Cosequential Processing :
- Multiway Merge
- Sorting Large Files

Reference : Folk, Zoellick and Riccardi. Sections 8.3, 8.5 (up to 8.5.3).

Cosequential processing and Multiway Merging

K-way merge algorithm : merge K sorted input lists to create a single sorted output list.

We will adapt our 2-way merge algorithm :

- Instead of `List1` and `List2` keep an array of lists : `List[1]`, `List[2]`, ..., `List[K]`.
- Instead of `item(1)` and `item(2)` keep an array of items : `item[1]`, `item[2]`, ..., `item[K]`.

Merging Eliminating Repetitions

We modify our synchronization step :

```
if item(1) < item(2) then ...
if item(1) > item(2) then ...
if item(1) = item(2) then ...
```

As follows :

```
(1) minitem = index of minimum item in item[1], item[2], ..., item[K]
(2) output item[minitem] to output list
(3) for i=1 to K do
(4)    if item[i]=item[minitem] then
(5)       get next item from List[i]
```

If there are no repeated items among different lists, lines (3)-(5) can be simplified to :
`get next item from List[minitem]`

Different ways of implementing the method :

Solution 1 : when the number of lists is small (say K $\leq$ 8).

- `Line(1)` does a sequential search on `item[1], item[2], ..., item[K]`.
  Running time : O(K)
- `Line(5)` just replaces `item[i]` with newly read item.
  Running time : O(1)

Solution 2 : when the number of lists is large.
Store current items `item[1], item[2], ...,item[K]` into priority queue (say, an array heap).

- `Line(1)` does a `min` operation on the array-heap.
  Running time : $O(1)$
- `Line(5)` performs a `extract-min` operation on the array-heap :
  Running time : $O(\log K)$
  and an `insert` on the array-heap
  Running time : $O(\log K)$

The detailed analysis of both algorithm is somewhat involved.

Let N = Number of items in output list
M = Number of items summing up all input lists
(Note $N \leq M$ because of possible repetitions.)

Solution 1
`Line(1)`: $K \cdot N$ steps
`Line(5)`, counting all executions: $M \cdot 1$ steps
Total time: $O(K \cdot N + M) \subseteq O(K \cdot M)$

Solution 2
`Line(1)` : $1 \cdot N$ steps
`Line(5)`, counting all executions : $M \cdot 2 \cdot \log K$ steps
Total time : $O(N + M \cdot \log K) = O(log(K \cdot M))$

Merging as a Way of Sorting Large Files

Let us consider the following example :

- Characteristics of the file to be sorted:

  8,000,000 records

  Size of a record = 100 bytes

  Size of the key = 10 bytes

- Memory available as a work area : 10 MB (Not counting memory used to hold program, operating system, I/O buffers, etc.)

- Total file size = 800 MB
- Total number of bytes for all the keys = 80 MB

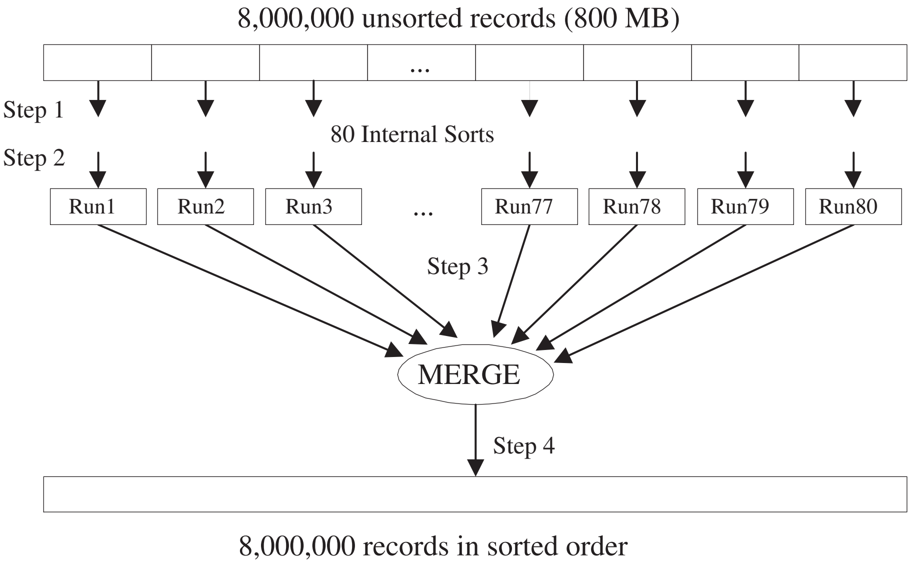
So, we cannot do internal sorting nor keysorting.

Idea :

1. Forming runs : bring as many records as possible to main memory, do internal sorting and save it into a small file.
   Repeat this procedure until we have read all the records from the original file.

2. Do a multiway merge of the sorted files.

   In our example, what could be the size of a run ?
   Available memory = 10 MB = 10,000,000 bytes
   Record size = 100 bytes
   Thus,
   Number of records that can fit into available memory = 100,000 records
   Number of runs = 80 runs



I/O operations are performed in the following times:

1. Reading each record into main memory for sorting and forming the runs.

2. Writing sorted runs to disk.

   The two steps above are done as follows:
   Read a chunk of 10 MEGS; Write a chunk of 10 MEGS
   (Repeat this 80 times)

   In terms of basic disk operations, we spend :
   For reading : 80 seeks[1] + transfer time for 800 MB
   Same for writing.

3. Reading sorted runs into memory for merging. In order to minimize "seeks" read one chunk of each run, so 80 chunks. Since the memory available is 10 MB each chunk can have 10,000,000/80 bytes = 125,000 bytes = 1,250 records

[1] Each chunk is read right after we wrote the previous run, so there is an initial seeking.

How many chunks to be read for each run?
size of a run/ size of a chunk = 10,000,000 / 125,000 = 80

Total number of basic "seeks" = Total number of chunks (counting all the runs) is 80 runs x 80 chunks/run = 80^2 chunks.

Reading each chunk involves **basic seeking**.

4. Writing sorted file to disk the number of basic seeks depends on the size of output buffer : bytes in file/ bytes in output buffer.

   For example, if the output buffer contains 200 K, the number of basic seeks is : 200,000,000 / 200,000 = 4,000.

From steps 1-4 as the number of records (N) grows, step 3 dominates the running time.

There are ways of reducing the time for the bottleneck step (step 3):

- Allocate more hardware (e.g disk drives, memory)
- Perform the merge in more than one step - this reduces the order of each merge and increases the run sizes.
- Algorithmically increase the length of each run.
- Find ways to overlap I/O operations.

For details in the above steps see sections : 8.5.4 - 8.5.11.