External Sorting & the Memory Hierarchy

What happens to sorting when the data is bigger than your memory — and why the answer is "stop counting comparisons, start counting disk blocks."

Dr. Manu Shrivastava
Course Instructor · Consultation Fri 2–5 PM, LHC 308F

40 minutes
Session outcome: understand external sorting and disk-based processing

Today, minute by minute

The problem — and why your fast sorts break 00–05

Lecture 2's promise comes due: 100 GB of data, 8 GB of RAM.

The memory hierarchy, with real numbers 05–11

Registers to disk; seek, latency, transmission; the block; the golden rule.

The idea: external merge sort in two phases 11–16

Run generation + merging. Why merge is the one primitive disks love.

Full trace — 13 records, memory of 3 16–22

Every run, every pass, every merge shown.

The classic analysis — 4,500 records 22–30

750-record memory, 250-record blocks: derive the full cost formula.

Counting passes; why k-way merging wins 30–35

passes = $\log_k m$, and each pass rereads everything.

Real systems + recap + homework 35–40

The 900 MB / 100 MB recipe; Unix sort; database ORDER BY.

Lecture 2's IOU comes due

In Lecture 2 we said basic structures silently assume **data fits in memory**, and promised a fix. Here it is. The setting: a file of records on disk, far larger than RAM. Sort it.

WHY NOT JUST RUN QUICKSORT?

Because the array isn't in memory — only a window of it is

Quicksort, heapsort, and friends assume **O(1) random access**: touch element 7, then element 4,000,000, then element 12 — all equally cheap. On RAM, true. But if the "array" lives on disk, every jump to a far-away element is a **disk seek costing ~10 ms**. A quicksort partition pass over 100 GB makes hundreds of millions of scattered accesses.

The OS will try to hide this with paging — loading 4 KB pages in and out of RAM — and the result is **thrashing**: the machine spends its life moving pages, not sorting. Days, not hours.

EVERYDAY ANALOGY — CARRY IT THROUGH THE LECTURE

Grading 4,500 answer sheets on a small desk

You must arrange 4,500 answer sheets by roll number. The sheets are in the **store room** (disk). Your **desk** (RAM) holds only 750 sheets at a time. Walking to the store room is slow, and each trip you carry a fixed **bundle** of 250 sheets (a block).

You cannot "quicksort" 4,500 sheets on a 750-sheet desk. But you *can*: bring 750, sort them on the desk, tie them into a sorted bundle, return it — repeat 6 times. Then **merge** the 6 sorted bundles, carrying only the top few sheets of each to the desk. That, exactly, is external sorting.

Definitions. `Internal sorting` — all records in main memory throughout (bubble, insertion, quick, heap, merge…). `External sorting` — records live on secondary storage; only chunks visit memory. Different game, different rules, different cost model.

Know your ladder — and what each rung costs

OPERATION	ACTUAL TIME	HUMAN SCALE (1 NS → 1 S)
L1 cache reference	0.5 ns	half a second
Main memory (RAM) reference	100 ns	~1.5 minutes
Read 4 KB randomly from SSD	150 µs	~1.7 days
Read 1 MB sequentially from RAM	250 µs	~3 days
Disk seek (HDD)	**10 ms**	**~4 months**
Read 1 MB sequentially from HDD	30 ms	~1 year

Canonical figures ("latency numbers every programmer should know," Jeff Dean). Exact values drift with hardware; the *ratios* are what matter: RAM vs disk seek is a factor of ~100,000.

ANATOMY OF ONE DISK ACCESS — THREE COSTS (FROM THE TEXT)

- **Seek time (t_s)** — move the read/write head to the right cylinder. The big one: ~10 ms.
- **Latency time (t_l)** — wait for the platter to rotate the right sector under the head.
- **Transmission time (t_{rw})** — actually stream the block of data.

So one input/output costs $t_{IO} = t_s + t_l + t_{rw}$. Seek + latency are pure overhead paid *per access*, not per byte — which is why disks reward **few, large, sequential** accesses and punish **many, small, random** ones.

The block: the unit of data read or written to disk in one go (one "bundle of sheets"). You can never fetch one record; you always fetch its whole block.

THE GOLDEN RULE OF THIS LECTURE

When data lives on disk, **stop counting comparisons — count block I/Os.** A comparison costs nanoseconds; a block access costs milliseconds. One disk access buys you time for roughly a *million* comparisons. Every design decision in external sorting is an attempt to do fewer I/Os, and to make the unavoidable ones sequential.

External merge sort: two phases, both sequential

PHASE 1 — RUN GENERATION (THE SORT PHASE)

Read as many records as memory holds (M records), sort them *internally* with any good in-memory sort (quicksort, heapsort), and write the sorted chunk back to disk. Each sorted chunk is called a **run**. Repeat until the whole file is consumed.

```
while file not exhausted:
    read M records          // sequential read
    sort internally         // free! (vs I/O)
    write sorted run        // sequential write
```

Result: [n/M] sorted runs on disk. Nothing here ever jumps around the disk — every read and write is a straight stream.

PHASE 2 — MERGING (WHY MERGE, OF ALL THINGS?)

Because merging two sorted lists only ever looks at the **front** of each list. To merge two runs of 750 records each, you don't need 1,500 records in memory — you need *one block of each* plus an output block:

- 2 **input buffers** — hold the current block of each run,
- 1 **output buffer** — collects merged records; written out when full,
- input buffer empties → refill with that run's *next* block.

Memory needed: **3 blocks, regardless of run length**. Every block of every run is read once, sequentially. Merge is the only classic sort primitive with this property — that's why external sorting is always merge-based.

Merging pairs of runs doubles run length each pass: 6 runs → 3 → 2 → 1. When one run remains, the file is sorted. This is **2-way external merge sort**, today's workhorse; k-way (merging k runs at once) is the upgrade we'll motivate at the end and build properly next lecture.

INTERACTIVE REFRESHER — MERGE TWO SORTED RUNS, ONE STEP AT A TIME

Two sorted runs, one finger (**amber outline**) on the front of each. Each step: compare the fingers, copy the smaller (**teal**) to the output, advance that finger. Drive it yourself:

> Start. Fingers on A→1 and B→2. Press Next to make the first comparison.

← Prev | Next → | ▶ Auto-play | ↺ Reset | step 0 / 8

The property to notice: each element is **read once, in order** — no finger ever moves backwards. That is what lets each run stream from disk block by block, sequentially.

Watch it work: 13 records, desk of size M = 3

Unsorted file on disk (13 records): `81, 94, 11, 96, 12, 35, 17, 99, 28, 58, 41, 75, 15`. Memory holds only 3 records at a time.

INTERACTIVE TRACE — DRIVE BOTH PHASES YOURSELF

Legend: **amber** = being processed now · **teal** = just written to disk · faded = already consumed · dotted = idle, waiting for a partner.

STEP 1 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11	96 12 35	17 99 28	58 41 75	15

SORTED RUNS WRITTEN TO DISK

— none yet —

Phase 1 begins. The 13-record file sits on disk. Memory holds only M = 3 records. Press Next to read the first chunk.

STEP 2 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11	96 12 35	17 99 28	58 41 75	15

SORTED RUNS WRITTEN TO DISK

— none yet —

Read **81 94 11** into memory — that fills all M = 3 slots. Nothing is sorted or written yet; this is only the read.

STEP 3 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11	96 12 35	17 99 28	58 41 75	15

SORTED RUNS WRITTEN TO DISK

11 81 94

Now sort the 3 resident records → **11 81 94**, then write them back to disk as run R1. Reading and writing are two separate steps.

STEP 4 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11	96 12 35	17 99 28	58 41 75	15

SORTED RUNS WRITTEN TO DISK

11 81 94

Read **96 12 35** into memory.

STEP 5 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96

Sort → **12 35 96**, write as R2. Note: the file as a whole is not remotely sorted — R1 ends at 94, R2 starts at 12.

STEP 6 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96

Read **17 99 28** into memory.

STEP 7 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96 | 17 28 99

Sort → **17 28 99**, write as R3. Same rhythm every time: read, sort, write — two clicks per run.

STEP 8 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96 | 17 28 99

Read **58 41 75** into memory.

STEP 9 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96 | 17 28 99 | 41 58 75

Sort → **41 58 75**, write as R4.

STEP 10 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | **15**

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96 | 17 28 99 | 41 58 75

Only one record remains — read **15** alone into memory.

STEP 11 OF 79

UNSORTED FILE ON DISK (READ 3 AT A TIME)

81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

SORTED RUNS WRITTEN TO DISK

11 81 94 | 12 35 96 | 17 28 99 | 41 58 75 | 15

A single record is trivially "sorted" — write it as run R5 by itself. **Phase 1 complete: 5 sorted runs on disk** (13 = 3+3+3+3+1; a short last run is normal).

STEP 12 OF 79

RUNS AT START OF PASS 1

11 81 94 | 12 35 96 | 17 28 99 | 41 58 75 | 15

Phase 2, pass 1 begins. 5 runs on disk. This pass pairs them up: R1+R2, then R3+R4 — R5, the odd one out, is carried forward untouched.

STEP 13 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

–

Pass 1: merge R1 + R2. Memory holds just the current front of R1, the current front of R2, and the output slot — never a whole run at once, however long the runs are.

STEP 14 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

–

Compare the two fronts: 11 (R1) vs 12 (R2) → 11 is smaller.

STEP 15 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

11

Write 11 to output, advance R1's finger by one.

STEP 16 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

11

Compare the two fronts: 81 (R1) vs 12 (R2) → 12 is smaller.

STEP 17 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

11 12

Write 12 to output, advance R2's finger by one.

STEP 18 OF 79

R1

11 **81** 94

R2

12 **35** 96

MERGED OUTPUT

11 12

Compare the two fronts: 81 (R1) vs 35 (R2) → 35 is smaller.

STEP 19 OF 79

R1

11 **81** 94

R2

12 35 **96**

MERGED OUTPUT

11 12 35

Write 35 to output, advance R2's finger by one.

STEP 20 OF 79

R1

11 **81** 94

R2

12 35 **96**

MERGED OUTPUT

11 12 35

Compare the two fronts: 81 (R1) vs 96 (R2) → 81 is smaller.

STEP 21 OF 79

R1

11 81 **94**

R2

12 35 **96**

MERGED OUTPUT

11 12 35 81

Write 81 to output, advance R1's finger by one.

STEP 22 OF 79

R1

11 81 **94**

R2

12 35 **96**

MERGED OUTPUT

11 12 35 81

Compare the two fronts: 94 (R1) vs 96 (R2) → 94 is smaller.

STEP 23 OF 79

R1

11 81 94

R2

12 35 **96**

MERGED OUTPUT

11 12 35 81 94

Write 94 to output, advance R1's finger by one.

STEP 24 OF 79

R1

11 81 94

R2

12 35 **96**

MERGED OUTPUT

11 12 35 81 94

R1 is exhausted. The rest of R2 (96) is already sorted — no more comparisons needed, next just copy it straight across.

STEP 25 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

11 12 35 81 94 96

Copied. All of R2 is now in the output.

STEP 26 OF 79

R1

11 81 94

R2

12 35 96

MERGED OUTPUT

11 12 35 81 94 96

Merge complete: the first length-6 run = 11 12 35 81 94 96.

STEP 27 OF 79

R3

17 28 99

R4

41 58 75

MERGED OUTPUT

–

Pass 1: merge R3 + R4. Memory holds just the current front of R3, the current front of R4, and the output slot — never a whole run at once, however long the runs are.

STEP 28 OF 79

R3

17 28 99

R4

41 58 75

MERGED OUTPUT

–

Compare the two fronts: 17 (R3) vs 41 (R4) → 17 is smaller.

STEP 29 OF 79

R3

17 28 99

R4

41 58 75

MERGED OUTPUT

17

Write 17 to output, advance R3's finger by one.

STEP 30 OF 79

R3

17 **28** 99

R4

41 58 75

MERGED OUTPUT

17

Compare the two fronts: 28 (R3) vs 41 (R4) → 28 is smaller.

STEP 31 OF 79

R3

17 28 **99**

R4

41 58 75

MERGED OUTPUT

17 28

Write 28 to output, advance R3's finger by one.

STEP 32 OF 79

R3

17 28 **99**

R4

41 58 75

MERGED OUTPUT

17 28

Compare the two fronts: 99 (R3) vs 41 (R4) → 41 is smaller.

STEP 33 OF 79

R3

17 28 **99**

R4

41 **58** 75

MERGED OUTPUT

17 28 41

Write 41 to output, advance R4's finger by one.

STEP 34 OF 79

R3

17 28 **99**

R4

41 **58** 75

MERGED OUTPUT

17 28 41

Compare the two fronts: 99 (R3) vs 58 (R4) → 58 is smaller.

STEP 35 OF 79

R3

17 28 **99**

R4

41 58 **75**

MERGED OUTPUT

17 28 41 58

Write 58 to output, advance R4's finger by one.

STEP 36 OF 79

R3

17 28 **99**

R4

41 58 **75**

MERGED OUTPUT

17 28 41 58

Compare the two fronts: 99 (R3) vs 75 (R4) → 75 is smaller.

STEP 37 OF 79

R3

17 28 **99**

R4

41 58 75

MERGED OUTPUT

17 28 41 58 75

Write 75 to output, advance R4's finger by one.

STEP 38 OF 79

R3

17 28 **99**

R4

41 58 75

MERGED OUTPUT

17 28 41 58 75

R4 is exhausted. The rest of R3 (99) is already sorted — no more comparisons needed, next just copy it straight across.

STEP 39 OF 79

R3

17 28 99

R4

41 58 75

MERGED OUTPUT

17 28 41 58 75 99

Copied. All of R3 is now in the output.

STEP 40 OF 79

R3

17 28 99

R4

41 58 75

MERGED OUTPUT

17 28 41 58 75 99

Merge complete: the second length-6 run = 17 28 41 58 75 99.

STEP 41 OF 79

AFTER PASS 1

11 12 35 81 94 96 | 17 28 41 58 75 99 | 15

R5 = **15** has no partner this pass — it is carried down unchanged. Pass 1 over: 5 runs became 3. (Spot the waste? Hold that thought for the question below.)

STEP 42 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 99

MERGED OUTPUT

–

Pass 2: merge the first six-run + the second six-run. Memory holds just the current front of the first six-run, the current front of the second six-run, and the output slot — never a whole run at once, however long the runs are.

STEP 43 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 99

MERGED OUTPUT

–

Compare the two fronts: 11 (the first six-run) vs 17 (the second six-run) → 11 is smaller.

STEP 44 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 99

MERGED OUTPUT

11

Write 11 to output, advance the first six-run's finger by one.

STEP 45 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 99

MERGED OUTPUT

11

Compare the two fronts: 12 (the first six-run) vs 17 (the second six-run) → 12 is smaller.

STEP 46 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

MERGED OUTPUT

Write 12 to output, advance the first six-run's finger by one.

STEP 47 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

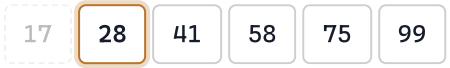
MERGED OUTPUT

Compare the two fronts: 35 (the first six-run) vs 17 (the second six-run) → 17 is smaller.

STEP 48 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN



MERGED OUTPUT

Write 17 to output, advance the second six-run's finger by one.

STEP 49 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

MERGED OUTPUT

Compare the two fronts: 35 (the first six-run) vs 28 (the second six-run) → 28 is smaller.

STEP 50 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

MERGED OUTPUT

Write 28 to output, advance the second six-run's finger by one.

STEP 51 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

MERGED OUTPUT

Compare the two fronts: 35 (the first six-run) vs 41 (the second six-run) → 35 is smaller.

STEP 52 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

MERGED OUTPUT

Write 35 to output, advance the first six-run's finger by one.

STEP 53 OF 79

THE FIRST SIX-RUN

THE SECOND SIX-RUN

MERGED OUTPUT

Compare the two fronts: 81 (the first six-run) vs 41 (the second six-run) → 41 is smaller.

STEP 54 OF 79

THE FIRST SIX-RUN

11 12 35 **81** 94 96

THE SECOND SIX-RUN

17 28 41 **58** 75 99

MERGED OUTPUT

11 12 17 28 35 41

Write 41 to output, advance the second six-run's finger by one.

STEP 55 OF 79

THE FIRST SIX-RUN

11 12 35 **81** 94 96

THE SECOND SIX-RUN

17 28 41 **58** 75 99

MERGED OUTPUT

11 12 17 28 35 41

Compare the two fronts: 81 (the first six-run) vs 58 (the second six-run) → 58 is smaller.

STEP 56 OF 79

THE FIRST SIX-RUN

11 12 35 **81** 94 96

THE SECOND SIX-RUN

17 28 41 58 **75** 99

MERGED OUTPUT

11 12 17 28 35 41 58

Write 58 to output, advance the second six-run's finger by one.

STEP 57 OF 79

THE FIRST SIX-RUN

11 12 35 **81** 94 96

THE SECOND SIX-RUN

17 28 41 58 **75** 99

MERGED OUTPUT

11 12 17 28 35 41 58

Compare the two fronts: 81 (the first six-run) vs 75 (the second six-run) → 75 is smaller.

STEP 58 OF 79

THE FIRST SIX-RUN

11 12 35 **81** 94 96

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75

Write 75 to output, advance the second six-run's finger by one.

STEP 59 OF 79

THE FIRST SIX-RUN

11 12 35 **81** 94 96

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75

Compare the two fronts: 81 (the first six-run) vs 99 (the second six-run) → 81 is smaller.

STEP 60 OF 79

THE FIRST SIX-RUN

11 12 35 81 **94** 96

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75 81

Write 81 to output, advance the first six-run's finger by one.

STEP 61 OF 79

THE FIRST SIX-RUN

11 12 35 81 **94** 96

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75 81

Compare the two fronts: 94 (the first six-run) vs 99 (the second six-run) → 94 is smaller.

STEP 62 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 **96**

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75 81 94

Write 94 to output, advance the first six-run's finger by one.

STEP 63 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 **96**

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75 81 94

Compare the two fronts: 96 (the first six-run) vs 99 (the second six-run) → 96 is smaller.

STEP 64 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75 81 94 96

Write 96 to output, advance the first six-run's finger by one.

STEP 65 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 **99**

MERGED OUTPUT

11 12 17 28 35 41 58 75 81 94 96

the first six-run is exhausted. The rest of the second six-run (99) is already sorted — no more comparisons needed, next just copy it straight across.

STEP 66 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 99

MERGED OUTPUT

11 12 17 28 35 41 58 75 81 94 96 99

Copied. All of the second six-run is now in the output.

STEP 67 OF 79

THE FIRST SIX-RUN

11 12 35 81 94 96

THE SECOND SIX-RUN

17 28 41 58 75 99

MERGED OUTPUT

11 12 17 28 35 41 58 75 81 94 96 99

Merge complete: the length-12 run = 11 12 17 28 35 41 58 75 81 94 96 99.

STEP 68 OF 79

AFTER PASS 2

11 12 17 28 35 41 58 75 81 94 96 99 | 15

Poor 15 waits *again* — still no partner near its size. Pass 2 over: 3 runs became 2.

STEP 69 OF 79

THE LENGTH-12 RUN

11 12 17 28 35 41 58 75 81 94 96 99

R5

15

MERGED OUTPUT

–

Pass 3 — final merge: merge the length-12 run + R5. Memory holds just the current front of the length-12 run, the current front of R5, and the output slot — never a whole run at once, however long the runs are.

STEP 70 OF 79

THE LENGTH-12 RUN

11	12	17	28	35	41	58	75	81	94	96	99

R5

15

MERGED OUTPUT

–

Compare the two fronts: 11 (the length-12 run) vs 15 (R5) → 11 is smaller.

STEP 71 OF 79

THE LENGTH-12 RUN

11	12	17	28	35	41	58	75	81	94	96	99

R5

15

MERGED OUTPUT

11

Write 11 to output, advance the length-12 run's finger by one.

STEP 72 OF 79

THE LENGTH-12 RUN

11	12	17	28	35	41	58	75	81	94	96	99

R5

15

MERGED OUTPUT

11

Compare the two fronts: 12 (the length-12 run) vs 15 (R5) → 12 is smaller.

STEP 73 OF 79

THE LENGTH-12 RUN

11	12	17	28	35	41	58	75	81	94	96	99

R5

15

MERGED OUTPUT

11	12

Write 12 to output, advance the length-12 run's finger by one.

STEP 74 OF 79

THE LENGTH-12 RUN

R5

MERGED OUTPUT

Compare the two fronts: 17 (the length-12 run) vs 15 (R5) → 15 is smaller.

STEP 75 OF 79

THE LENGTH-12 RUN

R5

MERGED OUTPUT

Write 15 to output, advance R5's finger by one.

STEP 76 OF 79

THE LENGTH-12 RUN

R5

MERGED OUTPUT

R5 is exhausted. The rest of the length-12 run (17, 28, 35, 41, 58, 75, 81, 94, 96, 99) is already sorted — no more comparisons needed, next just copy it straight across.

STEP 77 OF 79

THE LENGTH-12 RUN

R5

MERGED OUTPUT

Copied. All of the length-12 run is now in the output.

STEP 78 OF 79

THE LENGTH-12 RUN

11	12	17	28	35	41	58	75	81	94	96	99

R5

15

MERGED OUTPUT

11	12	15	17	28	35	41	58	75	81	94	96	99

Merge complete: the fully sorted file = 11 12 15 17 28 35 41 58 75 81 94 96 99.

STEP 79 OF 79

FINAL SORTED FILE

11 12 15 17 28 35 41 58 75 81 94 96 99

Done. 3 merge passes for 5 runs, exactly $\lceil \log_2 5 \rceil = 3$. Every pass read the whole file and wrote it back, using only 3 blocks of memory throughout — never two whole runs at once, only their current fronts plus one output slot.

QUESTION TO THE CLASS (30 SECONDS)

The lone run 15 was read and written in *every* pass while waiting for its turn. Wasteful? Could we have scheduled the merges differently so short runs are handled cheaply? Hold the thought — that is exactly the **optimal merging of runs** problem, and it leads to Huffman trees in Lecture 6.

WHAT WAS IN MEMORY AT ANY MOMENT?

Never more than 3 records' worth of blocks: two input fingers and an output collector. The file could have had 13 billion records — the memory bill for 2-way merging stays 3 blocks. **Memory bounds the run size and merge width, never the file size.**

4,500 records, 750-record memory, 250-record blocks

The classic textbook example (it appears in university question papers almost every year). Setup: file of **4,500 records** on disk; internal memory sorts at most **750**; one block = **250 records**, so memory holds **3 blocks** and the file spans **18 blocks**.

PHASE 1 — SIX RUNS

Read 3 blocks (750 records) at a time, sort internally (heapsort/quicksort), write back:

`R1: 1–750` `R2: 751–1500` `R3: …` `R4: …` `R5: …` `R6: 3751–4500`

6 runs of 750 records (3 blocks) each. I/O: 18 block-reads + 18 block-writes, plus 6 internal sorts.

PHASE 2 — MEMORY AS THREE 250-RECORD BUFFERS

Split the 750-record memory into **two input buffers + one output buffer** (250 each). Merge R1+R2 block by block: read one block of each run; merge into the output buffer; when it fills, write it out; when an input buffer drains, refill it from the same run. Then R3+R4, then R5+R6. Result of pass 1: three runs of 1,500. Pass 2: merge two of them → 3,000 (the third 1,500-run sits idle). Pass 3: 3,000 + 1,500 → 4,500. Done.

INTERACTIVE — WATCH THE RUNS EVOLVE AND THE BILL GROW

STEP 1 OF 8

FILE ON DISK

4,500 records = 18 blocks

INTERNAL MEMORY

750 records = 3 blocks of 250

Setup. File of 4,500 records; memory sorts at most 750; a block holds 250 records. Everything that follows is arithmetic on these three numbers.

STEP 2 OF 8

RUNS AFTER PHASE 1

R1 · 750 | R2 · 750 | R3 · 750 | R4 · 750 | R5 · 750 | R6 · 750

Run generation + 36 t_{IO} + 6 t_{IS}

Phase 1. Read 3 blocks (750 records) at a time — heapsort in memory — write back. Six runs. I/O: 18 block-reads + 18 block-writes = **36 t_{IO}**, plus **6 t_{IS}** of sorting.

STEP 3 OF 8

RUNS OF 750 (3 BLOCKS EACH)

R1 · 750 | **R2 · 750** | R3 · 750 | R4 · 750 | R5 · 750 | R6 · 750

PASS-1 OUTPUT

1,500

Run generation + 36 t_{IO} + 6 t_{IS}

Pass 1 begins. Memory becomes 2 input buffers + 1 output buffer (250 records each). Merge R1 + R2 block by block: output buffer fills → write it; input buffer drains → refill from the same run.

STEP 4 OF 8

RUNS OF 750 (3 BLOCKS EACH)

R1 · 750 | R2 · 750 | **R3 · 750** | **R4 · 750** | R5 · 750 | R6 · 750

PASS-1 OUTPUT

1,500 | 1,500

Run generation + 36 t_{IO} + 6 t_{IS}

Merge R3 + R4 the same way. Notice every block of every run is read exactly once per pass — all sequential.

STEP 5 OF 8

RUNS OF 750 (3 BLOCKS EACH)

R1 · 750 | R2 · 750 | R3 · 750 | R4 · 750 | **R5 · 750** | **R6 · 750**

PASS-1 OUTPUT

1,500 | 1,500 | 1,500

Run generation + 36 t_{IO} + 6 t_{IS}

Merge pass 1 + 36 t_{IO} + 4500 t_m

Merge R5 + R6. **Pass 1 complete:** the whole file moved once — 18 blocks in + 18 out = **36 t_{IO}**, and all 4,500 records passed through the merger: **4500 t_m**.

STEP 6 OF 8

RUNS AT START OF PASS 2

1,500 | **1,500** | 1,500

PASS-2 OUTPUT

3,000 | 1,500

Run generation + 36 t_{IO} + 6 t_{IS}

Merge pass 1 + 36 t_{IO} + 4500 t_m

Merge pass 2 + 24 t_{IO} + 3000 t_m

Pass 2. Merge two 1,500-runs → 3,000. The third run **sits idle and costs nothing**: only 3,000 records moved = 12 blocks in + 12 out = **24 t_{IO} + 3000 t_m**. Merge *order* changed the bill — remember this for Lecture 6.

STEP 7 OF 8

RUNS AT START OF PASS 3

3,000 | **1,500**

FINAL OUTPUT

4,500 – sorted

Run generation + 36 t_{IO} + 6 t_{IS}

Merge pass 1 + 36 t_{IO} + 4500 t_m

Merge pass 2 + 24 t_{IO} + 3000 t_m

Merge pass 3 + 36 t_{IO} + 4500 t_m

Pass 3. Merge 3,000 + 1,500 → the sorted file of 4,500. Whole file moves again: **36 t_{IO} + 4500 t_m**.

STEP 8 OF 8

FINAL OUTPUT

4,500 — sorted

Run generation + 36 t_{IO} + 6 t_{IS}

Merge pass 1 + 36 t_{IO} + 4500 t_m

Merge pass 2 + 24 t_{IO} + 3000 t_m

Merge pass 3 + 36 t_{IO} + 4500 t_m

Total so far = 132 t_{IO} + 12000 t_m + 6 t_{IS}

Read the total like an engineer: t_{IO} ≈ 10–30 ms while t_m is nanoseconds — the 132 t_{IO} term dominates by orders of magnitude. The CPU terms are a rounding error. The golden rule, in one formula.

The same analysis as a formal table, with proper notation — t_s = max seek, t_l = max latency, t_{rw} = read/write one block, **$t_{IO} = t_s + t_l + t_{rw}$**, t_{IS} = internally sort 750 records, $n \cdot t_m$ = merge n records buffer-to-buffer:

STEP	OPERATION	WHY THAT COUNT	TIME
1	Run generation	read 18 blocks + write 18 blocks + sort 6 chunks	36 t_{IO} + 6 t_{IS}
2	Pass 1: merge R1–R6 in pairs	whole file moves: 18 in + 18 out; all 4,500 records pass through the merger	36 t_{IO} + 4500 t_m
3	Pass 2: merge two 1,500-runs	only 3,000 records move = 12 blocks in + 12 out — **the idle run costs nothing**	24 t_{IO} + 3000 t_m
4	Pass 3: 3,000 + 1,500	whole file again: 18 + 18 blocks	36 t_{IO} + 4500 t_m
	Total		**132 t_{IO} + 12000 t_m + 6 t_{IS}**

- **Read the total like an engineer:** with t_{IO} ≈ 10–30 ms and t_m in nanoseconds, the 132 t_{IO} term dominates by orders of magnitude. The CPU work (12000 t_m + 6 t_{IS}) is almost a rounding error — the golden rule in action.
- **Step 3 is the seed of next lecture:** merging unequal runs in a clever *order* changed the cost. Choosing the best merge order is the WEPL / Huffman-tree problem of Lecture 6.

Every pass rereads everything — so count passes

THE FORMULAS (M RUNS, K-WAY MERGE)

$$\text{levels} = \log_k m + 1 \quad \cdot \quad \textbf{passes} = \log_k m$$

Check against what we've seen (2-way, m = 6): $\log_2 6 \approx 2.58 \rightarrow 3$ passes, 4 levels — matches the 4,500-record example. And each pass costs ≈ one full read + one full write of the file (36 t_{IO} there).

So total merge I/O ≈ **2 × (file blocks) × $\log_k$ m**. The file size is fixed; the only lever you own is **the number of passes**.

PULL THE LEVER YOURSELF: 16 RUNS, PICK YOUR K

K = 2

LEVEL 0 — 16 RUNS

LEVEL 1 — 8 RUNS

LEVEL 2 — 4 RUNS

LEVEL 3 — 2 RUNS

LEVEL 4 — 1 RUN

passes = $\log_2 16$ = **4** → total merge I/O ≈ **8 × file size**

K = 4

LEVEL 0 — 16 RUNS

LEVEL 1 — 4 RUNS

LEVEL 2 — 1 RUN

passes = $\log_4 16$ = **2** → total merge I/O ≈ **4 × file size** — versus 8 × for 2-way.

K = 16

LEVEL 0 — 16 RUNS

LEVEL 1 — 1 RUN

passes = $\log_{16} 16$ = **1** → total merge I/O ≈ **2 × file size** — versus 8 × for 2-way. One pass: the dream, if memory affords 16 buffers.

Merging k runs at once means the whole file is touched only $\log_k m$ times. The dream: enough memory buffers to merge *all* runs in a single pass.

SO WHY NOT K = 1000? THE TWO CATCHES (PREVIEWS OF LECTURE 5)

- **Catch 1 — memory:** a k-way merge needs k input buffers + 1 output buffer minimum (and 2k + 2 for smooth parallel I/O — see next lecture's floating buffers). Buffers shrink as k grows, so each holds fewer records, forcing more frequent refills. k is capped by memory.
- **Catch 2 — CPU per record:** naively finding the minimum among k fronts costs k comparisons per record output. Fix: a **tournament / loser tree** finds the next minimum in $\log_2 k$ comparisons — the star of Lecture 5. With it, internal merge time is $O(n \log_2 k \cdot \log_k m)$ = **$O(n \log_2 m)$, independent of k** — so raising k costs the CPU nothing and saves real I/O.

900 MB of data, 100 MB of RAM — start to finish

THE RECIPE (SINGLE MERGE PASS)

- **Step 1:** Read 100 MB, quicksort it in RAM, write it out as a run. Repeat — 900/100 = **9 runs** on disk.
- **Step 2:** Divide RAM into 10 buffers of 10 MB: **9 input buffers** (one per run) + **1 output buffer**.
- **Step 3:** Load the first 10 MB of each run; do a **9-way merge**. Output buffer fills → append to the final file. An input buffer drains → refill with that run's next 10 MB.

Because 9 runs fit the buffer budget, **one merge pass suffices**: $\log_9 9 = 1$.

TOTAL THE I/O — AND APPRECIATE IT

STAGE	I/O
Run generation: read + write 900 MB	1.8 GB
Merge pass: read + write 900 MB	1.8 GB
Total, all sequential	**3.6 GB**

At ~150 MB/s sequential HDD speed that's ~24 seconds of I/O. The thrashing quicksort alternative makes *random* 4 KB page accesses — at 10 ms each, even one access per 4 KB of data would take ~38 minutes' worth of seeks (900 MB ÷ 4 KB ≈ 230,400 accesses × 10 ms). Same machine, same data: the difference is purely **access pattern**.

WHERE YOU ALREADY USE THIS — EVERY DAY

The Unix/Linux `sort` command is an external merge sort (watch it drop temporary run files in `/tmp` on a big input). **PostgreSQL / MySQL**: an ORDER BY or index build that exceeds `work_mem` "spills to disk" — run generation and merging, verbatim. **MapReduce / Spark**: the shuffle phase between map and reduce is a giant distributed external merge sort. **DuckDB, SQLite** — same story. This lecture's algorithm is running in every serious data system you will ever touch.

Three takeaways, one cliffhanger

- **Count block I/Os, not comparisons.** One seek ≈ a million comparisons; disks reward few, large, sequential accesses.
- **External merge sort = run generation + merge passes.** Merge is the primitive because it streams: fronts only, every block read once per pass, fixed memory regardless of file size.
- **Passes are the currency: $\log_k m$, each costing a full read + write of the file.** Raise k to cut passes — capped by buffer memory, and needing a smarter min-finder than linear scan.

LECTURE 5

Tournament Trees, Buffering & Run Generation

The loser tree that finds the min in log k; floating buffers that keep disk and CPU busy simultaneously; replacement selection — runs 2× longer than memory.

LECTURE 6

Huffman Trees

Merging unequal runs in the cheapest order — today's "idle run 15" question answered optimally, via weighted external path length.

HOMEWORK — BRING SOLUTIONS TO LECTURE 5

- 10,000 records, memory sorts 1,000, block = 100 records. How many runs does Phase 1 produce? How many passes for 2-way merging? For 5-way? Compute total block I/Os for both.
- Re-derive the 4,500-record cost table with a 4,500-record file but **500-record** internal memory and 250-record blocks. Which term grows, and why?
- In pass 2 of the classic example, we merged the two 1,500-runs and let the third idle. Would merging (R + idle) differently change the total? Try all orders of merging runs of sizes 1500, 1500, 1500 — then sizes 750, 1500, 2250. What do you conjecture? (Lecture 6 will prove it.)
- On any Linux machine: `seq 1 50000000 | shuf > big.txt; sort -n big.txt > sorted.txt` — and watch `/tmp` while it runs. Report what you see.
- Reading: Weiss ch. 7 (external sorting); CLRS ch. 2 (the merge subroutine).

Questions?

Dr. Manu Shrivastava — LHC 308F — Friday 2:00–5:00 PM

> *Next: Lecture 5 — Tournament Trees, Buffering, and Run Generation.*