External Sorting

All the *internal sorting* algorithms require that the input fit into main memory. There are, however, applications where the input is much too large to fit into memory. For those external sorting algorithms, which are designed to handle very large inputs.

Why We Need New Algorithms

Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable. Shell sort compares elements $a[i]$ and $a[i - hk]$ in one time unit. Heap sort compares elements $a[i]$ and $a[i * 2]$ in one time unit. Quicksort, with median-of-three partitioning, requires comparing $a[left]$, $a[center]$, and $a[right]$ in a constant number of time units. If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially. Even if the data is on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.

The time it takes to sort the input is certain to be insignificant compared to the time to read the input, even though sorting is an $O(n \log n)$ operation and reading the input is only $O(n)$.

Model for External Sorting

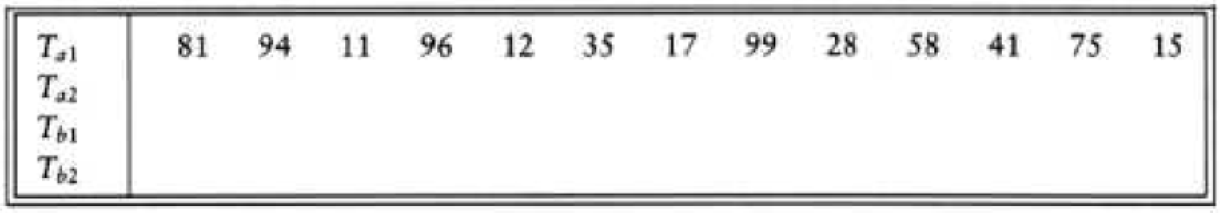
The wide variety of mass storage devices makes external sorting much more device dependent than internal sorting. The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium. Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order (in either direction).

We will assume that we have at least three tape drives to perform the sorting. We need two drives to do an efficient sort; the third drive simplifies matters. If only one tape drive is present, then we are in trouble: any algorithm will require $O(n^2)$ tape accesses.

The Simple Algorithm

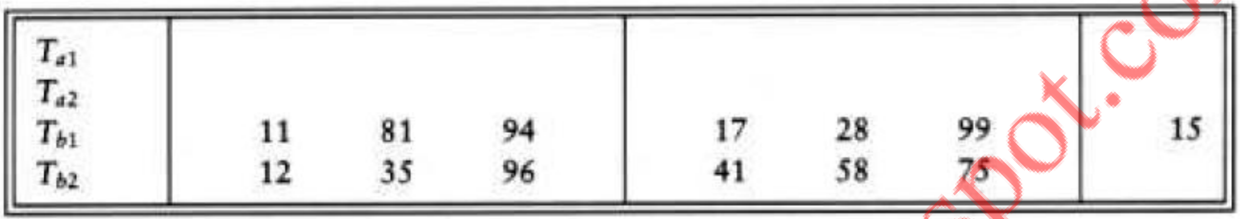
The basic external sorting algorithm uses the *merge* routine from merge sort. Suppose we have four tapes, $Ta1$, $Ta2$, $Tb1$, $Tb2$, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes.

Suppose the data is initially on $Ta1$. Suppose further that the internal memory can hold (and sort) *m* records at a time. A natural first step is to read *m* records at a time from the input tape, sort the records internally, and then write the sorted records alternately to $Tb1$ and $Tb2$. We will call each set of sorted records a *run*. When this is done, we rewind all the tapes. Suppose we have the same input as our example for Shell sort.



T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

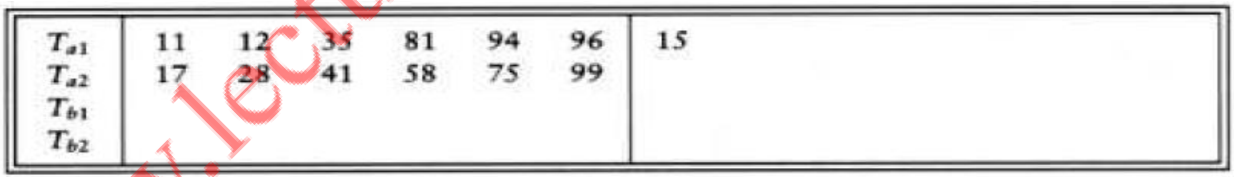
If *m* = 3, then after the runs are constructed, the tapes will contain the data indicated in the following figure.



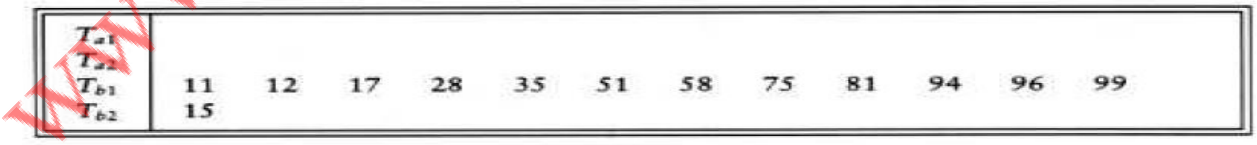
T_{a1}							
T_{a2}							
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

Now *Tb*1 and *Tb*2 contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto *Ta*1. Then we take the next run from each tape, merge these, and write the result to *Ta*2. We continue this process, alternating between *Ta*1 and *Ta*2, until either *Tb*1 or *Tb*2 is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes, and repeat the same steps, this time using the *a* tapes as input and the *b* tapes as output. This will give runs of 4*m*. We continue the process until we get one run of length *n*.

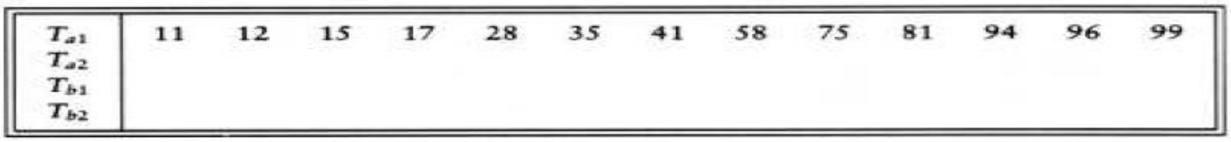
This algorithm will require log(*n/m*) passes, plus the initial run-constructing pass. For instance, if we have 10 million records of 128 bytes each, and four megabytes of internal memory, then the first pass will create 320 runs. We would then need nine more passes to complete the sort. Our example requires log 13/3 = 3 more passes, which are shown in the following figure.



T_{a1}	11	12	35	81	94	96	15
T_{a2}	17	28	41	58	75	99	
T_{b1}							
T_{b2}							



T_{a1}												
T_{a2}												
T_{b1}	11	12	17	28	35	51	58	75	81	94	96	99
T_{b2}	15											

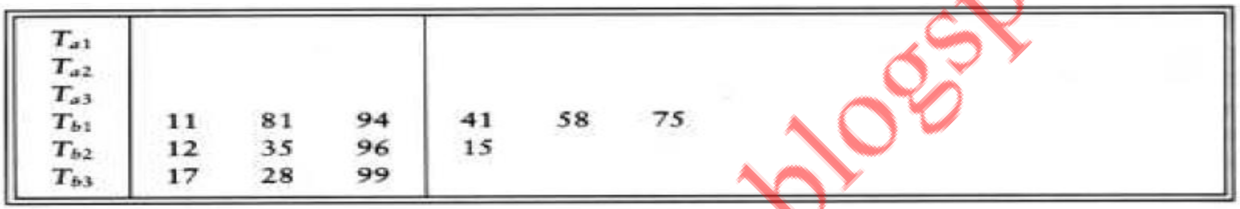


T_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

Multiway Merge

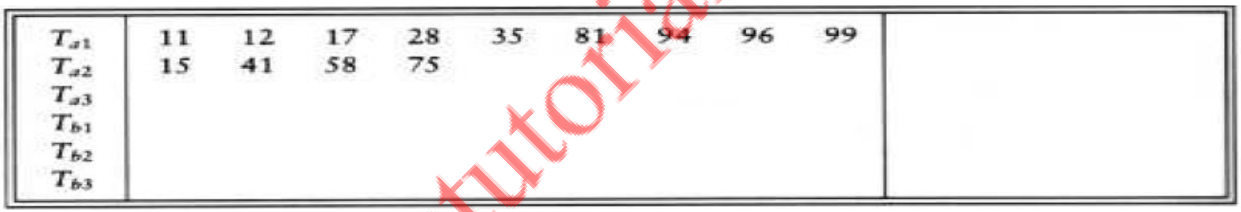
If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a *k*-way merge.

Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input tape is advanced. If there are *k* input tapes, this strategy works the same way, the only difference being that it is slightly more complicated to find the smallest of the *k* elements. We can find the smallest of these elements by using a priority queue. To obtain the next element to write on the output tape, we perform a *delete_min* operation. The appropriate input tape is advanced, and if the run on the input tape is not yet completed, we *insert* the new element into the priority queue. Using the same example as before, we distribute the input onto the three tapes.

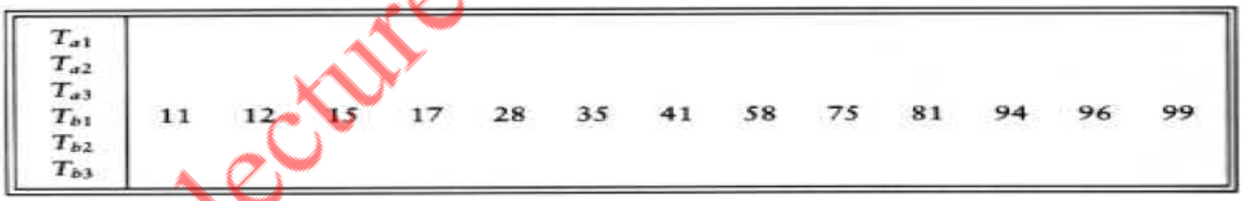


T_{a1}						
T_{a2}						
T_{a3}						
T_{b1}	11	81	94	41	58	75
T_{b2}	12	35	96	15		
T_{b3}	17	28	99			

We then need two more passes of three-way merging to complete the sort.



T_{a1}	11	12	17	28	35	81	94	96	99
T_{a2}	15	41	58	75					
T_{a3}									
T_{b1}									
T_{b2}									
T_{b3}									



T_{a1}													
T_{a2}													
T_{a3}													
T_{b1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{b2}													
T_{b3}													

After the initial run construction phase, the number of passes required using *k*-way merging is log*k*(*n*/*m*) , because the runs get *k* times as large in each pass. For the example above, the formula is verified, since log3 13/3 = 2. If we have 10 tapes, then *k* = 5, and our large example from the previous section would require log5 320 = 4 passes.

Polyphase Merge

The *k*-way merging strategy developed in the last section requires the use of 2*k* tapes. This could be prohibitive for some applications. It is possible to get by with only *k* + 1 tapes. As an example, we will show how to perform two-way merging using only three tapes.

Suppose we have three tapes, *T*1, *T*2, and *T*3, and an input file on *T*1 that will produce 34 runs. One option is to put 17 runs on each of *T*2 and *T*3. We could then merge this result onto *T*1, obtaining one tape with 17 runs. The problem is that since all the runs are on one tape, we must now put some of these runs on *T*2 to perform another merge. The logical way to do this is to copy the first eight runs from *T*1 onto *T*2 and then perform the merge. This has the effect of adding an extra half pass for every pass we do.

An alternative method is to split the original 34 runs unevenly. Suppose we put 21 runs on *T*2 and 13 runs on *T*3. We would then merge 13 runs onto *T*1 before *T*3 was empty. At this point, we could rewind *T*1 and *T*3, and merge *T*1, with 13 runs, and *T*2, which has 8 runs, onto *T*3. We could then merge 8 runs until *T*2 was empty, which would leave 5 runs left on *T*1 and 8 runs on *T*3. We could then merge *T*1 and *T*3, and so on. The following table below shows the number of runs on each tape after each pass.

	Run Const.	After T3 + T2	After T1 + T2	After T1 + T3	After T2 + T3	After T1 + T2	After T1 + T3	After T2 + T3
*T*1	0	13	5	0	3	1	0	1
*T*2	21	8	0	5	2	0	1	0
*T*3	13	0	8	3	0	2	1	0

The original distribution of runs makes a great deal of difference. For instance, if 22 runs are placed on *T*2, with 12 on *T*3, then after the first merge, we obtain 12 runs on *T*1 and 10 runs on *T*2. Afte another merge, there are 10 runs on *T*1 and 2 runs on *T*3. At this point the going gets slow, because we can only merge two sets of runs before *T*3 is exhausted. Then *T*1 has 8 runs and *T*2 has 2 runs. Again, we can only merge two sets of runs, obtaining *T*1 with 6 runs and *T*3 with 2 runs. After three more passes, *T*2 has two runs and the other tapes are empty. We must copy one run to another tape, and then we can finish the merge.

It turns out that the first distribution we gave is optimal. If the number of runs is a Fibonacci number *Fn*, then the best way to distribute them is to split them into two Fibonacci numbers *Fn*-1 and *Fn*-2. Otherwise, it is necessary to pad the tape with dummy runs in order to get the number of runs up to a Fibonacci number. We leave the details of how to place the initial set of runs on the tapes as an exercise.

We can extend this to a *k*-way merge, in which case we need *k*th order Fibonacci numbers for the distribution, where the *k*th order Fibonacci number is defined as $F^{(k)}$ (n) = $F^{(k)}(n - 1)$ + $F^{(k)}(n - 2)$ + + $F^{(k)}(n - k)$, with the appropriate initial conditions $F^{(k)}(n)$ = 0,0 n k - 2, $F^{(k)}(k - 1)$ =1.

Replacement Selection

The last item we will consider is construction of the runs. The strategy we have used so far is the simplest possible: We read as many records as possible and sort them, writing the result to some tape. This seems like the best approach possible, until one realizes that as soon as the first record is written to an output tape, the memory it used becomes available for another record. If the next record on the input tape is larger than the record we have just output, then it can be included in the run.

Using this observation, we can give an algorithm for producing runs. This technique is commonly referred to as *replacement selection*.

Initially, *m* records are read into memory and placed in a priority queue. We perform a *delete_min*, writing the smallest record to the output tape. We read the next record from the input tape. If it is larger than the record we have just written, we can add it to the priority queue. Otherwise, it cannot go into the current run. Since the priority queue is smaller by one element, we can store this new element in the dead space of the priority queue until the run is completed and use the element for the next run. Storing an element in the dead space is similar to what is done in heapsort. We continue doing this until the size of the priority queue is zero, at which point the run is over. We start a new run by building a new priority queue, using all the elements in the dead space. Figure 7.18 shows the run construction for the small example we have been using, with *m* = 3. Dead elements are indicated by an asterisk.

In this example, replacement selection produces only three runs, compared with the five runs obtained by sorting. Because of this, a three-way merge finishes in one pass instead of two. If the input is randomly distributed, replacement selection can be shown to produce runs of average length 2*m*. For our large example, we would expect 160 runs instead of 320 runs, so a five-way merge would require four passes. In this case, we have not saved a pass, although we might if we get lucky and have 125 runs or less. Since external sorts take so long, every pass saved can make a significant difference in the running time.

	3 Elements In Heap Array			Output	Next Element Read
	H[1]	H[2]	H[3]		
Run 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	End of Run.	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75*
	58	99	75*	58	end of tape
	99		75*	99	
			75*	End of Run.	Rebuild Heap
Run 3	75			75	

Figure: Example of run construction

As we have seen, it is possible for replacement selection to do no better than the standard algorithm. However, the input is frequently sorted or nearly sorted to start with, in which case replacement selection produces only a few very long runs. This kind of input is common for external sorts and makes replacement selection extremely valuable.

K-Way Merging

- The 2-way merge algorithm is almost identical to the merge procedure in figure.

- In general, if we started with m runs, then the merge tree would have $\lceil \log_2 m \rceil + 1$ levels for a total of $\lceil \log_2 m \rceil$ passes over the data file. The number of passes over the data can be reduced by using a higher order merge, i.e., k-way merge for $k \geq 2$. In this case we would simultaneously merge k runs together.

- The number of passes over the data is now 2, versus 4 passes in the case of a 2-way merge. In general, a k-way merge on m runs requires at most $\lceil \log_k m \rceil$ passes over the data. Thus, the input/output time may be reduced by using a higher order merge.

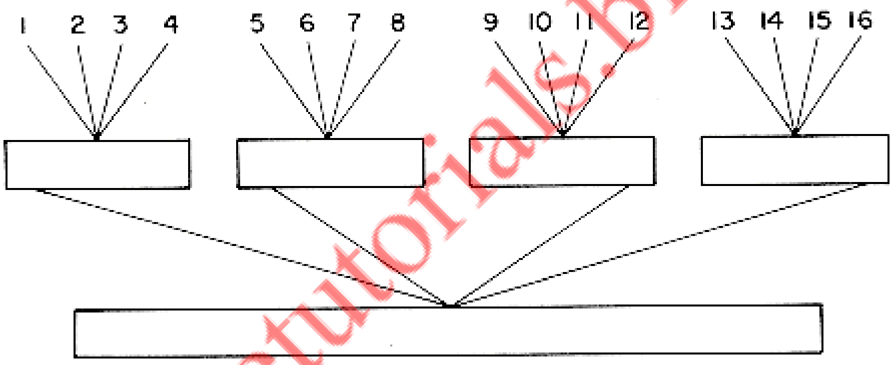


Figure: A 4-way Merge on 16 Runs

- The use of a higher order merge, has some other effects on the sort. To begin with, k-runs of size S_1, S_2, S_3, ...,S_k can no longer be merged internally in $O(\sum_1^k S_i)$ time.

- In a k-way merge, as in a 2-way merge, the next record to be output is the one with the smallest key. The smallest has now to be found from k possibilities and it could be the leading record in any of the k-runs.

  - The most direct way to merge k-runs would be to make k - 1 comparison to determine the next record to output. The computing time for this would be $O((k-1)\sum_1^k S_i)$. Since $\log_k m$ passes are being made, the total number of key comparisons being made is **$n(k - 1) \log_k m = n(k - 1) \log_2 m/\log_2 k$** where n is the number of records in the file. Hence, $(k-1)/\log_2 k$ is the factor by which the number of key comparisons increases. As k increases, the reduction in input/output time will be overweighed by the resulting increase in CPU time needed to perform the k-way merge.

- For large k (say, $k \geq 6$) we can achieve a significant reduction in the number of comparisons needed to find the next smallest element by using the idea of a selection tree. Hence, the total time needed per level of the merge tree is $O(n\log_2 k)$. Since the number of levels in this tree is $\boldsymbol{O(\log_k m)}$, the asymptotic internal processing time becomes $\boldsymbol{O(n \log_2 k \log_k m) = O(n \log_2 m)}$. The internal processing time is independent of k.

- In going to a higher order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_k m$ passes.

- This is so because the number of input buffers needed to carry out a k-way merges increases with k. Though $k + 1$ buffer are sufficient. Since the internal memory available is fixed and independent of k, the buffer size must be reduced as k increases. This in turn implies a reduction in the block size on disk. With the reduced block size each pass over the data results in a greater number of blocks being written or read.

- This represents a potential increase in input/output time from the increased contribution of seek and latency times involved in reading a block of data. Hence, beyond a certain k value the input/output time would actually increase despite the decrease in the number of passes being made. The optimal value for k clearly depends on disk parameters and the amount of internal memory available for buffers.

Buffer Handling for Parallel Operation

If k runs are being merged together by a k-way merge, then we clearly need at least k input buffers and one output buffer to carry out the merge. This, however, is not enough if input, output and internal merging are to be carried out in parallel. For instance, while the output buffer is being written out, internal merging has to be halted since there is no place to collect the merged records. This can be easily overcome through the use of two output buffers. While one is being written out, records are merged into the second. If buffer sizes are chosen correctly, then the time to output one buffer would be the same as the CPU time needed to fill the second buffer. With only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty and another block from the corresponding run is being read in. This input delay can also be avoided if we have $2k$ input buffers. These $2k$ input buffers have to be cleverly used in order to avoid reaching a situation in which processing has to be held up because of lack of input records from any one run. Simply assigning two buffers per run does not solve the problem. To see this, consider the following example.

Example 1: Assume that a two way merge is being carried out using four input buffers, $IN(i)$, $1 \leq i \leq 4$, and two output buffers, $OU(1)$ and $OU(2)$. Each buffer is capable of holding two records. The first few records of run 1 have key value 1, 3, 5, 7, 8, 9. The first few records of run 2 have key value 2, 4, 6, 15, 20, 25. Buffers $IN(1)$ and $IN(3)$ are assigned to run 1. The remaining two input buffers are assigned to run 2. We start the merging by reading in one buffer load from each of the two runs. At this time the buffers have the configuration of figure 8.12(a). Now runs 1 and 2 are merged using records from *IN* (1) and *IN*(2). In parallel with this the next buffer load from run 1 is input. If we assume that buffer lengths have been chosen such that the times to input, output and generate an output buffer are all the same then when *OU*(1) is full we have the situation of figure 8.12(b). Next, we simultaneously output *OU*(1), input into *IN*(4) from run 2 and merge into *OU*(2). When *OU*(2) is full we are in the situation of figure 8.12(c). Continuing in this way we reach the configuration of figure 8.12(e). We now begin to output *OU*(2), input from run 1 into *IN*(3) and merge into *OU*(1). During the merge, all records from run 1 get exhausted before *OU*(1) gets full. The generation of merged output must now be delayed until the inputting of another buffer load from run 1 is completed!

Example 1 makes it clear that if 2*k* input buffers are to suffice then we cannot assign two buffers per run. Instead, the buffers must be floating in the sense that an individual buffer may be assigned to any run depending upon need. In the buffer assignment strategy we shall describe, for each run there will at any time be, at least one input buffer containing records from that run. The remaining buffers will be filled on a priority basis. I.e., the run for which the *k*-way merging algorithm will run out of records first is the one from which the next buffer will be filled. One may easily predict which run's records will be exhausted first by simply comparing the keys of the last record read from each of the *k* runs. The smallest such key determines this run. We shall assume that in the case of equal keys the merge process first merges the record from the run with least index. This means that if the key of the last record read from run *i* is equal to the key of the last record read from run *j*, and $i < j$, then the records read from *i* will be exhausted before those from *j*. So, it is possible that at any one time we might have more than two bufferloads from a given run and only one partially full buffer from another run. All bufferloads from the same run are queued together. Before formally presenting the algorithm for buffer utilization, we make the following assumptions about the parallel processing capabilities of the computer system available:

(i) We have two disk drives and the input/output channel is such that it is possible simultaneously to read from one disk and write onto the other.

(ii) While data transmission is taking place between an input/output device and a block of memory, the CPU cannot make references to that same block of memory. Thus, it is not possible to start filling the front of an output buffer while it is being written out. If this were possible, then by coordinating the transmission and merging rate only one output buffer would be needed. By the time the first record for the new output block was determined, the first record of the previous output block would have been written out.

(iii) To simplify the discussion we assume that input and output buffers are to be the same size.

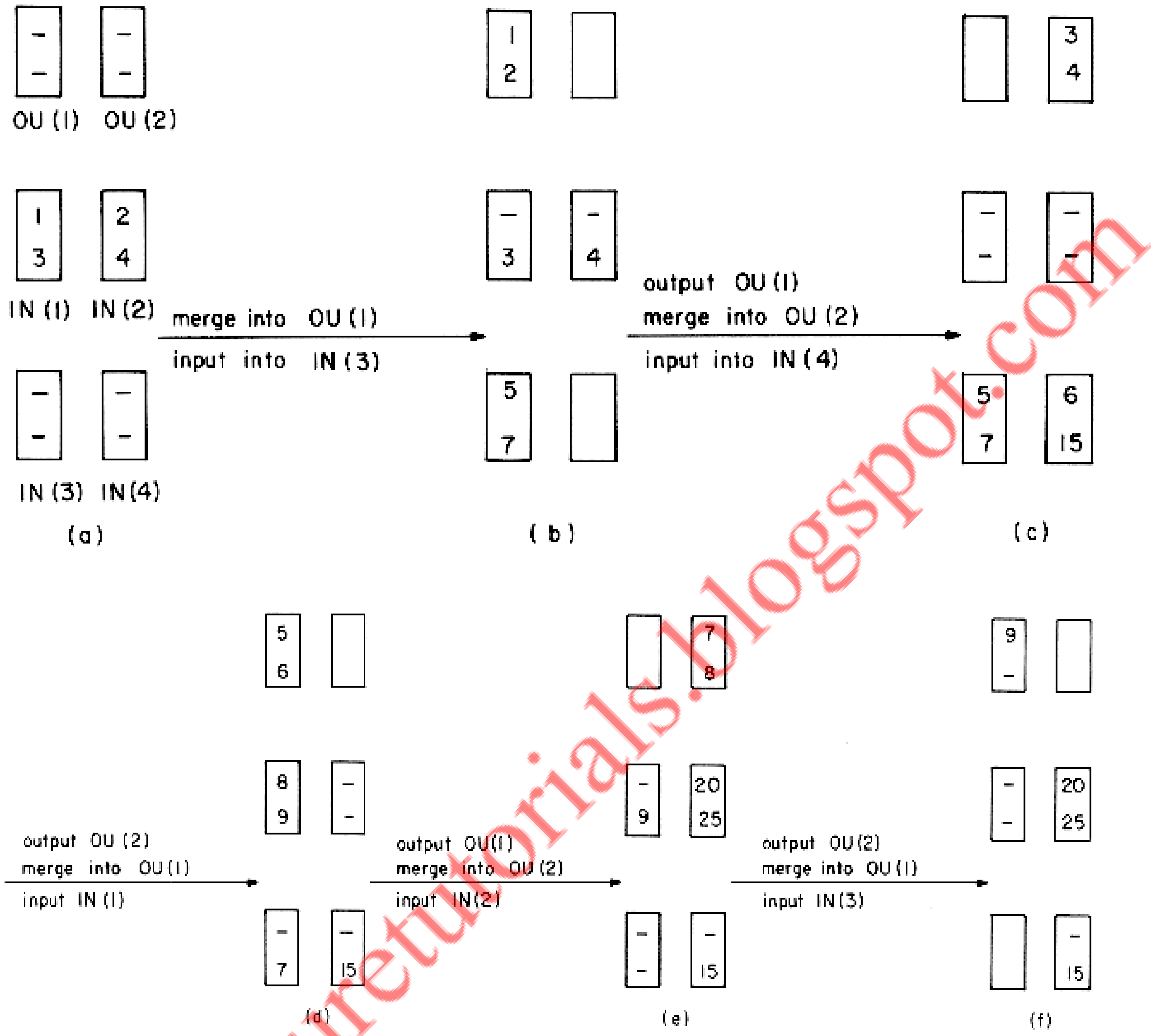


Figure 1 Example showing that two fixed buffers per run are not enough for continued parallel operation

Keeping these assumptions in mind, we first formally state the algorithm obtained using the strategy outlined earlier and then illustrate its working through an example. Our algorithm merges *k*-runs, k>=2, using a *k*-way merge. *2k* input buffers and 2 output buffers are used. Each buffer is a contiguous block of memory. Input buffers are queued in *k* queues, one queue for each run. It is assumed that each input/output buffer is long enough to hold one block of records. Empty buffers are stacked with AV pointing to the top buffer in this stack. The stack is a linked list. The following variables are made use of:

IN (i) ... input buffers, $1 \leq i \leq 2k$
OUT (i) ... output buffers, $0 \leq i \leq 1$
FRONT (i) ... pointer to first buffer in queue for run i, $1 \leq i \leq k$
END (i) ... end of queue for i-th run, $1 \leq i \leq k$

LINK (i) ... link field for i-th input buffer
in a queue or for buffer in stack $1 \le i \le 2k$
LAST (i) ... value of key of last record read
from run i, $1 \le i \le k$
OU ... buffer currently used for output.

The algorithm also assumes that the end of each run has a sentinel record with a very large key, say $+\infty$. If block lengths and hence buffer lengths are chosen such that the time to merge one output buffer load equals the time to read a block then almost all input, output and computation will be carried out in parallel. It is also assumed that in the case of equal keys the *k*-way merge algorithm first outputs the record from the run with smallest index.

```
procedure BUFFERING
for i <-- 1 to k do     //input a block from each run//
  input first block of run i into IN(i)
end
while input not complete do end     //wait//
for i <-- 1 to k do     //initialize queues and free buffers//
  FRONT(i) <-- END(i) <-- i
  LAST(i) <-- last key in buffer IN(i)
  LINK(k + i) <-- k + i + 1     //stack free buffer//
end
LINK(2k)  <-- 0; AV <-- k + 1; OU <-- 0
//first queue exhausted is the one whose last key read is smallest//
find j such that LAST(j) = $\min_{1 \le i \le k}$ {LAST(i)}
l <-- AV; AV <-- LINK(AV)     //get next free buffer//
if LAST(j) $\neq +\infty$ then [begin to read next block for run j into
                    buffer IN(l)]
repeat     //KWAYMERGE merges records from the k buffers
           FRONT(i) into output buffer OU until it is full.
           If an input buffer becomes empty before OU is filled, the
           next buffer in the queue for this run is used and the empty
           buffer is stacked or last key = $+\infty$//
  call KWAYMERGE
  while input/output not complete do     //wait loop//
  end
 if LAST(j) $\neq +\infty$    then
  [LINK(END(j)) <-- l; END(j) <-- l; LAST(j) <-- last key read
                              //queue new block//
  find j such that LAST(j) = $\min_{1 \le i \le k}$ {LAST(i)}
  l <-- AV; AV <-- LINK(AV)]     //get next free buffer//
  last-key-merged <-- last key in OUT(OU)
  if LAST(j) $\neq +\infty$ then [begin to write OUT(OU) and read next block of
                    run j into IN(l)]
                 else [begin to write OUT(OU)]
  OU <-- 1 - OU
```

```
    until last-key-merged = + ∞
    while output incomplete do      //wait loop//
    end
  end BUFFERING
```

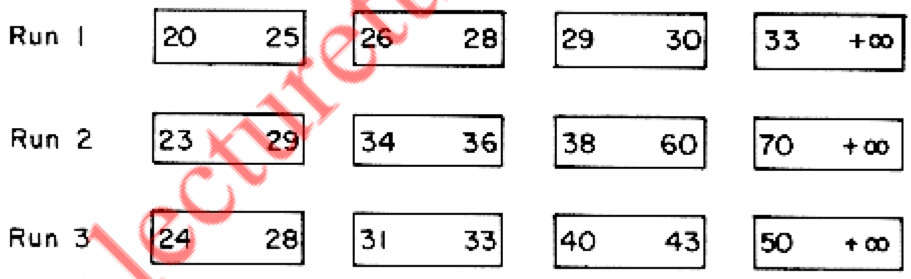
Notes: 1) For large *k*, determination of the queue that will exhaust first can be made in $\log_2 k$ comparisons by setting up a selection tree for LAST(*i*), $1 \le i \le k$, rather than making $k - 1$ comparisons each time a buffer load is to be read in. The change in computing time will not be significant, since this queue selection represents only a very small fraction of the total time taken by the algorithm.

2) For large *k* the algorithm KWAYMERGE uses a selection tree

3) All input/output except for the initial *k* blocks that are read and the last block output is done concurrently with computing. Since after *k* runs have been merged we would probably begin to merge another set of *k* runs, the input for the next set can commence during the final merge stages of the present set of runs. I.e., when LAST(*j*) = +∞ we begin reading one by one the first blocks from each of the next set of *k* runs to be merged. In this case, over the entire sorting of a file, the only time that is not overlapped with the internal merging time is the time for the first *k* blocks of input and that for the last block of output.

4) The algorithm assumes that all blocks are of the same length. This may require inserting a few dummy records into the last block of each run following the sentinel record +∞.

Example 2: To illustrate the working of the above algorithm, let us trace through it while it performs a three-way merge on the following three runs:



Run 1	20 25	26 28	29 30	33 +∞
Run 2	23 29	34 36	38 60	70 +∞
Run 3	24 28	31 33	40 43	50 +∞

Each run consists of four blocks of two records each; the last key in the fourth block of each of these three runs is +∞. We have six input buffers IN(*i*), $1 \le i \le 6$, and 2 output buffers OUT(0) and OUT(1). The status of the input buffer queues, the run from which the next block is being read and the output buffer being ouput at the beginning of each iteration of the **repeat-until** of the buffering algorithm.

Theorem : The following is true for algorithm BUFFERING:

(i) There is always a buffer available in which to begin reading the next block; and

(ii) during the k-way merge the next block in the queue has been read in by the time it is needed.

Proof: (i) Each time we get to line 20 of the algorithm there are at most $k + 1$ buffer loads in memory, one of these being in an output buffer. For each queue there can be at most one buffer that is partially full. If no buffer is available for the next read, then the remaining k buffers must be full. This means that all the k partially full buffers are empty (as otherwise there will be more than $k + 1$ buffer loads in memory). From the way the merge is set up, only one buffer can be both unavailable and empty. This may happen only if the output buffer gets full exactly when one input buffer becomes empty. But $k > 1$ contradicts this. So, there is always at least one buffer available when line 20 is being executed.

(ii) Assume this is false. Let run R_i be the one whose queue becomes empty during the KWAYMERGE. We may assume that the last key merged was not the sentinel key $+\infty$ since otherwise KWAYMERGE would terminate the search rather then get another buffer for R_i. This means that there are more blocks of records for run R_i on the input file and LAST(i) $\neq +\infty$. Consequently, up to this time whenever a block was output another was simultaneously read in (see line 22). Input/output therefore proceeded at the same rate and the number of available blocks of data is always k. An additional block is being read in but it does not get queued until line 18. Since the queue for R_i has become empty first, the selection rule for the next run to read from ensures that there is at most one block of records for each of the remaining $k - 1$ runs. Furthermore, the output buffer cannot be full at this time as this condition is tested for before the input buffer empty condition. Thus there are fewer than k blocks of data in memory. This contradicts our earlier assertion that there must be exactly k such blocks of data.

Run Generation

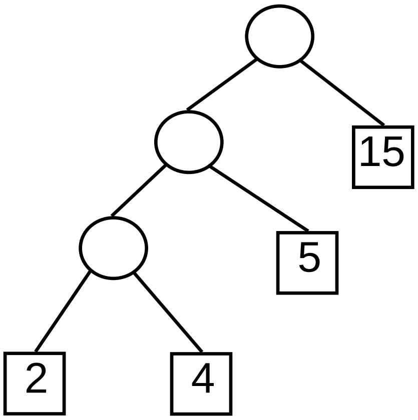
Using conventional internal sorting methods, it is possible to generate runs that are only as large as the number of records that can be held in internal memory at one time. Using a tree of losers it is possible to do better than this. In fact, the algorithm we shall present will on the average generate runs that are twice as long as obtainable by conventional methods. This algorithm was devised by Walters, Painter and Zalk. In addition to being capable of generating longer runs, this algorithm will allow for parallel input, output and internal processing. For almost all the internal sort methods discussed in Chapter 7, this parallelism is not possible. Heap sort is an exception to this. In describing the run generation algorithm, we shall not dwell too much upon the input/output buffering needed. It will be assumed that input/output buffers have been appropriately set up for maximum overlapping of input, output and internal processing. Wherever in the run generation algorithm there is an input/output instruction, it will be assumed that the operation takes place through the input/output buffers. We shall assume that there is enough space to construct a tree of losers for k records, $R(i)$, $0 \leq i < k$. This will require a loser tree with k nodes numbered 0 to $k - 1$. Each node, i, in this tree will have one field $L(i)$. $L(i)$, $1 \leq i < k$ represents the loser of the tournament played at node i. Node 0 represents the overall winner of the tournament. This node will not be explicitly present in the algorithm. Each of the k record positions $R(i)$, has a run number field $RN(i)$, $0 \leq i < k$ associated with it. This field will enable us to determine whether or not $R\ (i)$ can be output as part of the run currently being generated. Whenever the tournament winner is output, a new record (if there is one) is input and the tournament replayed as discussed in section Algorithm RUNS is simply an implementation of the loser tree strategy discussed earlier. The variables used in this algorithm have the following significance:

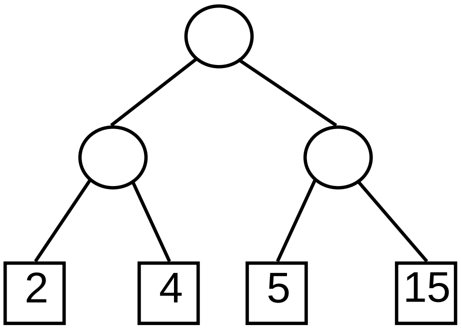
R(i), $0 \leq i < k$ ... the k records in the tournament tree
KEY(i), $0 \leq i < k$ ... key value of record R(i)
L(i), $0 < i < k$ ... loser of the tournament played at node i
RN(i), 0 <= i < k ... the run number to which R(i) belongs
RC ... run number of current run
Q ... overall tournament winner
RQ ... run number for R(Q)
RMAX ... number of runs that will be generated
LAST__KEY ... key value of last record output

The loop of lines 5-25 repeatedly plays the tournament outputting records. The only interesting thing about this algorithm is the way in which the tree of losers is initialized. This is done in lines 1-4 by setting up a fictitious run numbered 0. Thus, we have $RN(i) = 0$ for each of the k records $R(i)$. Since all but one of the records must be a loser exactly once, the initialization of $L(i) \leftarrow i$ sets up a loser tree with R(0) the winner. With this initialization the loop of lines 5-26 correctly sets up the loser tree for run 1. The test of line 10 suppresses the output of these k fictitious records making up run 0. The variable LAST__KEY is made use of in line 13 to determine whether or not the new record input, R(Q), can be output as part of the current run. If KEY(Q) < LAST__KEY then $R(Q)$ cannot be output as part of the current run *RC*as a record with larger key value has already been output in this run. When the tree is being readjusted (lines 18-24), a record with lower run number wins over one with a higher run number. When run numbers are equal, the record with lower key value wins. This ensures that records come out of the tree in non-decreasing order of their run numbers. Within the same run, records come out of the tree in non-decreasing order of their key values . RMAX is used to terminate the algorithm. In line 11, when we run out of input, a record with run number RMAX + 1 is introduced. When this record is ready for output, the algorithm terminates in line 8. One may readily verify that when the input file is already sorted, only one run is generated. On the average, the run size is almost $2k$. The time required to generate all the runs for an n run file is $O(n \log k)$ as it takes $O(\log k)$ time to adjust the loser tree each time a record is output. The algorithm may be speeded slightly by explicitly initializing the loser tree using the first k records of the input file rather than k fictitious records as in lines 1-4. In this case the conditional of line 10 may be removed as there will no longer be a need to suppress output of certain records.

Optimal Merging Of Runs

- When we merge by using the first merge tree, we merge some records only once, while others may be merged up to three times.
- In the second merge tree, we merge each record exactly twice.
- We can construct Huffman tree to solve this problem.





External path length:

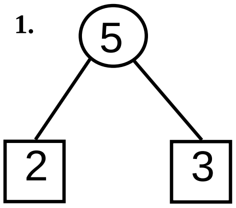
2*3+4*3+5*2+15*1=43

2*2+4*2+5*2+15*2=52

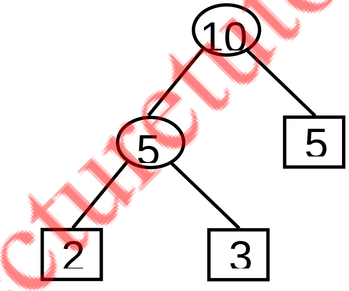
Construction of a Huffman tree

Runs of length : 2,3,5,7,9,13

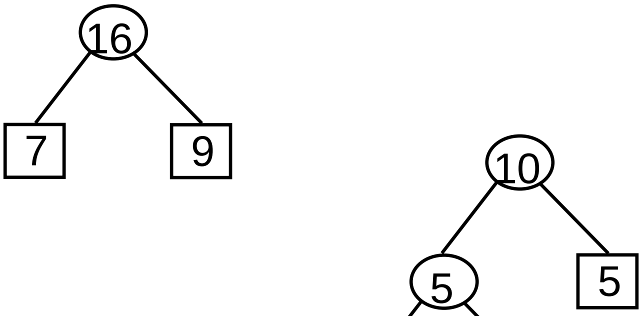
1.



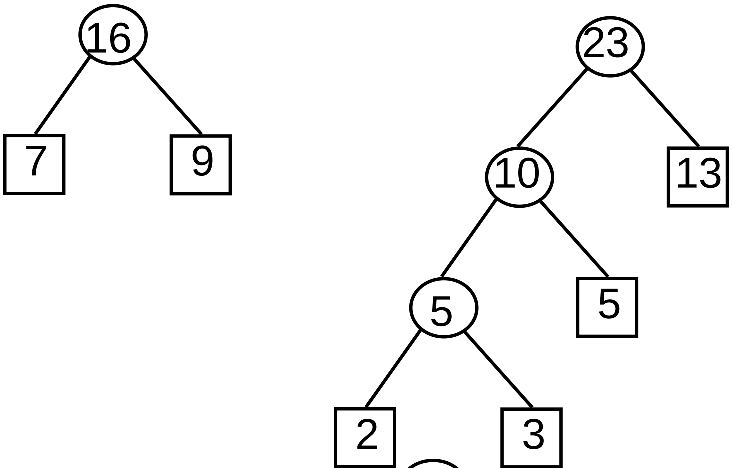
2.



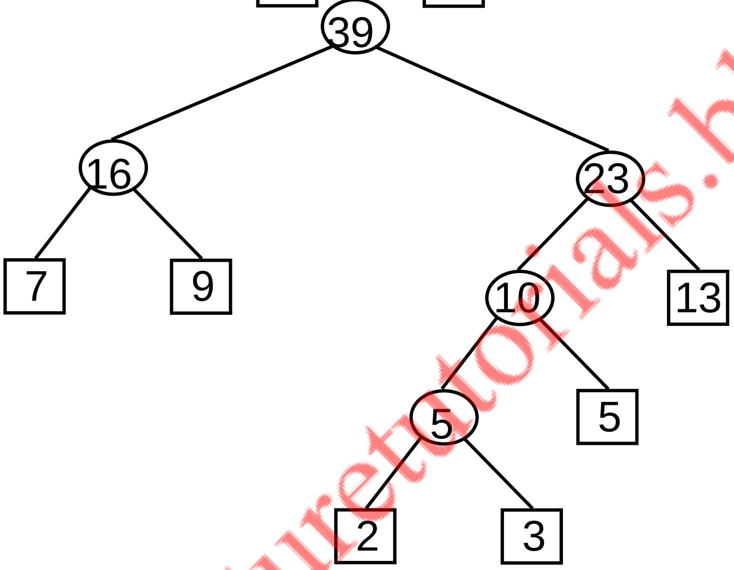
3.



4.



5.


39
16
23
7
9
10
13
5
5
2
3