, as desired.

   d. Of course, unless our original file length is a power of 2, there will be some irregularities in this pattern. In particular, we let the last run in the file be smaller than all the rest - possibly even of length zero.

      Example: To sort a file of 6 records:

      ```
      Initially:          6 runs of length 1
      After pass 1:       3 runs of length 2 + 1 "dummy" run of length 0
      After pass 2:       1 run of length 4 + 1 run of length 2
      After pass 3:       1 run of length 6
      ```

2. We will use a total of three scratch files to accomplish the sort.

   a. Initially, we distribute the input data over two files, so that half the runs go on each. We do this alternately - i.e. first we write a run to one file, then to the other - in order to ensure stability.

   b. After the initial distribution, each pass entails merging runs from two of the scratch files and writing the generated runs on the third. At the end of the pass, if we are not finished, we redistribute the runs from the third file alternately back to the first two.

   Example: original file: B D E C F A G H

   ```
            initial distribution: B E F G         (File SCRATCH1)
                                  D C A H         (File SCRATCH2)

            (remember we ignore runs existing in the raw data)

            ------------------------------------------------
            after first merge:    BD CE AF GH     (File SCRATCH3)
                                                              PASS 1
            redistribution:       BD AF           (File SCRATCH1)
                                  CE GH           (File SCRATCH2)

            ------------------------------------------------
            after second merge:   BCDE AFGH       (File SCRATCH3)
                                                              PASS 2
            redistribution:       BCDE            (File SCRATCH1)
                                  AFGH            (File SCRATCH2)
   ```

```
                ------------------------------------------------
                after third merge:    ABCDEFGH        (File SCRATCH3)  PASS 3

                (no redistribution)
```

3. Code: TRANSPARENCY

4. Analysis: we said earlier that all variants of external merge sort are O(nlogn) for time and O(n) for extra space. To compare different variants, we will need to use a more precise figure than just these "big O" bounds.

   a. We will, therefore, measure space in terms of the total number of records of scratch space needed. We will also give consideration to main memory space needed for buffers for the files and (later) for other purposes as well.

   b. We will measure time in terms of the number of records read and written.

      i. Actually, since each record read is eventually written to another file, it will suffice to simply count reads, and then double this number to get total records transferred.

      ii. Of course, data is normally transferred to/from the file in complete blocks, rather than record-by-record. However, the total number of transfers is directly proportional to the number of records transferred, so we will use the number of records transferred as our measure.

5. Analysis of the basic merge sort

   a. Space: three files, one of length n and two of length n/2. We can use the output file as one of the scratch files, so the total additional space is two files of length n/2

      = total scratch space for n records

   In addition, we need internal memory for three buffers - one for each of the three files. In general, each buffer needs to be big enough to hold an entire block of data (based on the blocksize of the device), rather than a single record.

   b. Time:

      - Initial distribution involves n reads

      - Each pass except the last involves 2n reads due to merging followed by redistribution. The last pass involves just n reads.

      - Total reads = 2 n ceil(log n), so total IO operations =

                4n ceil(log n)

D. Improving the basic merge sort: two possible improvements suggest themselves immediately, and entail minimal extra work.

   1. If we had four scratch files instead of three, we could combine the merging and redistribution into one operation as follows: On each pass, we use two files for input and two for output. We write the runs generated alternately to the two output files. After the pass, we switch the roles of the input and output files.

      a. Example: original file: B D E C F A G H

```
initial distribution: B E F G         (File SCRATCH1)
                      D C A H         (File SCRATCH2)

(remember we ignore runs existing in the raw data)

after first pass:     BD AF           (File SCRATCH3)
                      CE GH           (File SCRATCH4)

after second pass:    BCDE            (File SCRATCH1)
                      AFGH            (File SCRATCH2)

after third pass:     ABCDEFGH        (File SCRATCH3)
```

b. Code: TRANSPARENCY

c. Analysis:

i. Space: four files, two of which must now be of length n, since we don't know ahead of time, in general, whether we will have an odd or even number of passes. (The sorted file will end up on either the first or third scratch file.) The other two files can be of length n/2. As before, we can use the output file as one of the scratch files, so the extra space needed is one file of length n plus two of length n/2

= total scratch space for 2n records

Plus internal memory for 4 file buffers.

ii. Time: Because we have an initial distribution pass before we start merging, we have a total of

n (1 + ceil(log n)) reads = 2n(1 + ceil(log n)) transfers

iii. We have gained almost a factor of 2 in speed, at the cost of doubling the scratch file space.

d. This algorithm is known as BALANCED 2-WAY MERGE SORT.

2. Another improvement arises from the observation that our original algorithm started out assuming that the input file consists of n runs of length 1 - the worst possible case (a totally backward file.) In general, the file will contain many runs longer than one just as a consequence of the randomness of the data, and we can use these to reduce the number of passes.

a. Example: The sample file we have been using contains 3 runs, so we could do our initial distribution as follows:

```
initial distribution: BDE AGH         (File SCRATCH1)
                      CF              (File SCRATCH2)

after first pass:     BCDEF           (File SCRATCH3)
                      AGH             (File SCRATCH4)

after second pass:    ABCDEFGH        (File SCRATCH1)
```

(Note: we have assumed the use of a balanced merge; but a non-balanced merge could also have been used.)

b. Code: TRANSPARENCY

c. This algorithm is called a NATURAL MERGE. The term "natural" reflects the fact that it relies on runs naturally occurring in the data.

d. However, this algorithm has a quirk we need to consider.

   i. Since we merge one run at a time, we need to know where one run ends and another run begins. In the case of the previous algorithms, this was not a problem, since we knew the size of each run. Here, though, the size will vary from run to run.

      In the code we just looked at, the the solution to this problem involved recognizing that the boundary between runs is marked by a stepdown. Thus, each time we read a new record from an input file, we will keep track of the last key processed from that file; and if our newly read key is smaller than that key, then we know that we have finished processing one run from that file.

      Example: in the initial distribution above, we placed two runs in the first scratch file. The space between them would not be present in the file; what we would have is actually BDEAGH. But the run boundary would be apparent because of the stepdown from E to A.

   ii. However, if stability is important to us, we need to be very careful at this point. In some cases, the stepdown between two runs could disappear, and an unstable sort could result. Consider the following file:

```
F E D C B A M1 Z N M2          (where records M1 and M2 have
                                identical keys.)

                                       ___ No stepdown here, so
                                      |    2 runs look like one:
                                      v
Initial distribution:   F | D | B      | N
                        E | C | A M1 Z | M2

                        F | D | B N
                        E | C | A M1 Z | M2

First pass:             E F | A B M1 N Z
                        C D | M2
                            ^
                            |___ No stepdown here, so two runs
                                 look like one:

                        E F    | A B M1 N Z
                        C D M2 |

Second pass:            C D E F M2
                        A B M1 N Z

Third pass:             A B C D E F M2 M1 N Z
                                       ^
                                       |
                        In the case of equal keys, we take
                        record from first scratch file before
                        record from second, since first
                        scratch file should contain records
                        from earlier in original file.
```

   iii. If stability is a concern, we can prevent this from occurring by writing a special run-separator record between runs in our scratch files. This might, for example, be a record whose key is some impossibly big value like maxint or '~~~~~'. Of course, processing these records takes extra overhead that reduces the advantage gained by using the natural runs.

e. Code for a stable variant of natural merge: TRANSPARENCY

f. Analysis:

i. Space is the same as an ordinary merge if no run separator records are used. However, in the worst case of a totally backward input file, we would need n run separator records on our initial distribution, thus potentially doubling the scratch space needed.

ii. The time will be some fraction of the time needed by an ordinary merge, and will depend on the average length of the naturally occurring runs.

- If the naturally occurring runs are of average length 2, then we save 1 pass - in effect we start where we would be on the second pass of ordinary merge.

- In general, if the naturally occuring runs are of average length m, we save at least floor(log m) passes. Thus, if we use a balanced 2-way merge, our time will be

```
n (1 + ceil(log n - log m)) reads =

n (1 + ceil(log n/m)) reads or

2n (1 + ceil(log n/m)) IO operations
```

- Of course, if run separator records are used, then we actually process more than n records on each pass. This costs additional time for

```
n/m  reads on first pass
n/2m reads on second pass
n/4m reads on third pass
...

= (2n/m - 1) additional reads,

or about 4n/m extra IO operations
```

- Obviously, a lot depends on the average run length in the original data (m). It can be shown that, in totally random data, the average run length is 2 - which translates into a savings of 1 merge pass, or 2n IO operations. However, if we use separator records, we would need 2n extra IO operations to process them - so we gain nothing! (We could still gain a little bit by omitting separator records if stability were not an issue, though.)

- In many cases, though, the raw data does contain considerable natural order, beyond what is expected randomly. In this case, natural merging can help us a lot.

E. Two further improvements to the basic merge sort are possible at the cost of significant added complexity. The first improvement builds on the idea of the natural merge by using an internal sort during the distribution phase to CREATE runs of some size.

1. The initial distribution pass now looks like this - assuming we have room to sort s records at a time internally:

```
while not eof(infile) do
   read up to s records into main memory
```

```
      sort them
      write them to one of the scratch files

2. Clearly, the effect of this is to reduce the merge time by a
   factor of (log (n/s)) / (log n).  For example, if s = sqrt(n),
   we reduce the merge time by a factor of 2.  The overall time is
   not reduced as much, of course, because

   a. The distribution pass still involves the same number of reads.

   b. We must now add time for the internal sorting!

   c. Nonetheless, the IO time saved makes internal run generation
      almost always worthwhile.

   Example: suppose we need to sort 65536 records, and have room
            to internally sort 1024 at a time.

   - The time for a simple merge sort is

          65536 * (1 + log 65536) reads + the same number of writes

          = 65536 * 17 * 2 = 2,228,224 IO operations

   - The time with internal run generation is

          65536 * (1+ log 65536/1024) reads + the same number of writes +
            internal sort time

          = 65536 * 7 * 2 = 917,504 IO operations + 64 1024-record sorts

3. Code: TRANSPARENCY

   Note that this particular algorithm is unstable. Why?  (ASK CLASS)

   a. The merging process is perfectly stable.

   b. But we use an unstable sort - quicksort - for initial run
      generation.

   c. Clearly, if stability were an issue, we would have to switch to
      a stable internal sort.

4. Actually, we have room for further improvements here.  The limiting
   factor on the size of the initial runs we can generate is obviously
   the amount of main memory available for the initial sort.

   a. If we had enough room, we could just sort all of the records
      internally at one time and be done with the job.  The fact that
      we are using external sorting presupposes limited main memory
      capacity.

   b. However, let's consider what happens after we have completed our
      internal sort and we are writing our run out to a scratch file.
      Each time we write a record out, we vacate a slot in main memory
      that could be used to hold a new record from the input file.
      Suppose we read one in at this point.  There are two possibilities:

      i. It could have a key >= the key of the record we just wrote out.
         If so, we can insert it into its proper position vis-a-vis the
         remaining records, and include it in the present run, thus
         increasing the run size by 1.

     ii. It could have a key < the key of the record we just wrote out.
         In this case, it cannot go into the present run, since it would
```

cause a stepdown. So we must put it at one end of the buffer to keep it out of the way.

c. This process is known as REPLACEMENT SELECTION. As each record is written to the file, a new one is read, and is either included in the present run or stuck away at the end of the internal buffer. Eventually, the records stuck away will fill the buffer, of course - at which point the run terminates.

d. It can be shown that, with random data, replacement selection increases the average run size by a factor of 2 - i.e. it is like doubling the size of the internal sorting buffer.

   Knuth gives an argument to show this based on an analogy to a snowplow plowing snow in a steady storm. (TRANSPARENCY - READ TEXT)

e. Code: TRANSPARENCY

   Note that this code is unstable, for three reasons:

   i. Use of unstable internal sorting method - heapsort - initially.

   ii. The replacement selection process is unstable. A newly read record, because it is placed on top of the heap, could get ahead of a preceeding record with the same key.

   iii. We detect run boundaries by using stepdowns, which can allow two runs to collapse into one in the absence of separator records (which we don't use.)

   iv. Two of these four problems (the first and third) could be easily fixed. As it turns out, though, the second cannot be fixed without going to an O(n^2) kind of process for readjusting the internal buffer after each new record is brought in. Thus, we leave the algorithm in an unstable form.

5. Data structures for replacement selection

   a. There are several different internal structures that can be used to facilitate replacement selection. In general, we want to sort the raw data, and then output records 1 at a time - each time replacing the record we outputted with a new one from the input and - if possible - putting it into its proper place vis-a-vis the remaining records. Ordinarily we would like to do the original sorting in O(n log n) time, and we would like each replacement to take O(log n) time. We do n replacements, so the total time for the output/replacement phase would then also be O(n log n).

   b. The example we just looked at uses a variant of heapsort, but builds an inverted heap in which the SMALLEST key is on top of the heap (rather than the largest). Further, we require each other key to be >= its parent (not <= as in standard heapsort.)

      i. This configuration can be achieved by running just the first phase of heapsort, with the inequalities reversed. (See BuildInvertedHeap in TRANSPARENCY). Clearly, this takes O(n log n) time.

      ii. As we output each record, we run one iteration of the second phase of heapsort (with inequalities reversed) to bring a new record from the top of the heap to its right place. (See AdjustInvertedHeap in TRANSPARENCY.)

          - If possible, the new record we put on top of the heap is from the input file.

- Otherwise, we promote a record from the rear of the heap, and put the newly-read record into its place - thus reducing the effective size of the heap.

Either way, each adjustment takes O(log n) time, so the expected n adjustments take O(n log n) time.

iii. Example: suppose we have internal storage to sort three records, and are generating runs from the following input file:

D B G F A H C I E

Initial read-in:

```
 D
B G
```

Build heap

```
 B
D G
```

Output B to run, replace with F:

```
 F
D G
```

Adjust heap

```
 D
F G
```

Output D to run. Since A is too small to participate in this run, replace D with G, G with A, and reduce size of heap:

```
 G __
F / A <- not part of heap
```

Adjust heap

```
 F __
G /A
```

Output F to run, replace with H:

```
 H ___
G / A
```

Adjust heap:

```
 G ___
H / A
```

Output G to run. Since C is too small to participate in this run, replace G with H, H with C, and reduce size of heap:

```
 H
----
C A <- not part of heap
```

Adjust heap (no change)

Output H to run, replace with I:

```
 I
----
C A
```

Adjust heap (no change)

Output I to run. Since E is too small to participate in this run, replace I with E and reduce size of heap:

```
 E       <- all not on heap
C A
```

Since the buffer is now filled up with records not eligible for the current run, terminate the current run. The records in the buffer can then be converted into a heap to start the

next run.

Resulting run: B D F G H I

c. Other structures can also be used - see Horowitz or Knuth. The structures they discuss are based on the idea of a tournament in which pairs of players compete; then the winners of each pair compete in new pairs etc until an overall winner is chosen. (Here the "winner" is the record with the smallest key). The "winner" is outputted to the run, and a new record - if eligible - is allowed to work its way through the tournament along the path the winner previously followed.

d. None of these structures produces a stable sort, however. For a stable sort, one would have to use an insertion or selection sort-like structure, with O(n) time for each replacement instead of O(log n) time.

F. Another improvement involves increasing the MERGE ORDER.

1. In our balanced merge sort, we merge two input files to produce two output files, then exchange roles and keep going - using a total of four files.

2. Suppose, however, that we could use six files. This would mean merging runs from 3 files and a time, and distributing the resulting runs alternately into 3 output files. How would this affect the overall run time?

a. Each merge pass would still process all n records, requiring 2n IO operations (one read + one write for each record.)

b. However, each pass would cut the number of runs by a factor of 3. Thus, we would need ceil(log n) passes instead of ceil(log n).
3 2
Since (log n) / (log n) = (log 2) = 0.63, this translates into
3 2 3
about a 37% savings in the number of passes and hence almost as big a savings in the number of IO operations. (Our proportional savings overall are less than 37% because we still have to do the initial distribution pass.)

3. Of course, further savings are possible by increasing the number of files. With eight files, we merge 4 runs at a time, and thus cut the number of passes in half. In general, a BALANCED M-WAY MERGE SORT has the following behavior:

a. Time: 2n (1 + ceil(log n)) IO operations.
m
(Number of merge passes reduced from 2-way merge by a factor of log 2.)
m

b. Space: 2m files - of which two are length n and the rest are of length n/m. If we use the output file as one of these, then the total scratch space needed is room for

$$n + (2m - 2)(n/m) = 3n - 2n/m \quad \text{records}$$

plus room for 2m buffers in main memory

(Notice that, when m=2, these reduce to our previous formulas for balanced 2-way merge sort, as we would expect.)

4. We can use both internal sorting and increased merge order together

to get even better performance. For example, an m-way merge sort of n records, assuming the ability to sort s records at a time internally, would require $\log_m (n/s)$ passes.

Example: 65536 records, sorted 1024 at a time, using 4-way merge:

Number of IOs = $65536 * (1 + \log_4 (65536/1024)) * 2$

= $65536 * (1 + 3) * 2 = 524{,}288$ IO operations - almost twice as good as our previous example using 1024 record internal sorting and 2-way merge.

G. Polyphase merging

1. We have seen that we can speed up an external merge sort by increasing the merge order. Obviously, if we carry this to its limit, we could sort a file of n records in just one pass by using n-way merging.

   a. The external storage required for scratch files would be room for 3n - (2n/n) = about 3n records, which seems manageable.

   b. However, the fly in the ointment would be the need for 2n buffers in main memory - each capable of holding at least one record. If we had internal memory sufficient to hold 2n records, we wouldn't be using external sorting in the first place!

   c. Thus, in general, internal memory considerations will limit the merge order to some value m (m >= 2), such that we have room for 2m buffers in main memory and no more.

2. These computations have assumed that we use a balanced merge sort. Suppose, however, that we had room for 2m buffers, but used an unbalanced merge - i.e.

   a. We merge 2m - 1 runs at a time to produce runs in a single output file.

   b. We then redistribute those runs over 2m - 1 files, as in our original basic merge sort algorithm.

   c. The space needed is obviously the same as for balanced merging. What about the time?

      - we now need $\log_{(2m-1)} n$ passes, as compared to $1 + \log_m n$.

      - but each pass requires 4n IO operations as opposed to 2n because we merge and redistribute in separate steps. (The initial distribution and the fact that no redistribution is needed after the last pass cancel each other.)

      - Thus, total IOs = $4n \log_{(2m-1)} n$ versus $2n (1 + \log_m n)$

        - alas, the figure for the unbalanced version is always slightly bigger than the figure for the balanced version. This is because the log is not quite cut in half, while the number of IOs per pass are doubled due to the need to redistribute.

3. We now consider a sorting algorithm known as POLYPHASE merging that uses 2m files to get a 2m-1 way merge WITHOUT a separate redistribution of runs after every pass. It thus has the advantages of an unbalanced

sort without the disadvantage.

4. Rather than discuss the strategy in general, let's look at an example. We will do a 2-way merge using 3 files:

```
Original file:         B D E C F A G H

Initial distribution:  B | E | F | G | H      (File SCRATCH1)
                       D | C | A              (File SCRATCH2)
                       (empty)                (File SCRATCH3)

          (note that one file contains more runs than the other - 5
           versus 3, and we have one empty file to receive the results
           of our first merge pass.  This is all part of the plan.
           Also, for simplicity we are ignoring runs existing in the
           raw data, treating it as 8 runs of length 1.)

After first phase:     G | H                  (File SCRATCH1)
                       (empty)                (File SCRATCH2)
                       BD | CE | AF           (File SCRATCH3)

          (note that we didn't use up all the runs in the first
           scratch file.  Rather, we stopped the pass when we ran out
           of runs in one of the scratch files - namely the second.
           That is, it is no longer the case that each pass processes
           all of the records.  Thus, we use the term "phase" instead
           of "pass" to avoid confusion.)


After second phase:    (empty)                (File SCRATCH1)
                       BDG | CEH              (File SCRATCH2)
                       AF                     (File SCRATCH3)

After third phase:     ABDFG                  (File SCRATCH1)
                       CEH                    (File SCRATCH2)
                       (empty)                (File SCRATCH3)

After fourth phase:    (empty)                (File SCRATCH1)
                       (empty)                (File SCRATCH2)
                       ABCDEFGH               (File SCRATCH3)
```

   Note that we end up with four phases - as opposed to three passes in a standard 2-way merge sort. However, the we don't process all the records on each phase. The total number of IOs is:

```
2 * (8 [initial distribution] + 6 [phase 1] + 6 [phase 2] +
     5 [phase 3] + 8 [phase 4]) = 2 * 33 = 66 - compared with

        2 * 8 * (1 + log 8) = 64 for a balanced 2-way merge

or      4 * 8 * (log 8) = 96 for an unbalanced 2-way merge
```

   That is, we got nearly the performance of a balanced 2-way merge (that would require 4 files) - but we only used three files!

5. Obviously, one key to the polyphase merge is the way runs are distributed. Let's look again at the numbers of runs on the two non-empty files:

```
Original file:         8
Initial distribution:  5 and 3
After phase 1:         3 and 2
After phase 2:         2 and 1
After phase 3:         1 and 1
Finally:               1
```

a. What do all these numbers have in common (ASK CLASS)?

   - They are all Fibonacci numbers!

b. To see why this works, note that, at any time, we have one file containing Fib_n runs, and another containing Fib_{n-1}. We merge Fib_n runs, yielding one file with Fib_{n-1} and one with $Fib_{n-1} - Fib_{n-2} = Fib_n$. This continues until we end up with Fib_2 (= 1) runs in one file, and Fib_1 (= 1) run in the other, yielding one file with one run, as desired.

c. Now it may appear that this pattern poses a problem, since it appears only to work if the TOTAL number of runs initially is a Fibonacci number. However, we can make it work in other cases by assuming the existence of one or more dummy runs in either or both files.

   Example: suppose we are sorting a file with 6 runs - say the following (ignoring pre-existing order):

   Original file: B D E C F A

   We distribute initially as follows: B | E | C | A + dummy
   (Note: we will look at the algorithm that does this shortly) D | F | + dummy

   It turns out that our merging is slightly more efficient if we regard the dummy runs as being at the START of the file, rather than the end.

   i. If we regard the dummies as being at the end, then our merging goes like this:

Initial distribution	B | E | C | A + dummy
12 read/writes	D | F | + dummy
After Phase 1 (merging 3 runs):	A + dummy
	(empty)
10 read/writes	BD | EF | C
After Phase 2 (merging 2 runs):	(empty)
	ABD | EF
10 read/writes	C
After Phase 3 (merging 1 run):	ABCD
	EF
8 read/writes	(empty)
After Phase 4 (merging 1 run):	(empty)
	(empty)
12 read/writes	ABCDEF

   Total read writes = 12 + 10 + 10 + 8 + 12 = 52

   ii. However, if we regard the dummies as being at the beginning of the files, then we have:

Initial distribution	dummy + B | E | C | A
12 read/writes	dummy + D | F |

```
   After Phase 1 (merging 3 runs)          C | A
                                           (empty)
    8 read/writes                          dummy + BD | EF

   After Phase 2 (merging 2 runs)          (empty)
                                           C | ABD
    8 read/writes                          EF

   After Phase 3 (merging 1 run)           CEF
                                           ABD
    6 read/writes                          (empty)

   After Phase 4 (merging 1 run):          (empty)
                                           (empty)
    12 read/writes                         ABCDEF

   Total read writes = 12 + 8 + 8 + 6 + 12          = 46
```

6. We have been considering a 2-way polyphase merge. If we have room for more scratch files, we can use a higher order merge.

   a. In this case, we base the distribution on GENERALIZED FIBONACCI NUMBERS.

   The pth order Fibonacci numbers $F_n^{(p)}$ are defined as follows:

$$F_n^{(p)} = 0 \text{ for } 0 \le n \le p - 2$$

$$F_n^{(p)} = 1 \text{ for } n = p - 1$$

$$F_n^{(p)} = F_{n-1}^{(p)} + F_{n-2}^{(p)} + \ldots + F_{n-p}^{(p)} \quad \text{for } n \ge p$$

   (I.e. after the basis cases, a given generalized Fibonacci number of order p is simply the sum of its p predecessors.)

   Example: the third-order Fibonacci numbers are the following sequence (where the first is counted as $F_0^{(3)}$):

   0 0 1 1 2 4 7 13 24 44 ...

   Note that the ordinary Fibonacci numbers are, in fact, Fibonacci numbers of order 2 by this definition

   b. At the start of any phase of a P-way polyphase merge, the P input files contain

$$F_n^{(p)}, \; F_n^{(p)} + F_{n-1}^{(p)}, \; F_n^{(p)} + F_{n-1}^{(p)} + F_{n-2}^{(p)}, \; \ldots, \; F_n^{(p)} + \ldots + F_{n-p+1}^{(p)}$$

   [1 term] [2 terms] [3 terms] [p terms]

   runs, respectively, for some n. During this phase, $F_n^{(p)}$ runs will be merged, leaving the following number of runs in each file:

$$0, \; F_{n-1}^{(p)}, \; F_{n-1}^{(p)} + F_{n-2}^{(p)}, \; \ldots, \; F_{n-1}^{(p)} + \ldots + F_{n-p+1}^{(p)}$$

[1 term] [2 terms] [p-1 terms]

In addition, the output file will now contain

$$F_n^{(p)} = F_{n-1}^{(p)} + \ldots + F_{n-p}^{(p)} \text{ runs}$$

[p terms]

But this has the same form as the initial distribution, with n replaced by n-1, and with the file that was the output file now occupying the last position in the list.

c. Eventually, the merge will reduce down to the pattern

$$F_{p-1}^{(p)}, F_{p-1}^{(p)} + F_{p-2}^{(p)}, \ldots, F_{p-1}^{(p)} + \ldots + F_0^{(p)}$$

$$= 1, 0, \ldots, 0$$

- which is our sorted file

d. Of course, we will not, in general, have a perfect number of runs to produce a Fibonacci distribution, so we will have to put some number of dummy runs at the front of one or more files.

e. Example - 3-way polyphase merge of our original 8 records.
(We will compare to a balanced 2-way merge that uses the same number of scratch files, and required a total of 64 record transfers.)

Original file: B D E C F A G H

Initial distribution
16 read/writes

B	C	A	G
D	F	H	
dummy + E			

Note: # of runs = 2, 3, 4 = $F_4^{(3)}, F_4^{(3)} + F_3^{(3)}, F_4^{(3)} + F_3^{(3)} + F_2^{(3)}$

Phase 1 - merge 2 runs:
10 read/writes

A	G
H	
(empty)	
BD	CEF

Phase 2 - merge 1 run:
8 read/writes

G	
(empty)	
ABDH	
CEF	

Phase 3 - merge 1 run:
16 read/writes

(empty)	
ABCDEFGH	
(empty)	
(empty)	

Total IO Operations = 16 + 10 + 8 + 16 = 50

f. There is a fairly straightforward way to calculate the distribution by hand:

i. If you are doing a P-way polyphase merge, you will use P+1 files - so draw P+1 columns on scratch paper. We will develop

the distributions by working BACKWARDS from the final state - so the first distribution we create will be that for the last phase of the merge, etc.

ii. Each row will record the distribution as of the start of one phase. The final distribution will, of course, be:

0 0 .... 0 1

And the distribution just before it will be:

1 1 .... 1 0

In each succeeding phase, the empty file will move left one slot - so at the start of the second-to-the-last phase, we have:

x x .... 0 x <- where the x's represent values to be filled in

Each x will be the sum of the value in this column for the previously calculated phase plus the value in the column that is now zero (1 in this case). Thus, our next distribution is

2 2 .. 2 0 1

And the next is

2+2 2+2 .. 0 0+2 1+2 =

4 4 .. 0 2 3

iii. Continue this process until the total number of runs at the start of the phase is >= the number of runs in the original input. Add dummy runs if necessary.

iv. Example: 3-way polyphase merge sort of raw data containing 17 runs intially (4 files used):

Final state	0 0 0 1	Total 1
Before last phase	1 1 1 0	Total 3
Before 2nd to last	2 2 0 1	Total 5
Before 3rd to last	4 0 2 3	Total 9
Before 4th to last	0 4 6 7	Total 17

7. Algorithm:

a. Assume a p-way polyphase merge, using p+1 files - here designated file[1] .. file[p+1]. The initial distribution will spread runs over file[1] .. file[p] - leaving file[p+1] empty. (In fact, if one wished to economize on files, file[p+1] could serve as the input file - but of course the input file would then be destroyed.)

b. The first merge phase will put its runs in file[p+1]; the second in file[p]; the third in file[p-1], etc. After a phase puts its runs in file[1], the next will put its runs in file[p+1] and thus repeat the cycle.

c. Variables used in both initial distribution and merging process:

level: current level number. Level 1 = final merge, 0 = done.
distribution[i]: total number of runs in file i (1 <= i <= p+1)
dummy[i]: number of dummy runs in file i (1 <= i <= p+1)

d. Initial distribution phase:

set level = 1, distribution[i] = 1 for all i <= p,

```
                    dummy[i] = 1 for all i <= p,
                    distribution[p+1] = dummy[p+1] = 0

    set f = 1   <- we will put the next run into file f

    generate one run and put it into file f

    (* Run generation may be by any method - treat each record as a
       run, natural runs, internal sorting, sorting with replacement
       selection *)

    dummy[f] := 0

    while the input file is not empty do
        if dummy[f] < dummy[f+1] then
            f := f + 1
        else
            if dummy[f] = 0 then
                (* At this point, all values of dummy[] must be zero *)
                level := level + 1;
                compute new values of distribution[] and dummy[]
            f := 1;
        generate one run and put it into file f
        dummy[f] := dummy[f] - 1

    where we compute new values of distribution[] and dummy[] as
    follows:

    merges_this_level := distribution[1];
    for i := 1 to p do
        dummy[i] := merges_this_level + distribution[i+1] -
                    distribution[i]
        distribution[i] := distribution[i] + dummy[i]

Example: Distribute 17 runs for a 3-way merge (p = 3)

Level  Distribution[i]/Dummy[i]        f #runs
                                         so far
       1       2       3       4

1      1/1     1/0     1/0     0/0     1  0    Initialization
       1/0                                1    Generate 1st run

               1/0                     2  2    First time thru loop
                       1/0             3  3    2nd time thru loop
                                            Enter loop 3rd time

       Compute new distribution.  Merge_this_level = 1

2      2/1     2/1     1/0             1

       2/0                                4    Finish loop
               2/0                     2  5    2nd time thru loop
                                            Enter loop 3rd time

       Compute new distribution.  Merges_this_level = 2.

3      4/2     3/1     2/1             1

       4/1                                6    Finish loop
       4/0                                7    2nd time thru loop
               3/0                     2  8    3rd time thru loop
                       2/0             3  9    4th time thru loop
                                            Enter loop 5th time
```

```
                Compute new distribution.  Merges_this_level = 4.

          4     7/3     6/3     4/2             1

                7/2                               10    Finish loop
                        6/2                     2 11    2nd time thru loop
                7/1                             1 12    3rd time thru loop
                        6/1                     2 13    4th time thru loop
                                4/1             3 14    5th time thru loop
                7/0                             1 15    6th time thru loop
                        6/0                     2 16    7th time thru loop
                                4/0             3 17    8th time thru loop

                OUT OF RUNS - 17 GENERATED SO FAR
```

e. Actually doing the merges

```
                for i := 1 to level do
                    case i mod (p + 1) of
                        1:   merge from files 1..p to p+1
                        2:   merge from files p+1, 1 .. p-1 to p
                        3:   merge from files p .. p+1, 1 .. p-2 to p-1
                        ...
                        0:   merge from files 2 .. p+1 to 1
```

Where for each merge, we do the following:

```
                    for j := 1 to distribution[last input file] do

                        if dummy[] > 0 for all input files then
                            dummy[output_file] := dummy[output_file] + 1
                        for each input file:
                            if dummy[this_file] > 0 then
                                dummy[this_file] := dummy[this_file] - 1
                                end_of_run[this_file] := true
                            else
                                end_of_run[this_file] := eof(file[this_file])
                        while not all files at end of run do
                            transfer record with smallest key among those not
                                at end of run to output file
```

8. Note that polyphase merging is NOT STABLE.

9. Knuth discusses, in detail, the analysis of polyphase merge sorting, various improvements to it, plus other similar algorithms. The following summarizes the time behavior as a function of the number of "tapes".

   TRANSPARENCY

II. The following summarizes the external sorts we have considered:

Method	Scratch space (records)	Records read/written	Internal space/time	Stable
Basic merge	n	4n ceil(log n)	3 buffers	yes
Balanced 2-way	2n	2n (1+ceil(log n))	4 buffers	yes
Balanced Natural - initial runs of average size m	2n	2n (1+ceil(log n/m))	4 buffers	no *
Balanced with	2n	2n (1+ceil(log n/s))	4 buffers +	yes

internal sort of s records			space to sort s records (n/s) log(s) internal sort time	
Balanced with internal sort & replacement selection - random data	2n	2n (1+ceil(log n/2s))= 2n (ceil (log n/s))	[SAME]	no
Balanced m-way	3n - 2n/m	2n (1+ceil log m n)	2m buffers	yes
Balanced m-way with internal sort of size s (no replacement selection)	3n - 2n/m	2n (1+ceil log m n/s)	2m buffers + space to sort s records	yes
Polyphase m-way	[must analyze distribution case-by-case]		m+1	no

* = can be made stable by using separator records

III. Sorting with multiple keys

A. Thus far, we have assumed that each record in the file to be sorted contains one key field. What if the record contains multiple keys - e.g. a last name, first name, and middle initial?

   1. We wish the records to be ordered first by the primary key (last name).

   2. In the case of duplicate primary keys, we wish ordering on the secondary key (first name).

   3. In the case of ties on both keys, we wish ordering on the tertiary key (middle initial).

   etc - to any number of keys.

B. The approach we will discuss here applies to BOTH INTERNAL AND EXTERNAL SORTS.

C. There are two techniques that can be used for cases like this:

   1. We can modify an existing algorithm to consider multiple keys when it does comparisons - e.g.

      a. Original algorithm says:

```
if item[i].key < item[j].key then
```

      b. Revised algorithm says:

```
if (item[i].primary_key < item[j].primary_key) or

   ((item[i].primary_key = item[j].primary_key) and
    (item[i].secondary_key < item[j].secondary_key) or

   ((item[i].primary_key = item[j].primary_key) and
    (item[i].secondary_key = item[j].secondary_key) and
```

```
                (item[i].tertiary_key < item[j].tertiary_key)) then
```

2. We can sort the same file several times, USING A STABLE SORT.

   a. First sort is on least significant key.

   b. Second sort is on second least significant key.

   c. Etc.

   d. Final sort is on primary key.

3. The first approach is useable when we are embedding a sort in a specific application package; the second is more viable when we are building a utility sorting routine for general use [but note that we are now forced to a stable algorithm.]