DM207
I/O-Efficient Algorithms and Data Structures

$\boxed{\text{Fall 2015}}$

Rolf Fagerberg

Prologue

You are working for *MegaHard*®, a large software firm whose latest product is the programming language $D^\flat$.

Your boss tells you to expand its standard library to include a sorting routine.

You are a well-trained computer scientist, and fondly remember your algorithms course, where you learned that sorting can be done in time $O(n \log n)$, and that this is optimal.

Browsing through your old textbook, you again delight in the details of the three $O(n \log n)$ algorithms you were taught:

Heapsort, Mergesort, Quicksort,

each ingenious and beautiful in its own way. Which one to choose?

Prologue

What about the constants involved in the O-notation?

You search the literature, and learn that the exact number of comparisons for all three algorithms are quite similar: they all lie between

$$n \log n \quad \text{and} \quad 2n \log n$$

You even inspect the code and conclude that the ratio between comparisons and other basic operations seem quite alike for all three algorithms.

Tough choice.

But there are other qualities to a sorting algorithm:

Prologue

Quicksort is only *expected* $O(n \log n)$ time (not *worst case*).

Mergesort needs *extra space* besides the input array (not *inplace*).

Summing up:

	Worstcase	Inplace
QuickSort		✓
MergeSort	✓	
HeapSort	✓	✓

Knowing that your boss loves a one-size-fits-all solution, you decide on Heapsort.

Prologue

However, Friday night you are bored. You decide to implement *all* three algorithms, to have some fun.

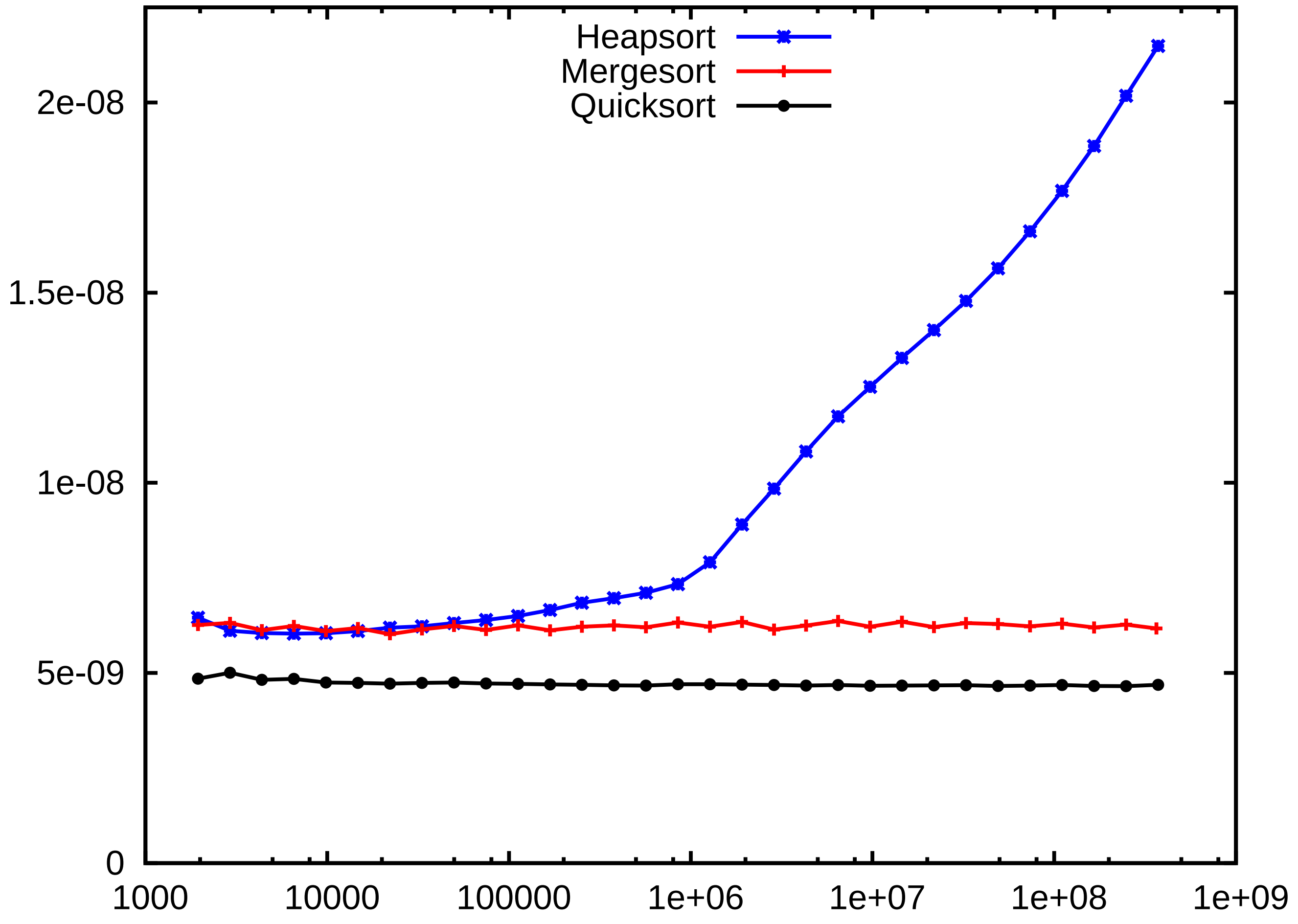
You then run them all on inputs of random `ints`, for growing input sizes n.

You measure running time in seconds, and plot time$/n \log n$ as a function of n.

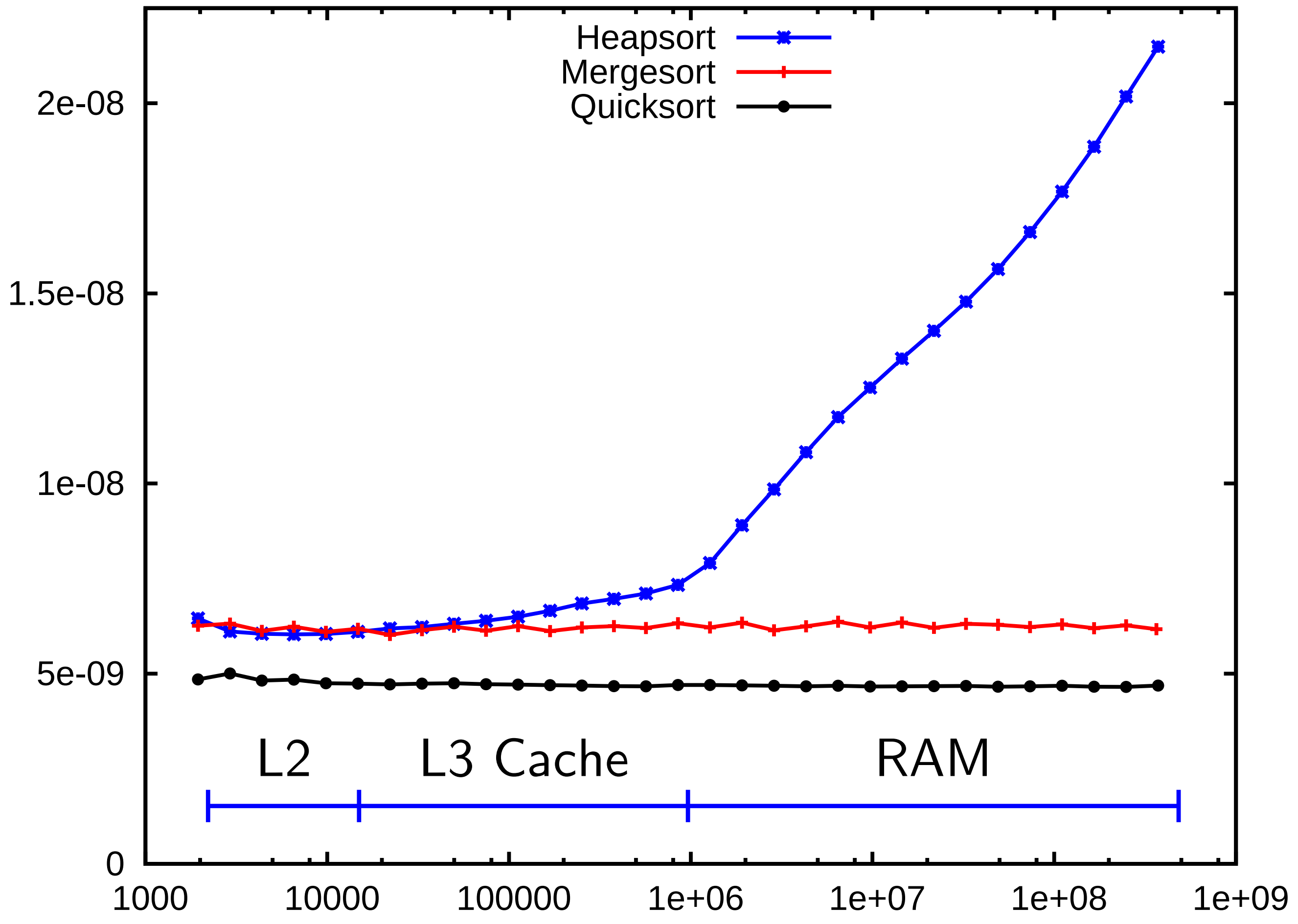
By your analysis above, you know this should generate horizontal lines for all three algorithms, with height of line revealing the constant in the $O(n \log n)$ bound for the algorithm.

Here is the result:

Reality-check

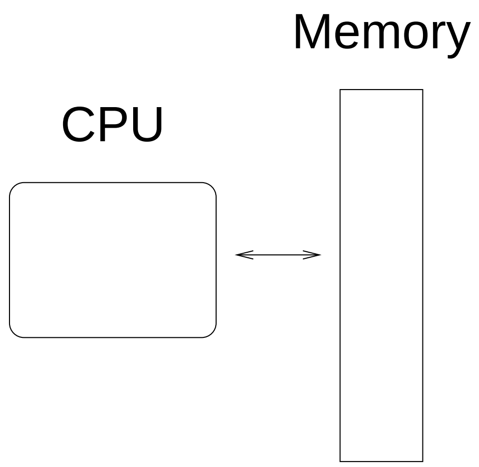


What happened?



Standard model for analysis of algorithms

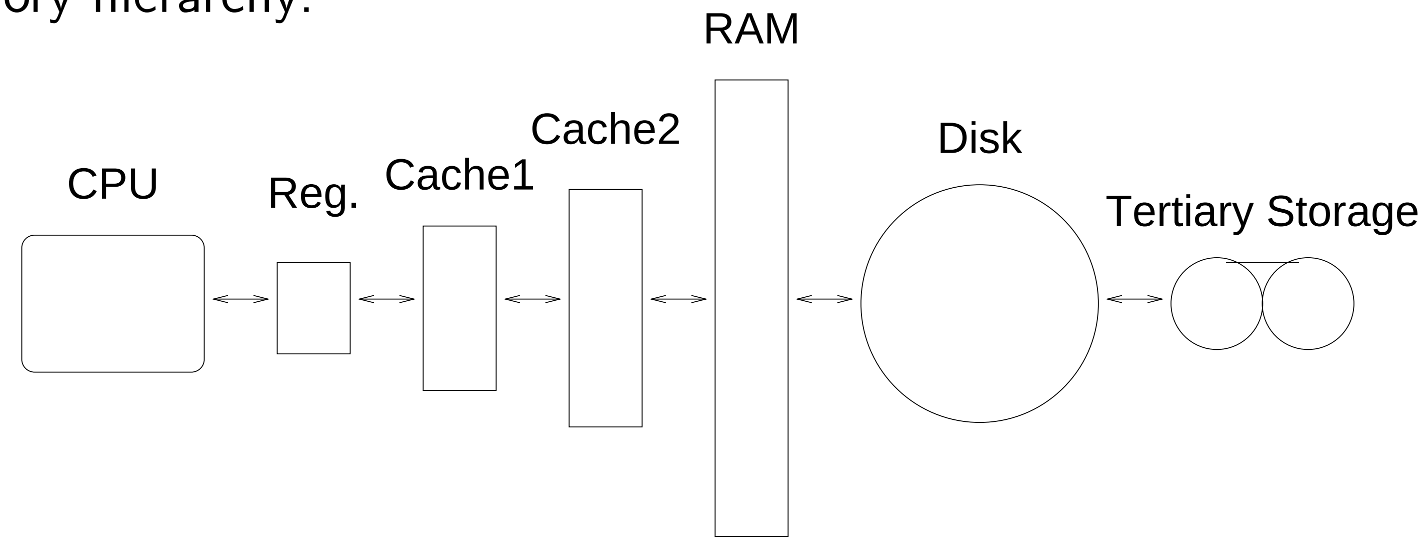
The standard model:



- ADD: 1 unit of time
- MULT: 1 unit of time
- BRANCH: 1 unit of time
- MEMORYACCESS: 1 unit of time $\leftarrow$ Realistic?

Reality

Memory hierarchy:



	Access time	*Volume*
Registers	1 cycle	1 Kb
Cache	5–10 cycles	1 Mb
RAM	50–100 cycles	1 Gb
SSDisk	300,000 cycles	0.1 Tb
HDisk	30,000,000 cycles	1 Tb

Reality

Many real-life problems of **Terabyte** and even **Petabyte** size:

- weather
- geology/geography
- astronomy
- financial
- WWW
- phone companies
- banks

Memory bottleneck

Memory access the bottleneck

$\Downarrow$

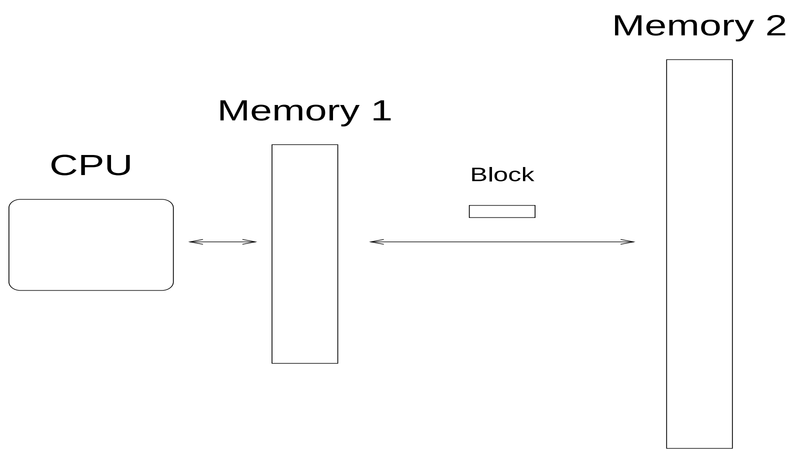
Memory access should be optimized

(not (just) instruction count)

We need new models for this.

Analysis of algorithms

New **I/O-model:**



Parameters:

Aggarwal, Vitter, 1988

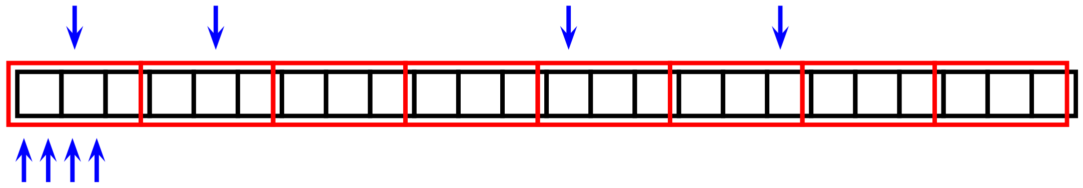
N	$=$	no. of elements in problem.
M	$=$	no. of elements that fits in Memory 1.
B	$=$	no. of elements in a block on disk.

Cost: Number of I/O's (block transfers) between Memory 1 and 2.

Simple Example

Consider two $O(N)$ algorithms:

1. Memory accessed randomly $\Rightarrow$ page fault at each memory access.
2. Memory accessed sequentially $\Rightarrow$ page fault every B memory accesses.



$$\boxed{O(N) \text{ I/Os} \quad \textit{vs.} \quad O(N/B) \text{ I/Os}}$$

Typically for RAM: $B = 4 - 8$. For disk: $B = 10^3 - 10^5$.

10^5 minutes $=$ 70 days, 10^5 days $=$ 274 years. Factor B can be make or break.

Back to the sorting algorithms

QuickSort, MergeSort $\sim$ sequential access

HeapSort $\sim$ random access

So in terms of I/Os:

QuickSort:	$O(N \log(N)/B)$
MergeSort:	$O(N \log(N)/B)$
HeapSort:	$O(N \log(N))$

Course Contents

- The I/O-model(s).
- Algorithms, data structures, and lower bounds for basic problems:
  - Permuting
  - Sorting
  - Searching (search trees, priority queues)
- I/O-efficient algorithms and data structures for problems from
  - computational geometry,
  - strings,
  - graphs.

Along the way I: Principles for designing I/O-efficient algorithms.
Along the way II: Lots of beautiful algorithmic ideas.
Along the way III: Hands-on experience via projects.

Course Style

Lectures:

- Theoretical/analytical (in the style of DM507, just a bit more advanced).
- New stuff: 1995-2015.
- Aim: Principles and methods.

Project work:

- Several small/moderate projects.
- Aim: Hands-on (programming), thinking (theory).

Course Formalities

Literature:

- Based on lecture notes and articles.

Prerequisites:

- DM507.

Duration:

- One full semester.

Credits:

- 10 ECTS (including project).

Exam:

- Project (pass/fail), oral exam (7-scale).

Statement of Aims

After the course, the participant is expected to be able to:

- Describe general methods and results relevant for developing I/O-efficient algorithms and data structures, as covered in the course.
- Give proofs of correctness and complexity of algorithms and data structures covered in the course.
- Formulate the above in precise language and notation.
- Implement algorithms and data structures from the course.
- Do experiments on these implementations and reflect on the results achieved.
- Describe the implementation and experimental work done in clear and precise language, and in a structured fashion.

Basic Results in the I/O-Model

To be proved in the course:

Scanning: $\Theta(\frac{N}{B})$	Permuting: $\Theta(\min\{N, \frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M})\})$
Sorting: $\Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M}))$	Searching: $\Theta(\log_B(N))$

Notable differences from standard internal model:

- Linear time $= O(\frac{N}{B}) \neq O(N)$
- Sorting very close to linear time for normal parameters
- Sorting $=$ permuting for normal parameters
- Permuting $>$ linear time
- Sorting using search trees is far from optimal (N x search $>>$ sort).

Basic Results in the I/O-Model

Scanning:	$\Theta(\frac{N}{B})$	Permuting:	$\Theta(\min\{N, \frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M})\})$
Sorting:	$\Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M}))$	Searching:	$\Theta(\log_B(N))$

Scanning is naturally I/O-efficient ($O(1/B)$ per operation).

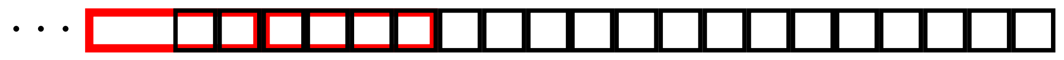
Hence, a few algorithms and data structures (selection, stacks, queues) are I/O-efficient ($O(1/B)$ per operation) out of the box, with the right implementation details. See following pages.

Most other algorithmic tasks need rethinking and new ideas.

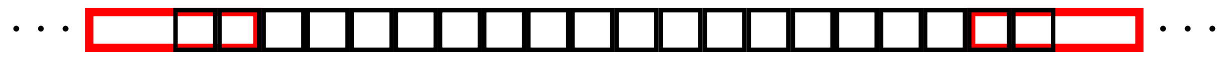
Stacks and Queues

With constant number of blocks in RAM:

$O(1/B)$ I/Os per Push/Pop operation.



$O(1/B)$ I/Os per Dequeue/Enqueue operation.



(The above illustration is for array implementations of stacks and queues. The same analysis will hold if they are implemented as a linked list of blocks of B elements.)

Selection

Recall the problem: For (unsorted) set of elements, find the kth largest.

The classic linear time (wrt. CPU time) algorithm:

1. Split into groups of 5 elements, select median of each.
2. Recursively find the median of this set of selected elements.
3. Split entire input into two parts using this element as pivot.
4. Recursively select in relevant part.

Step 1 and 3 are scans, step 2 recurse on $N/5$ elements, and none of the lists made in step 3 are larger than around $7N/10$ elements.

As (N/B) is the solution to $T(N) = O(N/B) + T(N/5) + T(7N/10)$, $T(M) = O(M/B)$, the algorithm is also linear in terms of I/Os.

(This holds assuming the memory touched by a recursive call (including all sub-calls) is contiguous, and that e.g. LRU caching is done.)