Sorting

We have seen how using sorted files can greatly reduce the amount of time spent hunting through files looking for stuff. And we know all about sorting lists in RAM. But how can we sort *files* and how can we *keep* them sorted?

Topic	***Folk & Zoellick***
• Sorting in RAM: a review	§§ 5.3.4, 5.3.5
• Heapsort	§ 7.4
• Keysort	§ 5.4
• Mergesort for Large Files	§ 7.5
• Tapesort	§ 7.6

Internal Sorting

Sorting done entirely in RAM

External Sorting

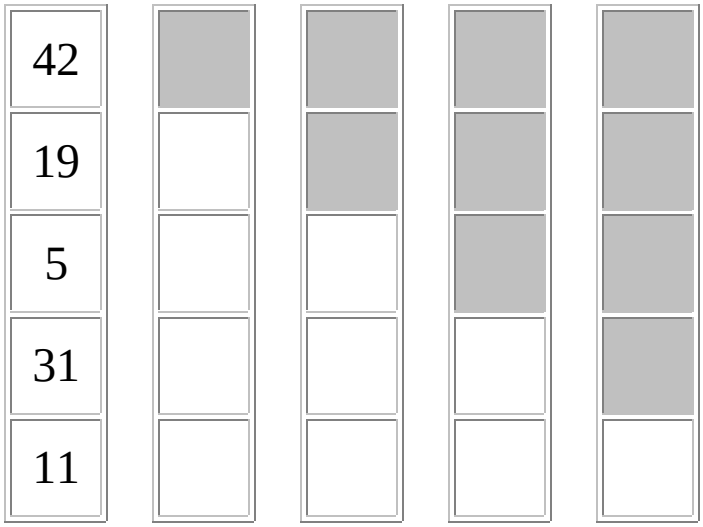
Sorting files that don't fit into RAM

Selection Sort

Selection Sort is a simple sorting algorithm whose performance is not that great.

Selection Sort on a list[1..n]

```
for i from 1 to n-1
   lowest = i

   for j from i+1 to n
      if list[j] < list[lowest]
         lowest = j
   endfor

   swap list[i], list[lowest]
endfor
```



Q: What is the complexity (big O) of *Selection Sort?*

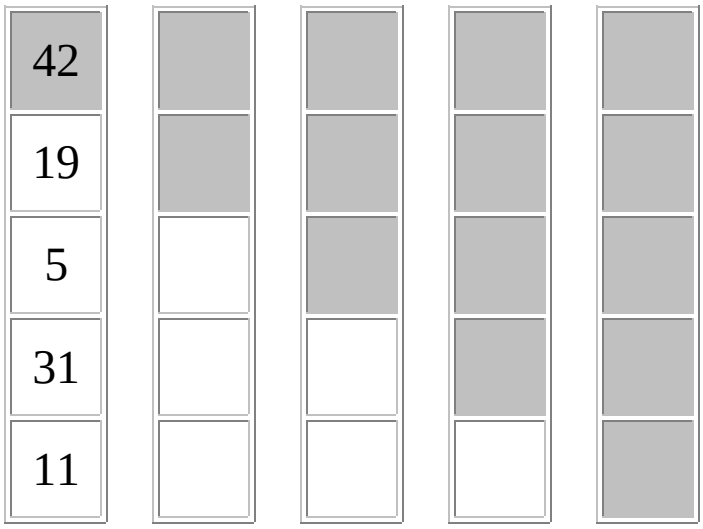
A:

Insertion Sort

Insertion sort is one of the simplest sorting algorithms, yet it also gives quite good performance for sorting lists in RAM.

Insertion Sort on a list[1..n]

```
for i from 2 to n
   insert list[i] into the correct position
   in the sorted list[1..i-1]
endfor
```



Q: What is the complexity (big O) of *Insertion Sort?*

A:

Using Selection Sort for Big Records

Let's say we want to use the *Selection Sort* on a file of records. The records are to be ordered according to their key values. That is, we have a file *F* with records r_1 to r_n having keys k_1 to k_n. We want to reorder the records based on their key values. The problem is, the records are so big that we can only fit a couple of them in RAM at any one time.

Selection Sort (in slightly more detail)

```
for i from 1 to n-1
   read record r_i having key k_i
   lowest = k_i
   lowrecord = i

   for j from i+1 to n
      read record r_j having key k_j
      if k_j < lowest
         lowest = k_j; lowrecord = j
   endfor

   read record r_i
   read record r_lowrecord
   write record r_lowrecord into position i
   write record r_i into position lowrecord
endfor
```

Q: What is the complexity of this algorithm?

A:

Insertion Sort for Big Records

Insertion Sort (in slightly more detail)

```
for j from 2 to n
   read record r_j having key k_j

   find record r_i in 1..j-1 such that
   record r_j belongs just before record r_i

   for k from j-1 down to i
      read record r_k and write it to position k+1
   endfor

   write record r_j into position i
endfor
```

Q: What has happened to the (worst case) complexity of the *Insertion Sort?*

A:

Sorting with Trees

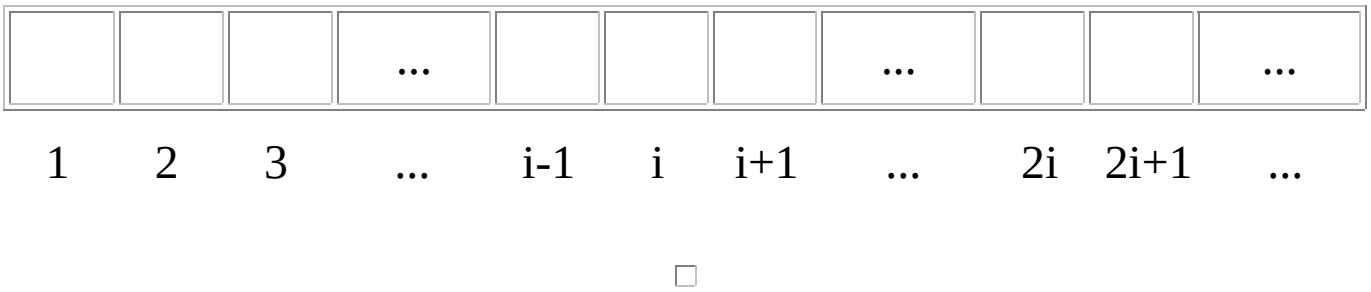
Binary Tree

- a hierarchical data structure
- a collection of nodes which is either empty or consists of a node that has left and right children (themselves binary trees).

Complete Binary Tree

- a binary tree with all leaves at the same level *or* all leaves on the bottom two levels, with leaves on the bottom level as far left as possible.

A complete binary tree can be represented using an array, where each element represents a node. For any node *i*, the value of the node is at position *i* in the array, the left child of *i* is at position *2i* in the array, the right child of *i* is at position *2i+1* in the array.



Complete Binary Tree Example

Here’s an example of the array representation of a complete binary tree:

19	5	31	25	19	22	21	24	11	29

Q: What does the complete tree look like?

A:

Heapsort

Heap

a complete binary tree in which the value of every node is greater than the value of its parent.

The definition of a heap ensures a certain amount of order in the data, but not enough that we can simply read the sorted data off the tree (as we can with a *binary search tree*). But a heap does have enough order that sorting the data in a heap is efficient.

Heapsort is an efficient sorting algorithm that consists of two parts:

1. build a heap from the data
2. output the data from the heap in sorted order

Building a heap from a file of n *records*

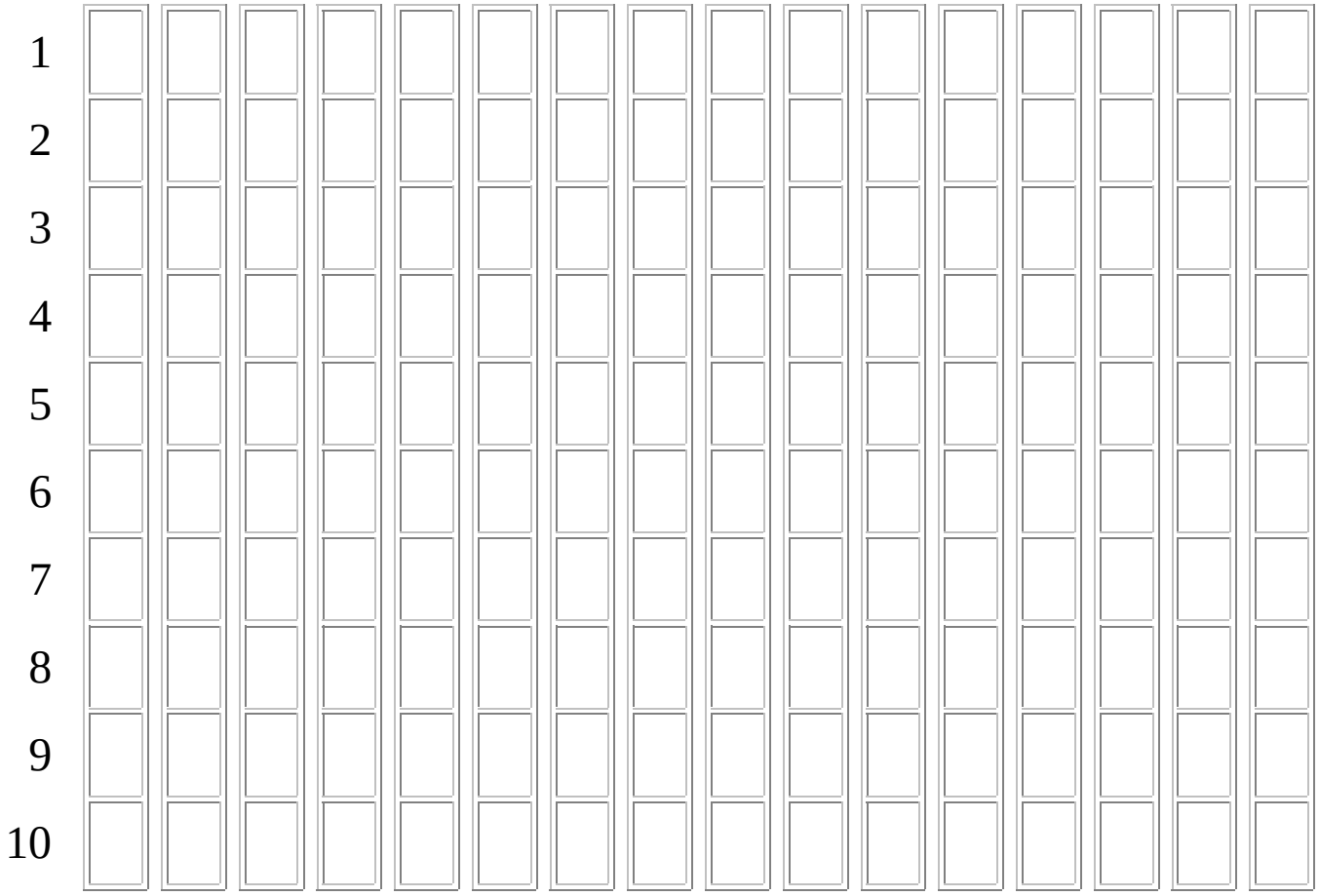
```
for i from 1 to n
   read the next record r having key k
   put record r at the end of the array

   while k < the key of r's parent
      exchange record r with its parent
endfor
```

Heap Building Example

Let’s build a heap from the following file (the first field is the key):

```
19  ctr ...
 5  def ...
31  g ...
25  ctr ...
19  ctr ...
22  ctr ...
21  lw ...
24  def ...
11  rw ...
29  def ...
```

Heap Building Example (cont.)



Heapsort (bis)

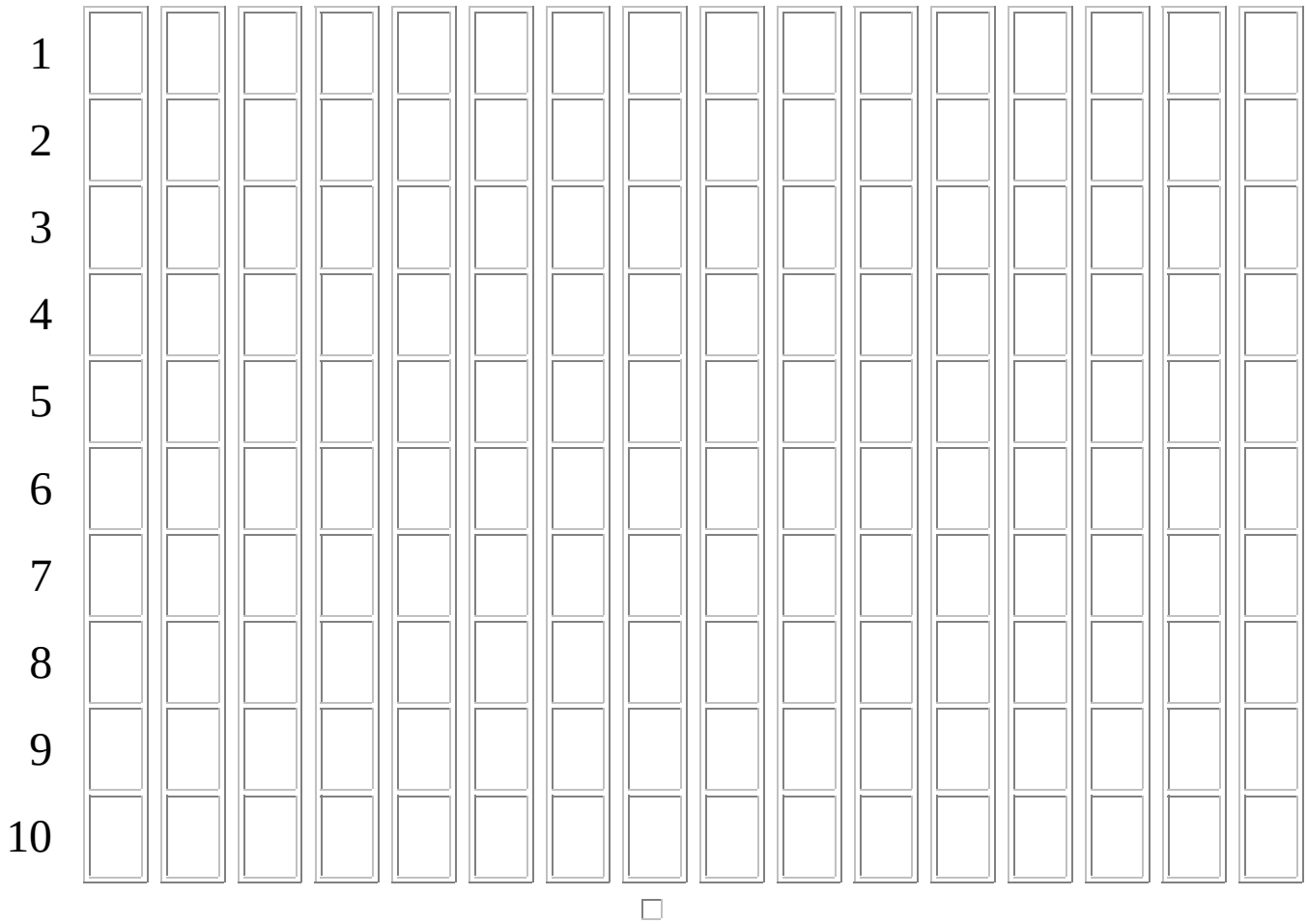
The second part of *heapsort* was to output the data from the heap in sorted order.

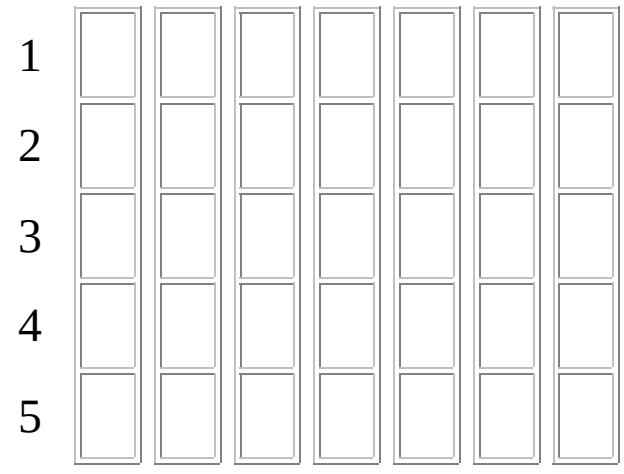
Output the data in the heap in sorted order

```
for i from 1 to n
   write the record in array[1]
   last = n - i
   move the record in array[last+1] to array[1]
      (and call its key k)

   while k > the keys of either of its children
      exchange record r with the child having
         the smaller key
endfor
```

(careful... the algorithm in Figure 7.23 of Folk & Zoellick has a bug)

Sorting While Writing the Heap





What's the Point?

The complexity of *heapsort* is also O(*n*log*n*). What makes it better suited to sorting files than other *n*log*n* sorting algorithms? Let's look at the two parts of *heapsort*:

- *Build a heap from a file of* n *records*
  Recall that the algorithm reads a record, sticks it at the end of the array, and then starts comparing it to records already in the array. That is, to position a record in the heap, you don't need to know the keys of the remaining records in the file.

  Why not keep reading records *i+1*... while moving record *i* into the correct position in the heap!

- *Output the data in the heap in sorted order*
  Instead of writing out every record one at a time, we could buffer records until there are enough to write an efficient amount of data to disk.

  But couldn't we do that with any sorting algorithm?
  Yes, but most algorithms require the data to be completely sorted before writing it out. The sorting part of *heapsort* is still underway during writing, meaning that there will be an advantage to concurrent writing. Furthermore, every time a record is written out, the heap shrinks by one record, meaning that *no extra RAM is required for buffering*.

Q: Isn't that cool?

A:

Please Keep Off the Disk

The best known sorting algorithms are O(*n*log*n*). But we have seen that even algorithms of linear complexity are expensive when the number of disk accesses is linear. Minimizing disk accesses is even more important than minimizing the overall complexity of the algorithm.

If a file is small enough to fit in RAM (if we have enough RAM to hold the whole file), we can sort the file with the number of disk accesses O(*n*):

1. `read the entire file into RAM`	O(*n*)
2. `do your favourite internal sort`	O(*n*log*n*)
3. `write the sorted data out to disk`	O(*n*)

The disk accesses will still be the slowest part, but O(*n*) reads from the file *in order* is *way* better than O(*n*log*n*) reads from random positions. We also saw that using *heapsort* in step 2 provides an improvement on this general algorithm, by allowing overlap between the steps.

□

Q: But what if the file is too big to fit in RAM?

Gimme the Keys

The disk-based versions of the sort algorithms read and write entire records, but it's only the keys that get compared. Why not just store the *keys* in RAM?

Keysort

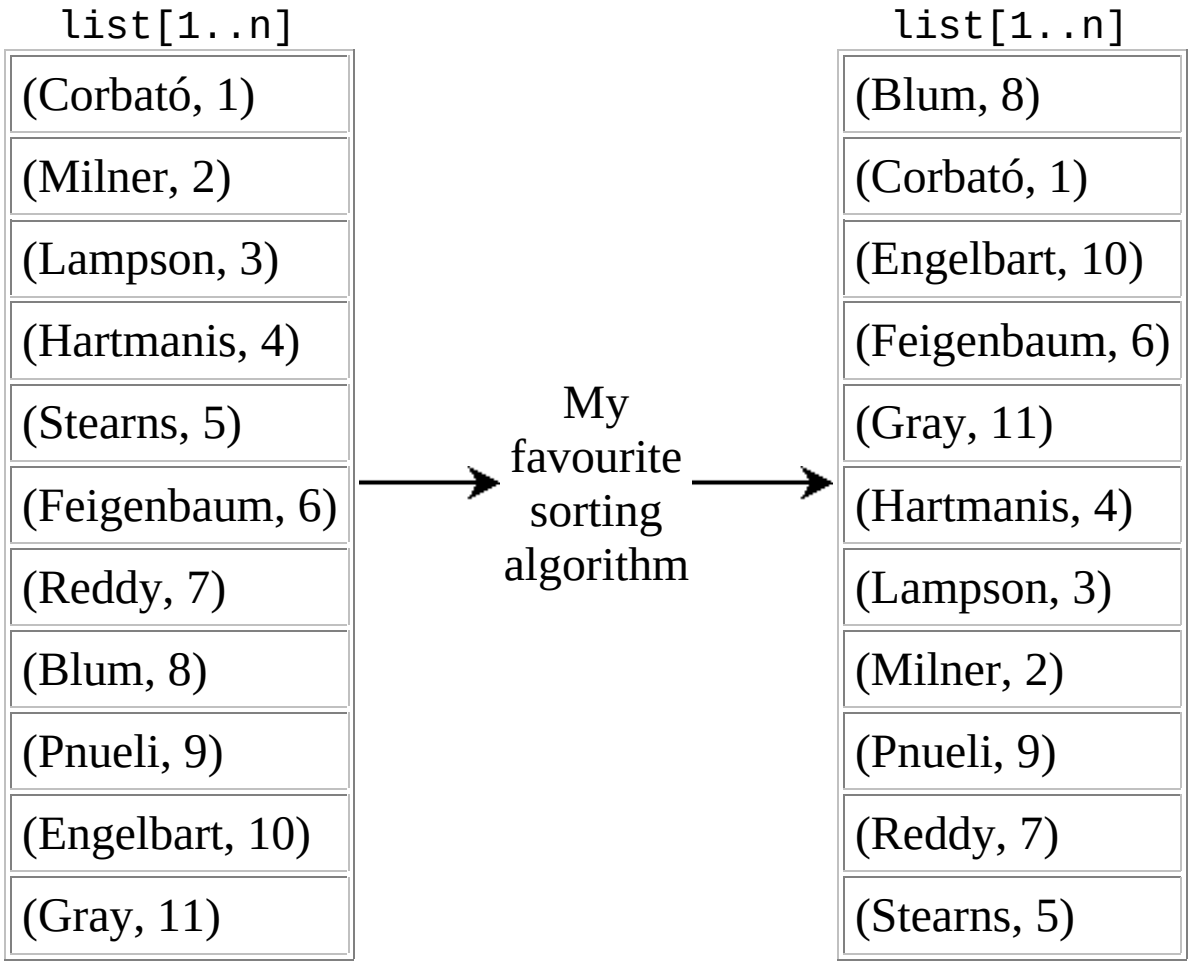
```
open File_in for reading

for i from 1 to n
   read record r_i having key k_i from File_in
   store (k_i, i) in list[i]
endfor

close File_in

do your favourite sorting algorithm on list

open File_in for reading
open File_out for writing

for i from 1 to n
   get (k_j, j) from of list[i]
   seek to record r_j in File_in
   read record r_j from File_in
   write record r_j to position i in File_out
endfor

close files
```

Keysort: An Example

Here’s a file full of records about some people we all know. Of course, these people have *huge* records, so the entire file won’t fit into RAM.

Corbató	Fernando J.	90	EECS	MIT	...
Milner	Robin	91	TCL	CAMB	...
Lampson	Butler W.	92		MSFT	...
Hartmanis	Juris	93	CISE	NSF	...
Stearns	Richard E.	93	CS	ALBY	...
Feigenbaum	Edward	94	CS	STFD	...
Reddy	Raj	94	SCS	CMU	...
Blum	Manuel	95	CSD	UCB	...
Pnueli	Amir	96	MATS	WEIZ	...
Engelbart	Douglas	97		BOOT	...
Gray	James	98		CNVL	...

Keysort: An Example (cont.)



All that Glitters is not Gold

There were three parts to the *keysort* algorithm:

1. read all records from the input file in sequence (storing just the keys)
2. sort the keys in RAM
3. use the sorted keys to read the records from the input file in sorted order, writing them out to the output file in sequence

There's probably no way to get around the first part (but reading records in sequence is fairly efficient).

The second part is probably the fastest: even if it's O(*n*log*n*), the whole shebang goes on in RAM.

What about the last part?

:-(

Q: Why not just *skip* the last part altogether?

Really External Sorting

Sometimes it's not just the *size* of the records that keeps us from sorting in RAM. Sometimes it's the *number* of records. *Keysort* won't help us if there are too many records for even the *keys* to fit in RAM.

Here's an example. Let's say we have 40MB RAM free for sorting. We want to sort a file that's 100MB long. The file has 5 million records, keys take up 10 bytes each.

Q: We obviously can't load the file into RAM. Why can't we use *keysort*?

A:

In order to sort this file, we have no choice but to break the file into smaller files and sort the smaller files individually.

Q: What do we do with the smaller files once they're sorted?

A:

Merging for Sorting Large Files

Using cosequential processing for merging sorted subfiles is obviously the solution to our *big file* problem. But doing so poses as many questions as it answers:

Q: How many subfiles should we split our big file into?

Q: What are the tradeoffs between a large number of small files and a small number of large files?

Q: Does it make sense to do more than one merge stage?

Q: Are there other ways to tailor the merge solution to this particular problem?

Answers to all these questions and more...
next week on File Management!