UNIT 3 SORTING TECHNIQUE - II

Structure

3.0 Introduction
3.1 Objectives
3.2 Data Storage
3.3 Sorting with Disk
3.4 Buffering
3.5 Sorting with Tapes
3.6 Summary
Review Questions
Suggested Readings

3.0 INTRODUCTION

In the previous unit, we were introduced to the importance of sorting and discussed many internal sorting methods. We will now talk about external sorting. These are methods employed to sort records of files which are too large to fit in the main memory of the computer. These methods involve as much external processing as processing in the CPU.

To study external sorting, we need to study the various external devices used for storage in addition to sorting algorithms. The involvement of external device make sorting algorithms complex because of the following reasons:
The cost of accessing an item is much higher than any computational costs.

Depending upon the external device, the method of access has different restrictions.

The variety of external storage device types changes depending upon the latest available technology. Therefore, external sorting methods are dependant on external factors also. External sorting methods should have equal emphasis on the systems aspect as well as on the algorithms aspect.

In this unit, we will just be introduced to some data storage devices and then study sorting algorithms for data stored on different devices.

3.1 OBJECTIVES

After going through this you will be able to:

Differentiate between internal sorting and external sorting

Discuss sorting with tapes

Sorting with disks

3.2 DATA STORAGE

External storage devices can be categorised into two types based on the access method. These are sequential access devices (e.g. magnetic tapes) and rando access devices (e.g. disks). We know from the

unit on files (Block 5, Unit the meaning of sequential access and random access. We will just see the characteristics magnetic tapes and then of disks.

3.2.1 Magnetic Tapes

Magnetic tape devices for computer input/output are similar in principle to audio tape recorders. Magnetic tape is wound on a spool. Tracks run across the length of the tape. Usuall there are 7 to 9 tracks across the tape width Data is recorded on the tape in a sequence of bits. The number that can be written per inch of track is called the tape density - measured in bits per inch.

Information on tapes is usually grouped into blocks, which may be of fixed or variable size. Blocks are separated by an inter- block gap. Because request to read or write blocks do not arrive at a tape drive at constant rate, there must be a gap between each pair of blocks forming a space to be passed over as the tape accelerates to read/write speed. The medium is not strong enough to withstand the stress that it would sustain with instantaneous starts and stops. Because the tape is not moving at a constant speed, a gap can not contain user data. Data is usually read/written from tapes in terms of blocks. This is shown in figure 1 .

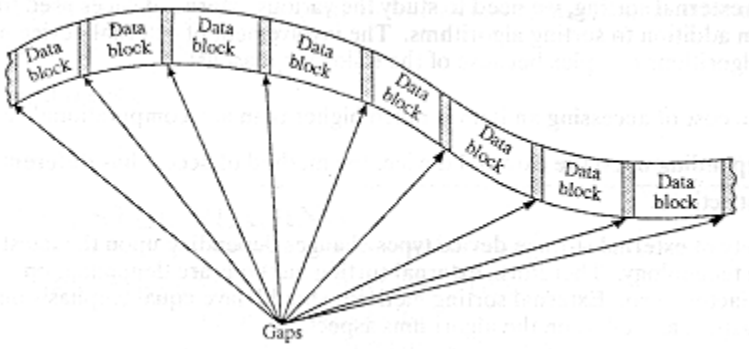


Figure 1: Interblock gaps

In order to read or write data to a tape the block length and the address in memory to/from which the data is to be transferred must be specified. These areas in memory from/to which data is transferred will be called buffers. Usually block size will respond to buffer size.

Block size is a crucial facator in tape access. A large block size is preferred because of the following reasons:

1) Consider a tape with a tape density of 600 bpi. and an inter block gap of 3/4". this gap is enough to write 450 characters. With a small block size, the number of blocks per tape length will increase. This means a larger number of inter block gaps, i.e. bits of data which cannot be utilised for data storage, and thus a decreased tape utilisation. Thus the larger the block size, fewer the number of blocks, fewer the number of inter block gaps and better the tape utilisation.

2) Larger block size reduces the input/output time. The delay time in tape access is the time needed to cross the inter block gap. This delay time is larger when a tape starts from rest than when the tape is already moving. With a small block size the number of halts in a read are considerable causing the delay time to be incurred each time.

While large block size is desirable from the view Of efficient tape usuage as well as reduced access time, the amunt of main memory available for use 1/0 buffers is a limiting factor.

3.2.2 Disks

Disks are an example of direct access storage devices. In contrast to the way information in recorded on a gramophone record, data are recorded on disk platter in concentric tracks. A disk has two surfaces on which data can be recorded. Disk packs have several such disks or platters rigidly mounted on a common spinder. Data is read/written to the disk by a read/write head. A disk pack would have one such head per surface.

Each disk surface has a number of concentric circles called tracks. In a disk pack, the set of parallel tracks on each surface is called a cylinder. Tracks are further divided into sectors. A sector is the smallest adressable segment of a track.

Data is stored along the tracks in blocks. Therefore to access a disk, the track or cylinder number and the sector number of the starting block must be specified. For disk packs, the surface must also be specified. The read/write head moves horizontally to position itself over the correct track for accessing disk data

This introduces three time components into disk access.

1. Seek time:- The time taken to position the read/write head over the correct cylinder.

2. Latency time:- The time taken to position the correct sector under head.

3. Transfer time:- The time taken to actually transfer the block between Main memory and the disk.

Having seen the structure of data storage on disks and tapes and the methods of accessing them, we now turn to specific cases of external sorting. Sorting data on disks and sorting data on apes. The general methods for external sorting is the merge sort.

In this segments of the file are sorted using a good internal sort method. These sorted segments, called runs, are written out onto the device. Then all the generated runs are merged into one. run.

3.3 SORTING WITH DISKS

We will first illustrate merge sort using disks and then analyse it as an external sorting method.

Example

The file F containing 600 records is to be sorted. The main memory is capable of sorting of 1000 records at a time. The input file F is stored on one disk and we have in addition another scratch disk. The block length of the input file is 500 records.

We see that the file could be treated as 6 sets of 1000 recordes each. Each set is sorted and stored on the scratch disk as a 'run'. These 5 runs will then be merged as follows:

Allocate 3 blocks of memory each capable of holding 500 records. Two of these buffers B1 and B2 will be treated as input buffers and the third B3 as the output buffer. We have now the following.

1. 6 runs Rl,R2,R3,R4,R5,R6 on the scratch disk.

2. 3 buffers Bl,B2 and B3.

- Read 500 records from R1 into B1.

- Read 500 records from R2 into B2.

- Merge B1 and B2 and write into B3.

- When B3 is full - write it out to the disk as run R11.

- Similarly merge R3 and R4 to get run R12.
- Merge R5 and R6 to get run R13.

Thus, from 6 runs of size 1000 each, we have now 3 runs of size 2000 each.

The steps are repeated for steps R11 and R12 to get a run of size 4000.

This run is merged with R13 to get a single sorted run of size 6000.

Pictorically, this can be represented as:

Input file F with 6000 records

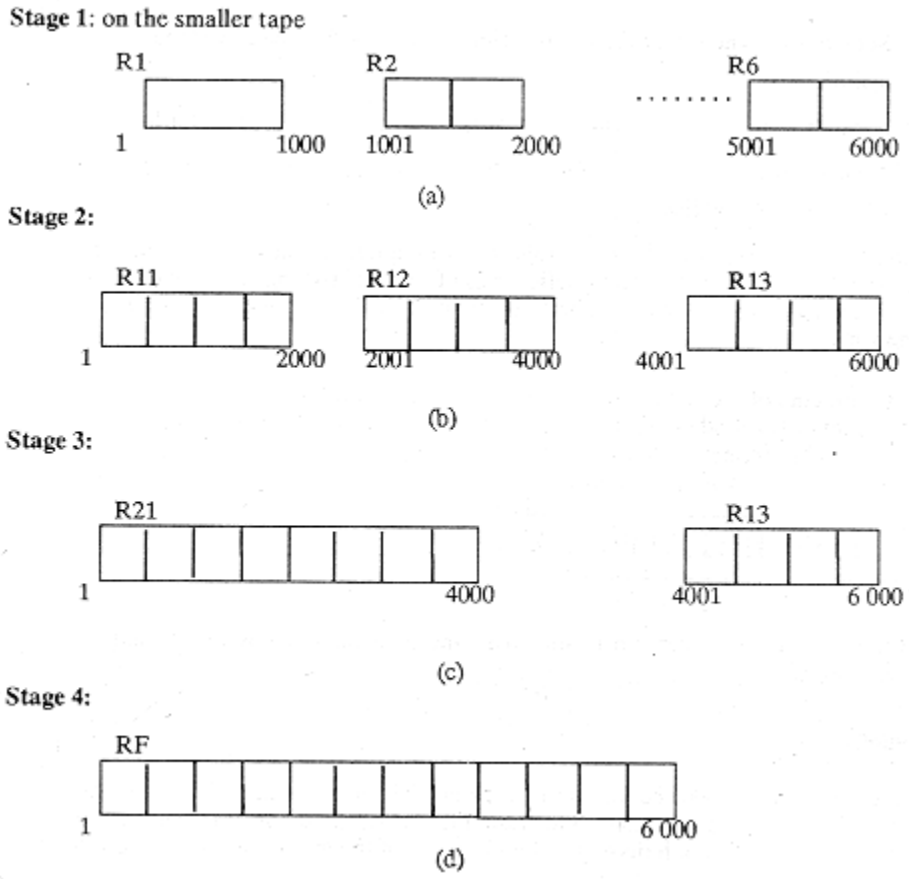


Figure 2: Merge Sort

The divisions in each run indicate the number of blocks.

Analysis

T1 - Seek time
T2 - Latency time
T3 - Transmission time for 1 block of 5000 records
T - T1 + T2 + T3

T4 - Time to internally sort 1000 records
nTM - Time to merge n records from input buffers to the output buffer.

In stage 1	we read 6000/500	= 12 blocks
	internally sort 6000/1000	= 6 sets of 1000 records
	write 6000/500	= 12 blocks
Therefore, time taken in stage 1		= 24T + 6T4
In stage 2	we read 12 blocks	
	write 12 blocks	

Merge 5 x 2000 = 6000 records
Time taken in stage 2 = 24T + 600OTM

In stage 3 we merge only 2 runs

Therefore we read 8 blocks

write 8 blocks
merge 2 x 2000 = 4000 records

Time taken in stage 3 = 16T + 400OTM

In stage 4 we read 12 blocks

write 12 blocks

merge 4000 + 2000 = 4000 records

Therefore time taken in stage 4 = 24T + 600OTM

Total time taken is = 24T + 6T4 + 24T 6000TM + 16T + 4000TM + 24T + 6000Tm
= 88T + 6T4 + 1000Tm

It is seen that the largest influencing factor is TM, which depends on the number of passes made over the data or the number of times runs must he combined.

We have assumed a uniform seek and latency time for all blocks for simplicity of analysis. This may not be true in real life situations.

Time could also be reduced by exploiting the parallel features available, i.e. input/output and CPU processing carried out at the same time.

We will now focus on method to optimise the effects of the above factors, i.e. to say we will be carefully concerned with buffering and block size, assuming the internal sorting algorithms and the seek and latency time factors are the best possible.

1.K-way merging

In the above example, we used 2 way merging, i.e. combinations of two runs at a time. The number of passes over the data can be reduced by combining more runs at a time, hence the K-way merge where K2. In the game example, suppose we had used a 3 way merge then

- at stage 2 we would have had 2 runs of size 3000 each

- at stage 3 we would have had a single sorted run of size 6000

This would have affected our analysis as follows:

Stage 1 = 24T + 6T4
Stage 2 = 24T + 600OTM
Stage 3 = 24T + 600OTM
Total = 74T + 6T4 + 1200Otm

There is obviously an enoromous dip in the contribution of TM to the total time.

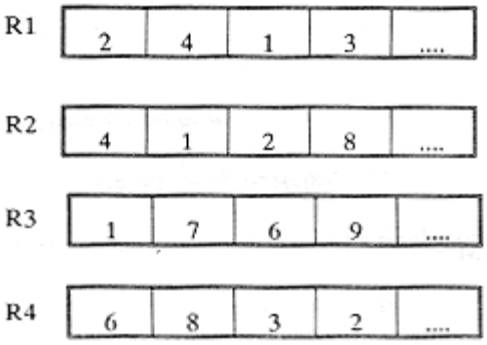
This advantage is accompanied by certain side effects. The time merge K runs , Would obviously be more than the time to merge 2 runs. Here the value of TM itself could out of K in each step of the merge is needed. One such method is the use of a selection tree.

A selection tree is a binary tree in which each node represents the smaller of its children. It therefore follows that the root will be the smallest node. The way it makes is simple, Initially, the selection tree is built from the lst item of each of the K runs. The one which gets to the root is selected as the smallest. Then the next item in the run from which the root was selected enters the tree and the tree gets restructured to get a new root and so on till the K runs are merged.

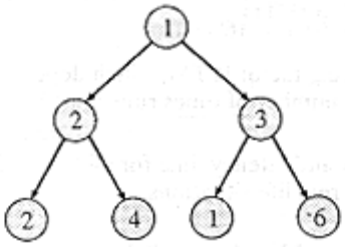
Example

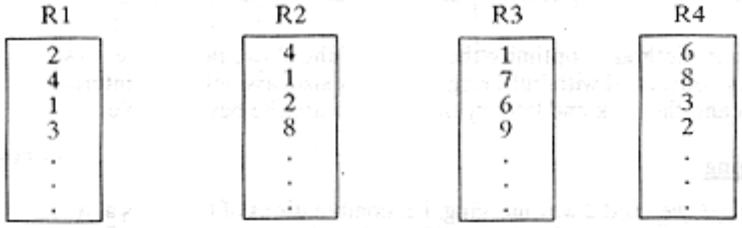
Consider a selection tree for the following 4 runs:

Consider a selection tree for the following 4 runs:



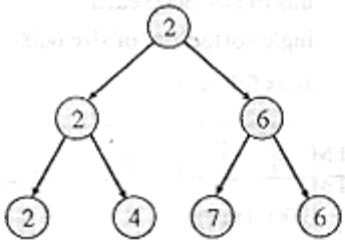
1. Construction

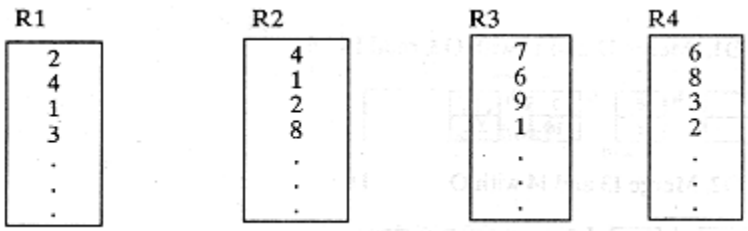




(a)

Node 1 was selected as the minimum. This is removed from the tree. Node belonged to R3, therefore the next item from R3 . 7 enters the tree.





(b)

Figure 3: K-way Merging

The new root is 2. This came from R1 in the next step, the next item from Rl will enter the tree and so on. In going into a higher order merge, we save on input/output time. But with the increase in the order of the merge the number of buffers required also increases at the minimum to merge K runs we need K + 1 buffers. Now internal memory is a fixed entity, therefore if the number of buffers increases, there size must decrease. This implies a smaller block size on the disk and hence larger number of input/outputs. So beyond an optional value of K, we find that the advantage of reduced number of passes is offset by the disadvantage of increased input/output.

3.4 BUFFERING

We saw that the second factor exploit in external sorting is parallelism. In the K-way merge discussion, we stated that K+ 1 buffers are enough to merge K-runs using K buffers for input and 1 for output. But this number of buffers is not adequate to exploit parallel operation in a computer.

When the full output buffer is being written onto the disk, no internal merging activity takes place, because there is no place to write the results of the merge to.

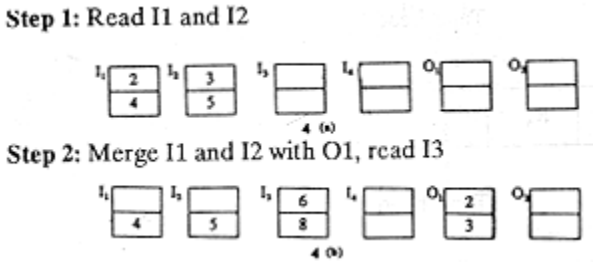
This problem can be overcome by having 2 output buffers, so that one can be filled while the other one is being written.

Now consider the input buffers. We have assigned one buffer per run assuming the buffer corresponding to one run as emptied, then again merging activity ceases till the input/output to fetch the next block from the run is complete. We will show by example that simply assigning 2 input buffers per run will not solve this problem.

Example Consider 2 runs with the following data.

R1 2,4,6,8,9,10
R2 3,5,7,16,21,26

Input buffer I1 and I2 are allotted to R1. Input buffers 12 and 14 are allotted to R2. The number of output buffers are 01 and 02. We assume a timing situation whereby it is possible to simultaneously write an output buffer, merge 2 runs and read an input buffer. The merging scenario can be represented as follows (figure 4):



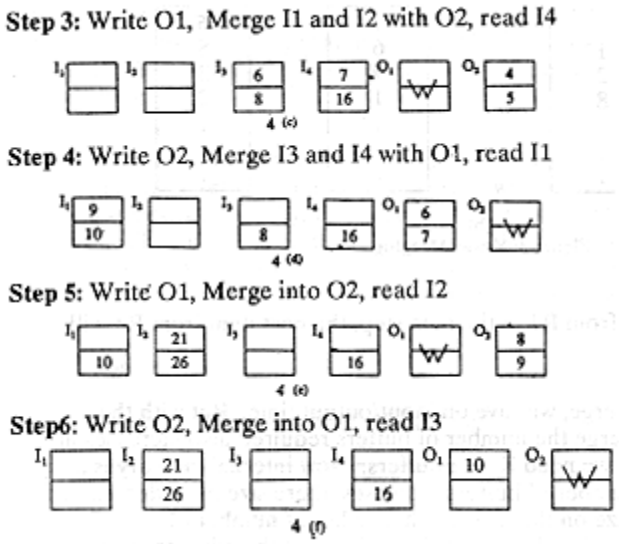


Figure 4

We have arrived at a situation where I1 and I3 have been exhausted at O1 is not yet full. Therefore, merge will be halted ti1ll step37.

A better way of using these 2K buffers is now described. We will continue to have 1 dedicated buffer for each run. The remaining K buffers are allocated to runs on a priority basis, i.e. the run for which the merge will run out of records is the next one filled.

3.5 SORTING WITH TAPES

Sorting with tapes is essentially similar to the merge sort used for sorting with disks. The differences arise due to the sequential access restriction of tapes. This makes the selection time prior to data transmission an important factor, unlike seek time and latency time. Thus in sorting with tapes we will be more concerned with arrangement of blocks and runs on the tape so as to reduce the selection or access time.

As in 6.3.4 we will use an example to determine the factors involved.

Example: A file of 6000 records is to be sorted. It is stored on a tape and the block length is 500. The main memory can sort upto a 1000 records at a time. We have in addition 4 search tapes T1 -T4 The steps in merging can be summarized as follows (figure 5):

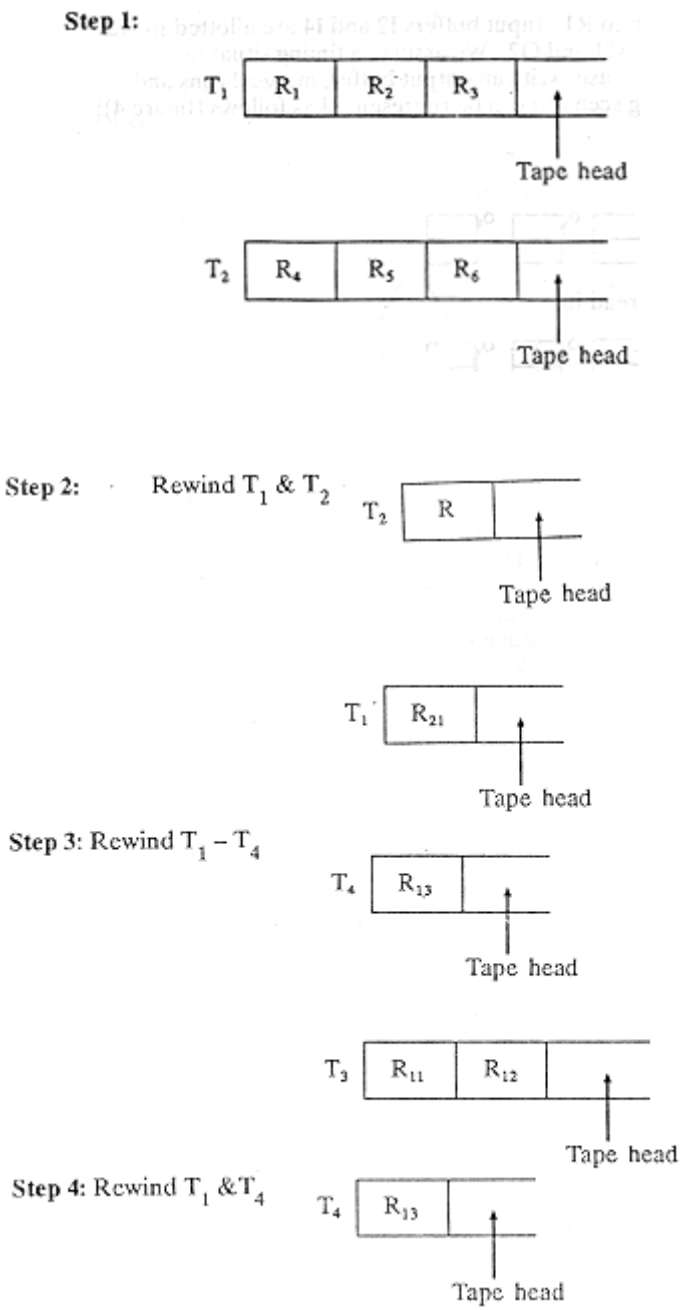


Figure 5: Sorting with Tapes

Analysis

t_{is} = time taken to internally sort 750 records.
t_{rw} = time to read or write. One block of 250 records. Onto tape starting from present position of tape read/write head.
t_{rew} = time to rewind tape over a length corresponding to one block.
nt_m = time to merge n records from input buffers to output buffers using a 2-way merge.
A = delay caused by having to wait for T_4 to be mounted in case we are ready to use Tn before it is mounted.

Total time = $6t_{is} + 132t_{rw} + 12000t_m + 51t_{rew} + A$

The above computing time analysis assumes that no operations are carried out in parallel. The analysis could be carried further or in the case of disks to show the dependence of sort time on the number of passes made on the data.

Balanced Merge Sorts:

We see that the computing time in the above example depends as, in the case of disk sorting, essentially on the number of passes being made over the data. Use of a higher order merge results in a decrease in the number of passes being made without significantly changing the internal merge time. Hence we would like to use as high an order merge as possible. In disk sorting, the order of merge was limited essentially by the amount of main memory available for input/output buffers. A k-way merge required the use of 2k + 2 buffers. Another more severe restriction on the merge order in the case of tapes is the number of tapes available. In order to avoid excessive seek time, it is necessary that runs being merged be on different tapes. Thus a k-way merge requires at least k-tapes for use as input tapes during the merge.

In addition, another tape is required for the output generated during the merge. Hence at least k + 1 tapes must be available for a k-way tape merge. Using k + 1 tapes for a k-way merge requires an additional pass over the output tape to redistribute the runs onto k-tapes for the next level of merge. This redistribution pass can be avoided by using 2k tapes. During the k-way merge, k of these tapes are used as input tapes and the remaining k as output tapes. At the next merge level, the role of input-output tapes is interchanged. These two approaches are now examined. Algorithms x1 and x2 perform a k-way merge with the k + 1 taoes , strategy while algorithm x3 performs a k-way merge using the 2k tapes strategy

```
        Procedure x1;

        [sort a file of records from a given input tape using a k-way
        merge given tapes t1, ... t k+1 are available for the sort.]
        label 200
        begin
        Create runs from the input tape distributing them evenly over tapes t1,..tk
        Rewind t1...tk and also at the input tape;
        If there is only one run goto 200; [the sorted file is on t1].
        replace input tape by tk+1;
while true do
        [repeatedly merge onto tk+1 and redistribute. back onto t1, .... tk].
begin
        merge runs from t1. ... t k+1 onto tk+1;
        rewind t1,.....tk+1;
        If number of + runs on tk+1 = 1 then goto 200;
        [output on tk+1]
        evenly distribute from tk+1 onto t1,---,tk;
        rewind t1 ... tk+1;
        end;
200 end; [of x1]
```

Analysis: To simplify the analysis we assume that the number of runs generated(m) is a power of k. One pass over the entire file includes both reading and writing. The number of passes in the while loop is log k^m merge passes and log k^{m-1} redistribution. passes . . the total number of passes is 21og k^m. If the time to rewind the entire input tape is trew, then the non-overlapping rewind time is approximately 21ogk$^{m.trew}$

A cleverer way to tackle the problem is to rotate the output tape, i.e. tapes are used as output tapes in the cyclic order, K+ 1, 1, 2,..., k. This makes the redistribution from the output tape make less than a full pass over the file. Algorithm x2 describes the process.

Procedure x_2

[Same definitions as for x1]

label 200

begin

Create runs from the input file distributing them evenly over tapes t_1 .... t_k ;

Rewind t_1,..,t_k and also the input tape;
If there is only one run then goto 200;
[sorted file on t_1]
replace input tape by t_{k+1};
i = k + 1 ; i is index of the output tape]
while true do

begin
merge runs from the k tapes t_j;
I < =j < = k + 1 and i onto t_i;
rewind t_1 ...,t_{k+1};
If number of runs on t_i = 1 then
goto 200 [output on t_1]
evenly distribute (k-1)1k of the runs on ti onto
tape t_j, k = j < = k + 1 and j i and j i mod (k + 1) + 1;
rewind tapes tj, 1 < = j < = k + 1 and j i;
i: = i mod (k+ 1) + 1;
end;

200 end; [end of x2]

Analysis: The only difference between algorithms x1 and x2 is in the redistributing time. m is the number of runs generated in line 5. Redistributing is done log k^{m-1} times, but each such pass reads and writes only (k-1)/k of the file. Hence the effective number of passes made over the data is (2-1/k) log k^m + 1/k. for two-way merging on 3 tapes algorithm x^2 will make 3/2 log k^m + 1/2 passes while x1 will make 21ogk^m passes. If trew is the rewind time then the non-overlapping rewind time for x^2 is utmost (1 + 1/k)(log k^m) trew + (1-1/k) trew. Instead of distributing runs as shown we could write the first m/k runs on one tape, begin rewind, write the next m/k runs on the second tape, begin rewind etc. In this case we can begin filling input buffers for the next merge level while some of the tapes are still rewinding.

In case a k-way merge is using 2k tapes, no redistribution is needed and so the number of passes made is only logk^{m+1}. This implies that if 2k + 1 tapes are available than a 2k way merge will make (2-1/2k)log2k^m + 1/(2k) passes while a k-way merge utilizing 2k tapes will make only logk^{m+1} passes. The table compares the number of passes being made in the two methods for some value of k.

k	2k-way	k-way
1.	3/2 $\log_2$m + 1/2	-
2.	7/8 $\log_2$m + 1/4	$\log_2$M + 1
3.	1.124 $\log_3$m + 1/6	$\log_3$m + 1
4.	1.25 $\log_4$m + 1/8	$\log_4$m + 1

As is evident from the table, for k 2 a k-way merge using 2k tapes is better than a 2k-way merge using 2k + 1 tapes.
Algorithm x3 makes a k-way merge sort using 2k tapes.

```
Procedure x3;

[sort a file of records from a given input tape using a k-way
merge on 2k tapes]

begin

        Create runs from the input file distributing them evenly over tapes t1 .. tk;
        rewind t1 ... tk; rewind the input tape;
        replace the input tape by tape t2k; i: = 0;
        while total number of runs tik+1 ... t1k+k 1 do
        begin
                j: = 1-i;
                perform a k-way merge from tik+1,... tik+k evenly
                distributing output runs onto tjk+1,....tjk+k;
                rewind t1 .. t2k;
                i: = j ; [switch input and output tapes]
        end;
        [sorted file is on tik+1]

end; [end of x3]
```

Analysis: To simplify the analysis, assume that m is a power of k. In addition to the initial run creation pass, the algorithm makes $\log_k m$ merge passes. Let trew be the time to rewind the entire input file. If m is a power of k then the time to rewind tapes $t_1,...t_{2k}$ in the while loop takes $t_{rew/k}$. for each but the last loop iteration. The last rewind takes time t_{rew} The total rewind time is therefore bound by $(2 + (\log_k m - 1)/k)t_{rew}$.

It should be noted that all the above algorithms use the buffering strategy developed in the k-way merge. The proper choice of buffer lengths and the merge order (restricted by the number of tapes available) would result in an almost complete overlap of internal processing with input/output time. At the end of each level of merge, processing will have to wait till the tapes rewind. This wait can be minimized using the run distribution strategy developed in algorithm x2.

Polyphase merging: The problem with balanced multiway merging is that it requires either an excessive number of tape units or excessive copying polyphase merging is a method to eliminate virtually all the copying by changing the way in which the small sorted blocks are merged together. The basic idea is to distribute the sorted blocks produced by replacement selection somewhat unevenly among the available tape units (learning one empty) and then to apply a 'merge = until-empty' strategy at which point one of the input tapes and the output tape switch roles.

For example suppose that we have just 3 tapes, and we start with the initial configuration of sorted blocks on the tapes as shown at the top of the figure. Tape 3is initially empty the output tape for the first merges.

```
Tape  1 B P S T U-J O - B H O -E F N S - H J O.
      2 FHY - BNQ - FM.
      3 .
Tape  1 EFNS.HJO.
      2 .
      3 B F H P S T U Y-B J N O Q. B F H M O.
Tape  1 .
      2 B E F F H N P S S T U Y- B H J J N O O Q.
      3 B F H M O.
```

Now after three two way merges from tapes 1 and 2 to tape 3, the second tape becomes empty. Then after two two-way merges from tapes 1 and 3 onto tape 2, the first tape becomes empty. The sort is completed in

two more steps. First a 2-way merge from tapes 2 and 3 onto tape 1 leaves one file on tape 1, one file on tape 2.Then a two-way merge from tapes 1 and 2 leaves the entire sorted file on tape 3.

This merge - until-empty strategy can be extended to work for an arbitrary number of tapes. The figure indicates how 6 tapes might be used to sort with 497 initial runs. If we start as shown with tape 2 the output tape, tape 1 having 61 initial runs etc, then after running a five-way "merge-until-empty", we have tape 1 empty, tape 2 with 61 runs etc. as shown in the second column of the figure. At this point we can rewind tape 1 and make it the output tape and rewind tape 2 and make it an input tape. Continuing in this way we arrive at the entire file sorted on tape 1 as shown by the last column. The merge is broken up into many phases which don't involve all the data, but no direct copying is involved.

The main difficulty in implementing polyphase merge is to determine how to distribute the initial runs. The table a can be built by working backwards: take the largest number in each column, make it zero and add it to each of the other numbers to get the previous column. This technique works for any number of tapes (at least 3): the numbers which arise are "generalized Fibonacci numbers". Of course the number of initial runs may not be known in advance, and it probably won't be exactly a generalized Fibonacci number. Thus a number of "dummy" runs must be added to make the number of initial runs exactly what 'Is needed for the table.

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

Run distribution for sin-tape polyphase merge.

Remarks: A factor we have not considered is the time taken to rewind the tape. Before the merge for the next phase can begin, the tape must be rewound and the computer is essentially idle. It is possible to modify the above method so that virtually all rewind time is overlapped with internal processing and read/write on other tapes. However, the savings over the multiway balanced merge are quite limited. Even polyphase merging is better than balanced merging only for small P, and that not substantially. For P8, balanced merging is likely to run faster than polyphase and for smaller P, the effect of polyphase merging is to save two tapes.

3.6 SUMMARY

External sorting is an important activity especially in large businesses. It is thus crucial that it be performed efficiently External sorting depends to a large extent on system considerations like the type of device and the number of such devices that can used at a time. Thus the choice of an external sorting algorithm is dependent upon external system considerations

The list of algorithms we have studied is not conclusive and the student is advised to consult the references for detailed discussions of additional algorithms. Analysis and distribution strategies for multiway merging and polyphase merging are to be found in Knuth, Volume 3.

3.7 REVIEW EXERCISES

1. Write an algorithm to construct a tree of losers for records Ri $1 \leq i \leq k$ with key values ki 1. Let the tree nodes be Ti $0 \leq i \leq$ =kO with Ti, 1

2. Write an algorithm using a tree of losers, to carry out a k-way merge of k runs, k = 2. Show that if there are n records in the k runs together, then the computing time is $0(n\log_2 k)$.

3. a) Modify algorithm x3 using the run distribution strategy described in the analysis of algorithm x2.

b) Let trw be the time to read/write a block and trew be the time to rewind over one block length. If the initial run creation pass generates m runs for m a power of k, what is the time for k-way merge using your algorithm? Compare this with the corresponding time for algorithm x2.

4. How would you sort the contents of a disk if only one tape (and main memory) were available for use?

5. How would you sort the contents of a disk if no other storage (except main memory) were available for use.

6. Compare the 4-tape and 6-tape multiway balanced merge to polyphase merge with the same number of tapes, for 31 initial runs.

7. How many phases does 5-tape polyphase merge use when started up with 4 tapes containing 26, 15, 22 and 28 runs initially?

8. Obtain a table corresponding to the one in the text for the case of a 5-way polyphase merge on 6 tapes. Use this to obtain the correct initial distribution for 497 runs so that the sorted file is on tape T1. How many passes are made over the data in achieving the sort? How many passes would have been made by a 5- way balanced merge sort on six tapes (algorithm x2)? How many passes would have been made by a 3-way balanced merge sort on 6 tapes (algorithm x3?

3.8 Suggested Reading

1. Fundamentals of Computer algorithms

   Horowitz & Sahni

2. Fundamentals of Data Structures in Pascal

   Horowitz & Sahni

3. The Art of Programming Volume

   SORTING & SEARCHIN
   DONALD. E. KNUTH