CSE 326: Data Structures

Topic #13: Sorting Lower Bounds and Breaking the $\Omega(n \log n)$ Barrier

Ashish Sabharwal
Autumn, 2003

Today's Outline

– *Thanks for the feedback!*

- Finish QuickSort, QuickSelect
- **Lower Bounds**
  – general flavor
  – for sorting
- Breaking the barrier: **BucketSort, RadixSort**

Feedback Summary

Things going well
– pace of lectures
– tablet PC stuff
– group quizzes, midterm review

Issues
– pace of lectures
– tablet PC stuff
– quiz section coordination with lecture / other Q.S.
– PS slides vs. PDF slides

Lower Bounds: for *An Algorithm*

Algorithm A has a lower bound $\Omega(T(n))$ if there exists an input of size n on which A takes $\Omega(T(n))$ time.

E.g.
- insertion in Binary Heap has lower bound $\Omega(\log n)$ because inserting a very small element requires $\Omega(\log n)$ percolateUp operations.
- Insertion Sort has lower bound $\Omega(n^2)$ because it needs so many operations when input it reverse sorted

---

Lower Bounds: for *A Problem*

Problem P has a lower bound $\Omega(T(n))$ if for *every algorithm A* that solves P, there exists an input of size n on which A takes $\Omega(T(n))$ time.

- Very hard to prove because they must hold for *any algorithm* to solve P !!!
- Strategy: restrict computational model
  – Turing machines: very general, no lower bounds known
  – Circuits with *and*, *or*, *not* gates : more structured, still hard
  – Circuits w/o any *not* gates : know non-trivial bounds
  – Proof systems : the area I work in
  – …

Lower Bounds: for *Classes of Algorithms*

Problem P has a lower bound $\Omega(T(n))$ under class C if for *every algorithm $A \in C$* that solves P, there exists an input of size n on which A takes $\Omega(T(n))$ time.

Still quite hard, but feasible. E.g.

- Sorting using only comparisons: $\Omega(n \log n)$
  – Applies to insertion sort, selection sort, bubble sort, shell sort, merge sort, quick sort, heap sort, tree sort, and *any other sorting algorithm based only on comparisons!*
- Sorting by only exchanging adjacent elements: $\Omega(n^2)$
  – Average-case; applies to insertion sort, selection sort, bubble sort, and *any other sorting algorithm satisfying the criterion!*

Lower Bound #1

Theorem: Any algorithm that sorts by comparing and exchanging only adjacent elements must take Ω(n^2) time *on average*.

Details on white board; in book

Proof idea:

- Count the average number of <u>inversions</u> in an array
- Argue that each exchange of adjacent elements can fix only one inversion
  - Gives Ω(n^2) average-case lower bound for insertion sort, selection sort, bubble sort, and *any* other sorting algorithm that satisfies the criterion!

Lower Bound #2

Theorem: Any algorithm that sorts by only comparing elements must take Ω($n \log n$) time *in the worst case*.

Details on white board; in book

Proof idea:

- Represent given algorithm as a <u>decision tree</u>
- Argue that decision tree must have depth Ω($n \log n$)
- Conclude that algorithm must take so much time
  - Gives Ω($n \log n$) worst-case lower bound for *all* sorting algorithms we have seen, and *any* others that satisfy the criterion!

BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and K, create an array `count` of size K, increment counts while traversing the input, and finally output the result.

Example K=5. Input = (5,1,3,4,3,2,1,1,5,4,5)

`count` array	
1	
2	
3	
4	
5	

Running time?

BucketSort Complexity: Θ($n+K$)

- Case 1: K is a constant
  - BinSort is linear time
- Case 2: K is variable
  - Not simply linear time
  - Could even be worst than quadratic!
- Case 3: K is constant but large (e.g. 2^{32})
  - ???

Digression: Stable Sorting

- Stable Sorting algorithm
  - Items in input with the same value end up in the same order as when they began.
- Are the following stable:
  - BucketSort?
  - MergeSort?
  - QuickSort?

Fixing impracticality: RadixSort

- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything
- <u>Idea</u>: BucketSort on each digit, least significant to most significant (lsd to msd)

RadixSort – magic!

- Input:126, 328, 636, 341, 416, 131, 328

BucketSort on lsd:

0	1	2	3	4	5	6	7	8	9

BucketSort on next-higher digit:

0	1	2	3	4	5	6	7	8	9

BucketSort on msd:

0	1	2	3	4	5	6	7	8	9

Not magic… it provably works

Claim: after i^{th} BucketSort, i lsd's are sorted.

– e.g. K=10, i=3, values 1776 and 8234:
8234 comes before 1776 after the 3rd pass.

Proof: By induction. (left as an exercise)

Time to play at home…

- RadixSort the following values using K=10:
  95, 3, 927, 187, 604, 823, 805, 422, 159, 98, 123, 3, 987, 125.

- Given arbitrary numbers A_1, A_2, …A_n, and a base K, what is the overall running time of radix sort?

(extra space)

Radixsort: Complexity

- How many passes?
- How much work per pass?
- Total time?
- Conclusion?
- In practice
  - RadixSort only good for large number of elements with relatively small values
  - Hard on the cache compared to MergeSort/QuickSort

What data types can you RadixSort?

- Any type T that can be BucketSorted
- Any type T that can be broken into parts A and B such that
  - You can reconstruct T from A and B
  - A can be RadixSorted
  - B can be RadixSorted
  - A is always more significant than B, in ordering

RadixSorting Numbers

- 1-digit numbers can be BucketSorted
- 2 to 5-digit numbers can be BucketSorted without using too much memory
- 6-digit numbers, broken up into A=first 3 digits, B=last 3 digits, can be RadixSorted
  - A and B can reconstruct original 6-digits
  - A and B are both RadixSortable as above
  - A always more significant than B

19

RadixSorting Strings

- 1 character can be BucketSorted
- A few characters can be BucketSorted
- Break larger strings into characters or groups of characters
  - e.g. break names into last name, first name;
    sort on first name, then sort (stably) on last name

20

To Do

- Keep working on Project #3
- Finish reading Chapter 7
  (don't spend too much time on External Sorting)

21