Notes on Linear Sorting

Geoffrey Matthews

May 9, 2016

Comparison sorts

- The only operation that may be used to gain information about a sequence is comparisons between pairs of elements.
- All sorts seen so far are comparison sorts:
  - insertion sort
  - merge sort
  - quicksort
  - heapsort

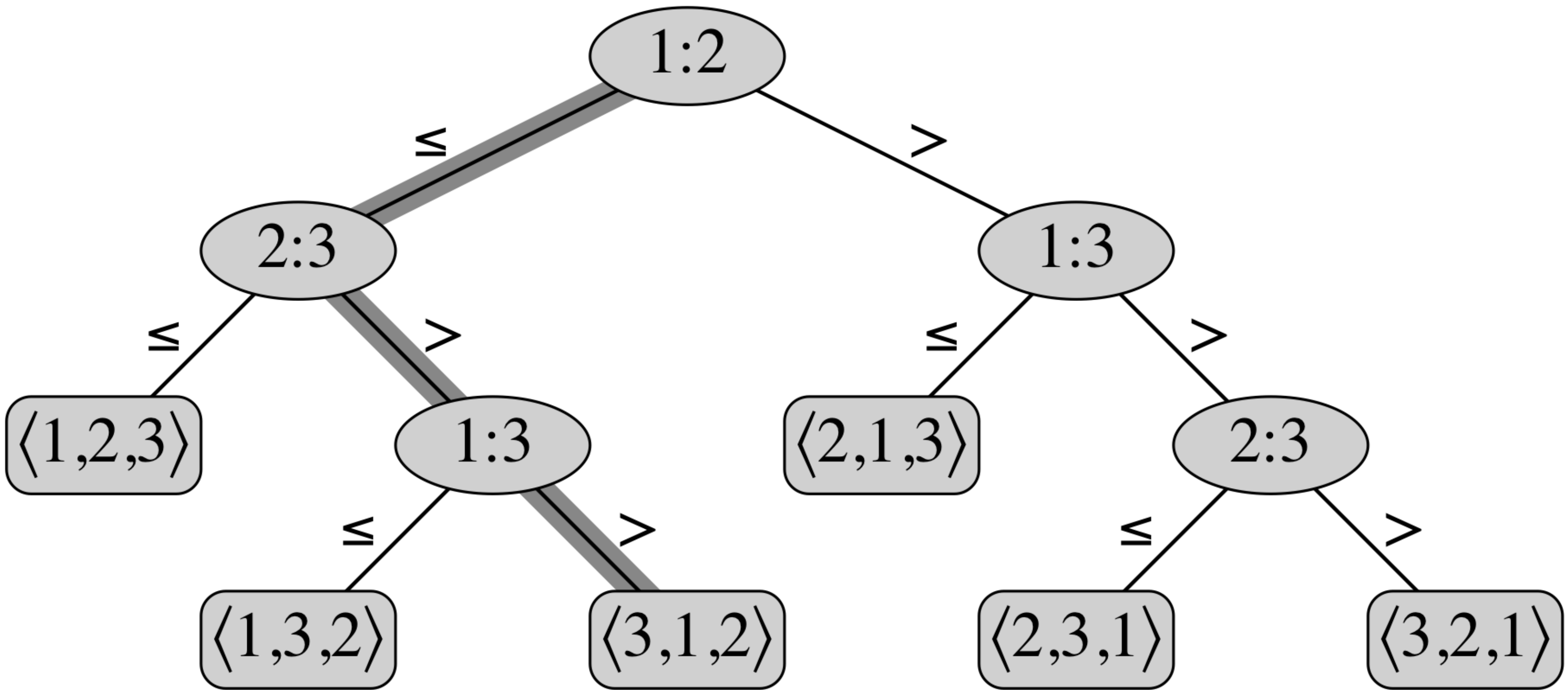
Lower bounds for comparison sorts

- $\Omega(n)$ to examine all the input
- All sorts seen so far are $\Omega(n \lg n)$
- We will show that all comparison sorts must be $\Omega(n \lg n)$

Decision tree

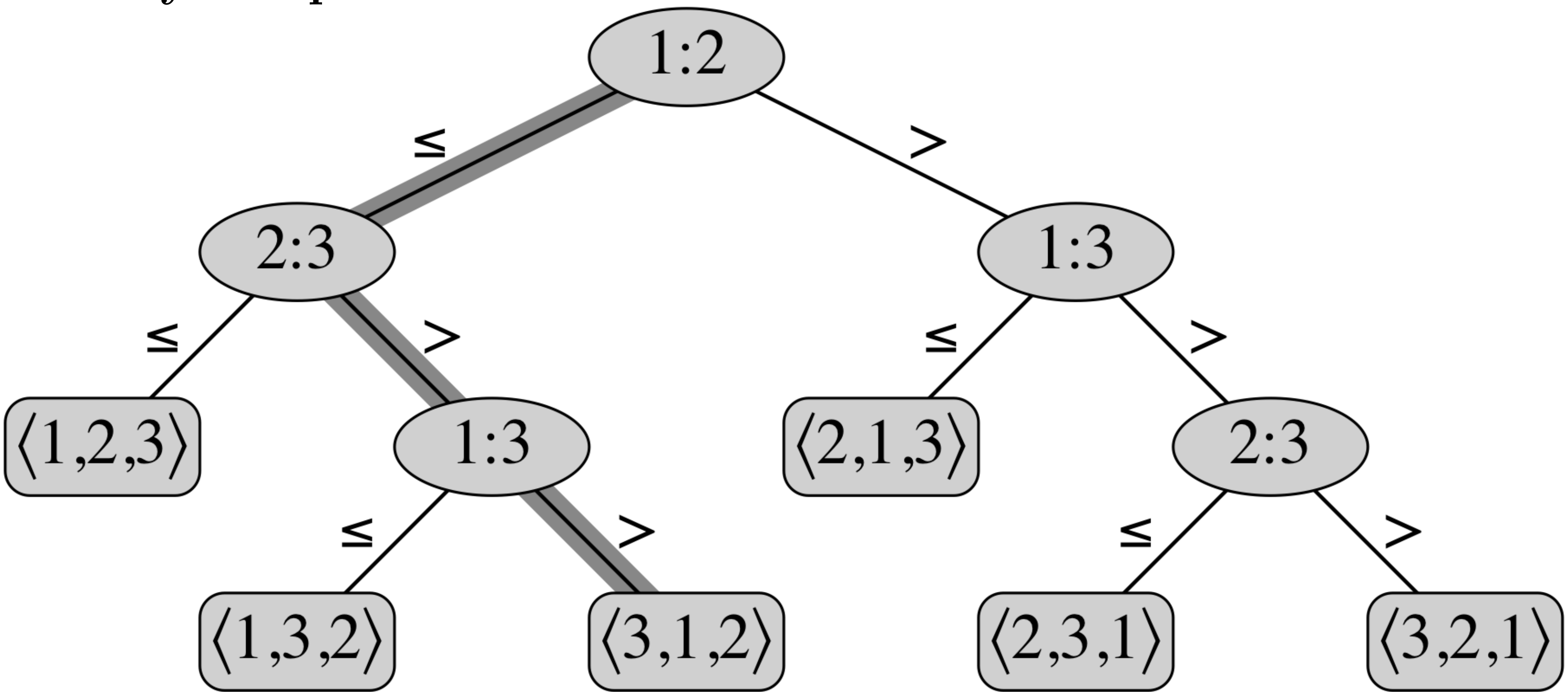
- Abstraction of any comparison sort
- Represents comparisons made by
  - a specific sorting algorithm
  - on inputs of a given size
- Abstracts away everything else: control and data movement.
- We're counting *only* comparisons.

Insertion sort on three elements



- Internal nodes labeled by comparisons (original positions).
- Leaf nodes labeled by permutation of order from original.
- Number of leaves $\geq n!$.

For any comparison sort



- 1 tree for each n
- View the tree as if the algorithm splits in two at each node.
- The tree models all possible execution traces.

What is the longest path from root to leaf?

- Depends on the algorithm.
- Insertion sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \lg n)$

Lemma: any binary tree of height h has $\leq 2^h$ leaves.

- ℓ =# of leaves
- h = height
- then $\ell \leq 2^h$

Proof by induction on h:

Base: $h = 0$. Tree is just one node, which is a leaf. $1 \leq 2^h$.

Inductive step: Assume true for $h-1$. Extend tree with as many new leaves as possible. Each leaf becomes the parent of two new leaves.

$$\begin{aligned}
\#\text{ of leaves for } h &= 2(\#\text{ of leaves for } h-1) \\
&\leq 2(2^{h-1}) \\
&= 2^h
\end{aligned}$$

Theorem: any decision tree that sorts n elements has height $\Omega(n \lg n)$

- $\ell \geq n!$
- $n! \leq \ell \leq 2^h$
- $h \geq \lg(n!)$
- Sterling's approximation: $n! > (n/e)^n$
- Therefore:

$$\begin{aligned} h &\geq \lg(n!) \\ &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Sorting in linear time

- Impossible with any comparison sort.
- **Counting sort**
  - Key assumption: numbers to be sorted are integers in $\{0, \ldots, k\}$.

Input: $A[1..n]$ where $A[j] \in \{0, ..., k\}$

Output: $B[1..n]$, sorted.

Auxiliary storage: $C[0..k]$

Counting sort example

COUNTING-SORT(A, B, n, k)

```
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
for j = 1 to n
        C[A[j]] = C[A[j]] + 1
for i = 1 to k
        C[i] = C[i] + C[i - 1]
for j = n downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

A:

2_1	5_1	3_1	0_1	2_2	3_2	0_2	3_3

After second **for** loop:

C:

2	0	2	3	0	1

After third **for** loop:

C:

2	2	4	7	7	8

Counting sort example

```
COUNTING-SORT(A, B, n, k)
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
for j = 1 to n
        C[A[j]] = C[A[j]] + 1
for i = 1 to k
        C[i] = C[i] + C[i − 1]
for j = n downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
```

A:

2_1	5_1	3_1	0_1	2_2	3_2	0_2	3_3

C:

2	2	4	7	7	8

B:

						3_3	
	0_2					3_3	
	0_2				3_2	3_3	
	0_2		2_2		3_2	3_3	
0_1	0_2		2_2		3_2	3_3	
0_1	0_2		2_2	3_1	3_2	3_3	
0_1	0_2		2_2	3_1	3_2	3_3	5_1
0_1	0_2	2_1	2_2	3_1	3_2	3_3	5_1

Counting sort is **stable**:

- Keys with the same value appear in the same order in output as in input.

Counting sort analysis

COUNTING-SORT(A, B, n, k)

let $C[0..k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** n
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

- $\Theta(n + k)$
  - which is $\Theta(n)$ if $k = O(n)$.
- How big a k is practical?
  - 32-bit values? No.
  - 16-bit? Probably not.
  - 8-bit? Maybe, depending on n.
  - 4-bit? Unless n is really small.

- Counting sort will be used in radix sort.

Radix sort example

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix sort

- IBM in early 20th century.
- Punch card sorting machines only sorted on one column.
- Humans would reload the cards and change the column.
- Human-machine cyborg algorithm!

Key idea:

Sort *least* significant digits first.

Radix sort

RADIX-SORT(A, d)

```
for i = 1 to d
        use a stable sort to sort A on digit i
```

Radix sort correctness

- Induction on number of passes.
- Assume digits $1, \ldots, i-1$ are sorted.
- Show that a stable sort on i leaves $1, \ldots, i-1$ sorted:
  - If 2 digits in position i are different,
    - ordering by i is correct and positions $1, \ldots, i-1$ are irrelevant.
  - If 2 digits in position i are equal,
    - numbers are already sorted by inductive hypothesis. Stable sort leaves them that way.

Radix sort analysis

Assume we use counting sort on each digit.

- $\Theta(n+k)$ per digit
- d digits
- $\Theta(d(n+k))$ total
- If $k = O(n)$, time $= \Theta(dn)$.

Radix sort: How to break each key into digits?

- n words
- b bits/word
- Break into r-bit digits. $d = \lceil b/r \rceil$
- Use counting sort, $k = 2^r - 1$.

  Example: 32-bit words, 8-bit digits.

$$b = 32 \qquad\qquad r = 8$$
$$d = \lceil 32/8 \rceil = 4 \qquad\qquad k = 2^8 - 1 = 255$$

- Time $= \Theta\left(\frac{b}{r}(n + 2^r)\right)$

How to choose r?

- Time $= \Theta\left(\frac{b}{r}(n + 2^r)\right)$
- Balance b/r and $n + 2^r$.
- Choosing $r \approx \lg n$ gives

$$\Theta\left(\frac{b}{\lg n}(n + n)\right) = \Theta(bn/\lg n)$$

- If we choose $r < \lg n$ then $b/r > b/\lg n$ and $n + 2^r$ doesn't improve.
- If we choose $r > \lg n$ then $n + 2^r$ term gets big.
- Sort 2^{16} 32-bit numbers, use $r = \lg 2^{16} = 16$ bits. $\lceil b/r \rceil = 2$ passes.

Compare radix to merge and quick

- 1 million (2^{20}) 32-bit integers.
- Radix sort: $\lceil 32/20 \rceil = 2$ passes.
- Merge/quick: $\lg n = 20$ passes.
- Each radix "pass" is 2 passes:
  - one to take census
  - one to move data

How does radix sort violate the $\Omega(n \lg n)$ speed limit?

- Counting sort allows us to gain information about keys
  - other than by directly comparing 2 keys.
- Used keys as array indices,
  - thus getting far more information out of each key.
  - branching factor of the decision tree is k