CSE 326 Lecture 17: Out of Sorts

- Items on Today's Menu:
  - How fast can we sort?
    - Lower bound on comparison-based sorting
  - Tricks to sort faster than the lower bound
  - External versus Internal Sorting
  - Practical comparisons of internal sorting algorithms
  - Summary of sorting
- Covered in Chapter 7 of the textbook

How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- Can we believe hacker/hackeress Pat Swe (pronounced "Sway") from Swetown (formerly **S**oft**w**ar**e**ville), USA, who claims to have discovered an O(N log log N) sorting algorithm?

The Answer is No! (if using comparisons only)

- Recall our basic assumption: we can only compare two elements at a time – how does this limit the run time?
- Suppose you are given N elements
  - Assume no duplicates – any sorting algorithm must also work for this case
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - How many distinct sequences exist?

The Answer is No! (if using comparisons only)

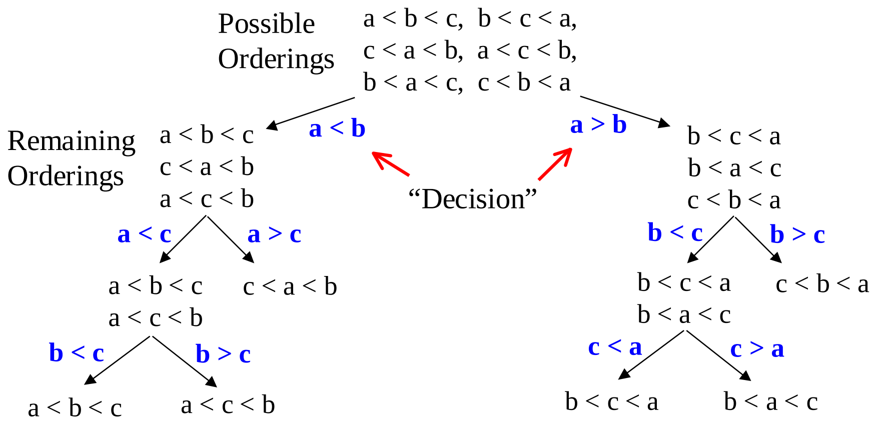
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - Orderings: 1. a b c 2. b c a 3. c a b 4. a c b 5. b a c 6. c b a
  - N = 3: We have 6 orderings = 3•2•1 = 3!
- For N elements, how many possible orderings exist?

The Answer is No! (if using comparisons only)

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - Orderings: 1. a b c 2. b c a 3. c a b 4. a c b 5. b a c 6. c b a
  - 6 orderings = 3•2•1 = 3!

- For N elements: $\underset{\text{N choices}}{\_\_\_}\ \underset{\text{(N-1) choices}}{\_\_\_}\ \_\_\_\ \ldots\ \_\_\_\ \underset{\text{1 choice}}{\_\_\_}$
  = N! orderings

A "Decision Tree" for Sorting N=3 Elements



Leaves contain all possible orderings of a, b, c

Decision Trees and Sorting

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for N distinct elements?
- Only 1 leaf has correct sorted ordering for given a, b, c
- Each sorting algorithm corresponds to a decision tree
  - Finds correct leaf by following edges (= comparisons)
- Run time $\geq$ maximum no. of comparisons
  - Depends on: depth of decision tree
  - What is the depth of a decision tree for N distinct elements?

Lower Bound on Comparison-Based Sorting

- Suppose you have a binary tree of depth d . How many leaves can the tree have?
  - E.g. Depth = 1 $\rightarrow$ at most 2 leaves
  - Depth = 2 $\rightarrow$ at most 4 leaves, etc.
  - Depth = d $\rightarrow$ how many leaves?

Lower Bound on Comparison-Based Sorting

- A binary tree of depth d has at most 2^d leaves
  - E.g. depth d = 1 2 leaves, d = 2 4 leaves, etc.
  - Can prove by induction
- Number of leaves $L \le 2^d$ **$d \ge \log L$**
- Decision tree has L = N! leaves
  - Depth $d \ge \log(N!)$
  - What is log(N!)? (first, what is log(A•B)?)

Lower Bound on Comparison-Based Sorting

- Decision tree has L = N! leaves
  - Depth $d \ge \log(N!)$
  - What is log(N!)?
  - $\log(N!) = \log N + \log(N-1) + \ldots \log(N/2) + \ldots + \log 1$

    $\ge \log N + \log(N-1) + \ldots \log(N/2)$ (N/2 terms only)

    $\ge (N/2)\cdot\log(N/2) =$ **$\Omega(N \log N)$**
- Result: Any sorting algorithm based on comparisons between elements requires **$\Omega(N \log N)$** comparisons

Lower Bound on Comparison-Based Sorting

- Decision tree has L = N! leaves
  - Depth d ≥ log(N!)
  - What is log(N!)? (first, what is log(A•B)?)
  - log(N!) = **Ω(N log N)**
- Result: Any sorting algorithm based on comparisons between elements requires **Ω(N log N)** comparisons
- Corollary: Run time of any comparison-based sorting algorithm is **Ω(N log N)**
  - Can never get an O(N log log N) comparison-based sorting algorithm (sorry, Pat Swe!)

Hey! (you say)…what about Bucket Sort?

- Recall: Bucket sort
  - Elements are integers in the range 0 to B-1
  - Idea: Array Count has B slots ("*buckets*")
  1. Initialize: Count[i] = 0 for i = 0 to B-1
  2. Given input integer i, Count[i]++
  3. After reading all inputs, scan Count[i] for i = 0 to B-1 and print i if Count[i] is non-zero
- What is the running time for sorting N integers?

What’s up with Bucket Sort?

- Recall: Bucket sort   Elements are integers known to always be in the range 0 to B-1
- What is the running time for sorting N integers?
  - Running Time: O(B+N)
    - B to zero/scan the array and N to read the input
  - If B is Θ(N), then running time for Bucket sort = **O(N)**
  - **Doesn’t this violate the Ω(N log N) lower bound result??**

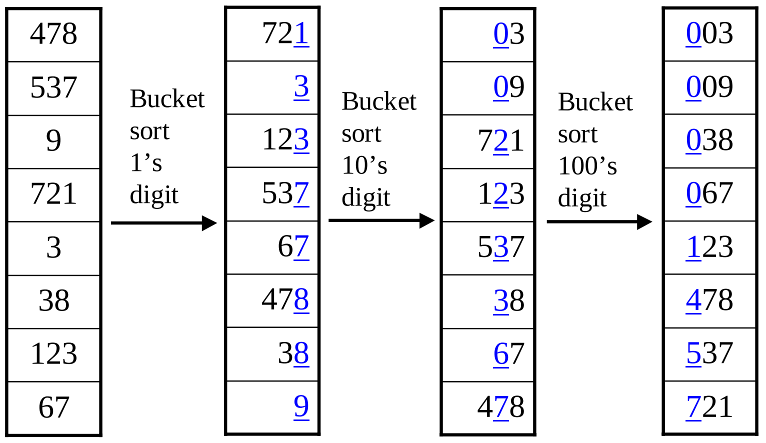
The Scoop behind Bucket Sort

- Recall: Bucket sort   Elements are integers known to always be in the range 0 to B-1
- What is the running time for sorting N integers?
  - Running Time: O(B+N)
  - If B is Θ(N), then running time for Bucket sort = **O(N)**
  - **Doesn’t this violate the O(N log N) lower bound result??**
- **No – When we do Count[i]++, we are comparing one element with all B elements, not just two elements**
  - Not regular 2-way comparison-based sorting

Radix Sort = Stable Bucket Sort

- Problem: What if number of buckets needed is too large?
- Recall: Stable sort = a sort that does not change order of items with same key
- Radix sort = **stable bucket sort on "slices" of key**
  1. Divide integers/strings into digits/characters
  2. Bucket-sort from least significant to most significant digit/character
     - Uses linked lists – see Chap 3 for an example
  - **Stability ensures keys already sorted stay sorted**
  - Takes O(P(B+N)) time where P = number of digits

Radix Sort Example



Internal versus External Sorting

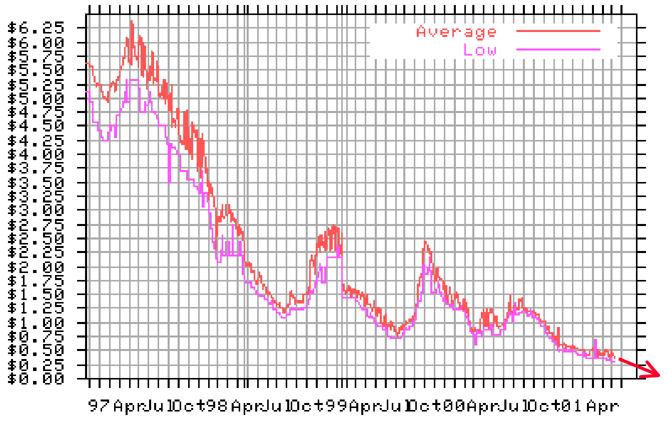
- So far assumed that accessing A[i] is fast – Array A is stored in internal memory (RAM)
  - Algorithms so far are good for internal sorting
- What if A is so large that it doesn't fit in internal memory?
  - Data on disk or tape
  - Delay in accessing A[i]
    - E.g. need to spin disk and move head
- Need sorting algorithms that minimize disk/tape accesses
  - Enter…External sorting

External Sorting

- Sorting algorithms that minimize disk/tape accesses
  - External sorting – Basic Idea:
    - Load chunk of data into RAM
      - Sort this data
      - Store this "run" back on disk/tape
    - Repeat for all data
    - Then: Use the Merge routine from Mergesort to merge the sorted runs
    - Repeat until you have only one run (one sorted chunk)
    - Text gives some examples
- Waittaminute!! How relevant is external sorting?

Internal Memory is getting dirt cheap…

Price (in US$) for 1 MB of RAM

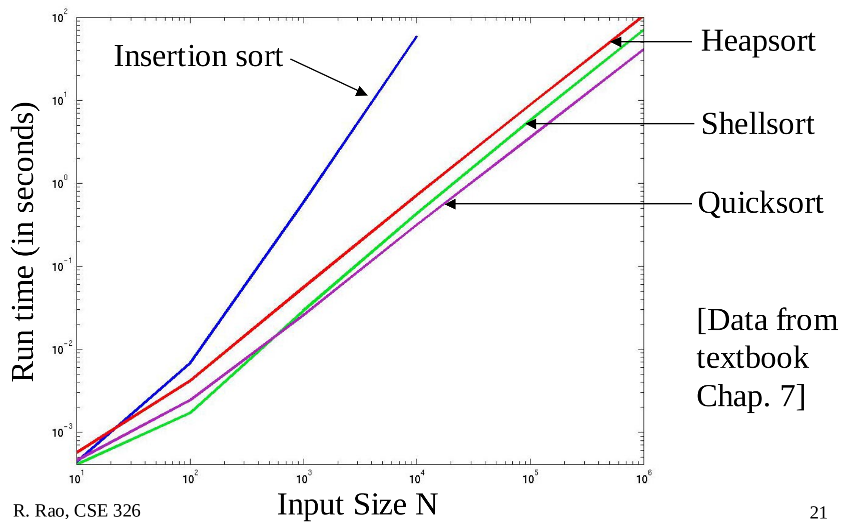


From: http://www.macresource.com/mrp/ramwatch/trend.shtml

External Sorting: A (soon-to-be) Relic of the Past?

- Price of internal memory is dropping, memory size is increasing, both at exponential rates (Moore’s law)
- Quite likely that in the future, data will probably fit in internal memory for reasonably large input sizes
- Tapes seldom used these days – disks are faster and getting cheaper with greater capacity
- So, for most practical purposes, internal sorting algorithms such as Quicksort should prove to be sufficiently efficient

Okay…so let's talk about practical performance



[Data from textbook Chap. 7]

Summary of Sorting

- Sorting choices:
  - $O(N^2)$ – Bubblesort, Selection Sort, Insertion Sort
  - $O(N^x)$ – Shellsort (x = 3/2, 4/3, 2, etc. depending on incr. seq.)
  - O(N log N) average case running time:
    - Heapsort: needs 2 comparisons to move data (between 2 children and parent) – may not be fast in practice (see graph)
    - Mergesort: easy to code but uses O(N) extra space
    - Quicksort: fastest in practice but trickier to code, $O(N^2)$ worst case
  - O(P·N) – Radix sort (using Bucket sort) for special cases where keys are P digit integers/strings

The Practical Side of Sorting

- Practical Choices:
  - When N is large, use Quicksort with median-of-three pivot
  - For small N (< 20), N log N sorts are slower due to extra overhead (larger constants in big-oh function)
  - For $N < 20$, use Insertion sort
  - A Good Heuristic:
    - In Quicksort, do insertion sort when sub-array size < 20 (instead of partitioning) and return this sorted sub-array for further processing
    - Speeds up the running time

Next time:

Data Structures for Union and Find operations

(sorry, not the kind seen in Frat parties)

To do:

Finish chapter 7

Read chapter 8