MORE ON SORTING

by

David Toman

University of Waterloo

List of Slides

External Sorting

Internal (in main memory) sorting algorithms rely on the fact that the main memory is just a big *array*

$\Rightarrow$ accessing an element is $O(1)$ (if we know its address)

What if the data resides on external device (disk, tape)?

$\Rightarrow$ and it doesn't fit in main memory

IDEA: we use Merge-sort like approach:

- divide data into "runs" that fit in main memory

  $\Rightarrow$ sort these chunks one-by-one using existing algorithms

- merge the runs at the end

  $\Rightarrow$ multiple "merging phases" may be needed

2-way Merge Sort

Formulated with **tapes**

$\Rightarrow$ to emphasize the sequentiality of the merging phases.

Algorithm (we need 4 tapes):

1. (a) read M element-sized runs from input,
   (b) sort each of them in main memory,
   (c) write odd (even) runs to tape 1 (2), respectively
   (d) rewind tapes 1, 2
2. read and merge the next available runs from tapes 1, 2,
   output on tape 3 (4) for odd (even) pair of input runs.
3. if tape 1 or tape 2 is nonempty go to 2
4. swap tapes 1, 2 for rewound tapes 3, 4
5. if both tapes 1 and 2 are non empty, go to 2

you ended with one non-empty tape—your sorted output.

Example

Assume $M = 4$ and the input is

$23, 54, 12, 14, 78, 53, 56, 90, 99, 87, 78, 74, 25, 51, 39, 47, 88, 72$

$$T_1 = \|12, 14, 24, 54\|74, 78, 97, 99\|72, 88\|$$
$$T_2 = \|53, 56, 78, 90\|25, 39, 47, 51\|\ \|$$

$$T_3 = \|12, 14, 24, 53, 54, 56, 78, 90\|72, 88\|$$
$$T_4 = \|25, 39, 47, 51, 74, 78, 97, 99\|\ \|$$

$$T_5 = \|12, 14, 24, 25, 39, 47, 51, 54, 56, 78, 78, 90, 97, 99\|$$
$$T_6 = \|72, 88\|$$

$$T_7 = \|12, 14, 24, 25, 39, 47, 51, 54, 56, 72, 78, 78, 88, 90, 97, 99\|$$
$$T_8 = \|\ \|$$

Number of Runs

How many times do we have to “swap the tapes”

$\Rightarrow$ this is equal to the # of times we **copy** the whole input

For input size N and run size M, $M < N$ we have:

1. there are $\lceil N/M \rceil$ runs $\lceil$half$\rceil$ on T_1 and $\lfloor$half$\rfloor$ on T_2.
2. every merging phase
   $\Rightarrow$ doubles the size of runs
   $\Rightarrow$ halves the number of runs

Therefore there are $O(\log N/M)$ merging phases
each of which reads/writes $O(N)$ values.

Multi-way Merge Sort

The expensive part of external merge-sort

$\Rightarrow$ every phase must copy the whole input

$\Rightarrow$ attempts to minimize the number of phases

IDEA: use $2k$ tapes instead of 2 tapes

$\Rightarrow$ k-way merging

$\Rightarrow$ the runs grow k times per pass

only $\lceil \log_k(N/M) \rceil$ passes

$\Rightarrow$ problem: needs $2k$ tapes

uneven splitting in the beginning allows us
to use only $k+1$ tapes
$\Rightarrow$ splits based on perfect Fibonacci sequence

Lower Bound for Sorting

We have seen several sorting algorithms:

- $O(n^2)$ (insertion-, selection-, bubble-, and quick-sort)
- $O(n^{3/2})$ (shell-sort)
- $O(n \log n)$ (merge-, heap-, and tree-sort)

can we do better?

$\Rightarrow$ no we can't: any-sort $\in \Omega(n \log n)$

$\Rightarrow$ but how to **prove** this???

Decision Trees

IDEA 1: build a *tree* that represents all possible paths of computation the analyzed algorithm can follow

$\Rightarrow$ we have a (binary) node for each "decision"
the algorithm makes

IDEA 2: figure out how many different **leaves** such a tree must have

$\Rightarrow$ often depends on number of inputs
on which the algorithm behaves "differently"
e.g, must perform different operations to finish
with a correct answer

Now we figure out what the minimal height of
such a tree must be
$\Rightarrow$ this is the worst case for **every** computation.

Decision Tree for Sorting

How does the tree look for an **arbitrary** sorting algorithm?

- internal nodes

  $\Rightarrow$ questions of the form $A[i] \leq A[j]$ for $0 < i \neq j \leq n$
  $\Rightarrow$ left subtree: further questions if the above is true
  $\Rightarrow$ right subtree: further questions if the above is false

- leaves

  $\Rightarrow$ permutations on n elements (representing the permutation that "sorts" the input array)

Lower Bound Theorem

Theorem: Comparison-based sort algorithms take $\Omega(n \log n)$ time.

Proof: An arbitrary decision tree must have $n!$ leaves $\Rightarrow$ there are $n!$ different inputs to be sorted.

Out of all such trees, the balanced one has minimal height $\lceil \log n! \rceil$—the minimal number of comparisons needed by a sorting algorithm in worst case.

$$
\begin{aligned}
\log n! &= \log \big(n(n-1)(n-2)\cdots \tfrac{n}{2} \cdot (\tfrac{n}{2}-1)\cdots 2 \cdot 1\big) \\
&\geq \log \Big(\big(\tfrac{n}{2}\big)^{n/2} \cdot 1^{n/2}\Big) = \log \big(\tfrac{n}{2}\big)^{n/2} \\
&= \tfrac{n}{2} \log \big(\tfrac{n}{2}\big) \in \Omega(n \log n)
\end{aligned}
$$

Linear Sort Algorithms

Consider the following Algorithm:

bucket-sort =

$B:$ **array** $[1..m]$ **of integer** initialized with 0's;

while not end-of-input

$i :=$ next element from input

$B[i] := B[i] + 1;$

for $i := 1$ **to** m

output $B[i]$ times the element i

Complexity (for input with n elements)?

$$O(m) + O(n) + O(m + n) = O(m + n)$$

Did we just violate the Lower Bound Theorem?

$\Rightarrow$ no: this is not a *comparison-based* sorting

Radix Sort

Bucket sort needs too much space (for the array B)

IDEA: "digital sorting": base sort on digits in the key(s)

- bucket-sort based on the *least significant* digit
  $\Rightarrow$ for each bucket maintain a *list* of elements
  $\Rightarrow$ sorted by the last digit
- take the buckets in *increasing order*
  $\Rightarrow$ insert content of the lists into 2nd set of buckets
  based on the *2nd least significant digit*
  $\Rightarrow$ now sorted by the last two digits
- etc, until you exhaust the maximal number of digits
  $\Rightarrow$ now the buckets contain "sorted input"

An Example

Alphabet = $\{0, \ldots, 3\}$
Input = $\{123, 122, 132, 223, 201, 301\}$

B	phase 1	phase 2	phase 3
0		$201, 301$	
1	$201, 301$		$122, 123, 132$
2	$122, 132$	$122, 123, 223$	$201, 223$
3	$123, 223$	132	301

Summary

- External sorting is **essential** in Database Systems
  - $\Rightarrow$ sort-merge join operation
  - $\Rightarrow$ extremely efficient n-way merging
- comparison-based sort $\Theta(n \log n)$
  - $\Rightarrow$ can't be improved
- if the key range is small we can use bucket-sort
  - $\Rightarrow$ or radix sort