CS3230 Lecture 5(a)

School of Computing

"Lower Bound for Sorting, Linear-Time Sorting"
"Order Statistics, and Linear Time OS"

- **Lecture Topics and Readings**
  - Lower Bound for Sorting [CLRS]-C8.1
  - Linear Time Sorting Algorithms [CLRS]-C8.2,8.3

Lower Bound for Sorting,
Optimal Sorting,
Thinking outside the Box,
Busting the Lower Bound

CS3230 Lecture 5(b)

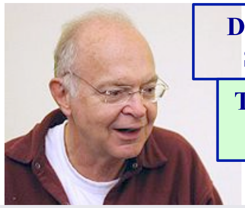
School of Computing

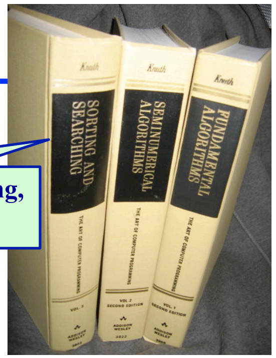
"Lower Bound for Sorting, Linear-Time Sorting"
"Order Statistics, and Linear Time OS"

- **Lecture Topics and Readings**
  - Order Statistics, Min, Max, Min-Max
  - Randomized Divide-and-Conquer [CLRS]-C9
  - Order Statistics in Linear Time [CLRS]-C9

Recursive algorithms are elegant!
Balancing leads to efficient algorithms

Sorting and Searching





Don Knuth, Stanford

The Art of Computing Programming, Vol 3, "*Sorting and Searching*"

Shen Chuan Qi (2011 SG IOI Team, now at Stanford) attending "Christmas Tree Lecture" given by Don Knuth, around Xmas 2013.

Topic: Planar Graphs and Ternary Trees

Search: Don Knuth, Christmas Tree Lectures, December 2013

Thank you.

Q & A

School of Computing

CS3230 Lecture 5(a)

NUS National University of Singapore
School of Computing

"Lower Bound for Sorting, Linear-Time Sorting"
"Order Statistics, and Linear Time OS"

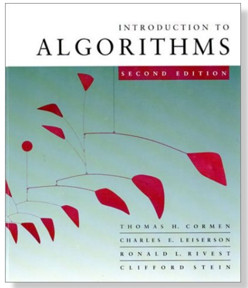
❑ Lecture Topics and Readings

- ❖ Lower Bound for Sorting [CLRS]-C8.1
- ❖ Linear Time Sorting Algorithms [CLRS]-C8.2,8.3

Lower Bound for Sorting,
Optimal Sorting,
Thinking outside the Box,
Busting the Lower Bound

Introduction to Algorithms

6.046J/18.401J



LECTURE 5

Sorting Lower Bounds
- Decision trees

Linear-Time Sorting
- Counting sort
- Radix sort

Appendix: Punched cards

Prof. Erik Demaine

How fast can we sort?

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

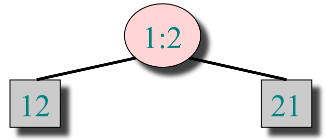
The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.

To sort 2 numbers

Sort $\langle a_1, a_2 \rangle$



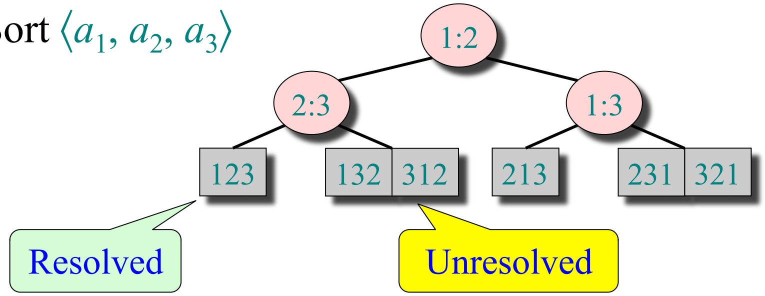
Just one comparison is needed. Trivial

Decision Tree Model

Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Can we sort 3 numbers with only 2 comparisons?

Sort $\langle a_1, a_2, a_3 \rangle$

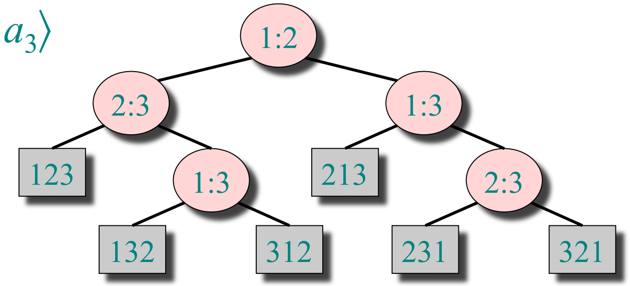


Some (2 of 6) input cases are resolved.
Some (4 of 6) input cases are not fully resolved.

- Need one more comparison to resolve all input cases.

Sorting 3 numbers with 3 comparisons.

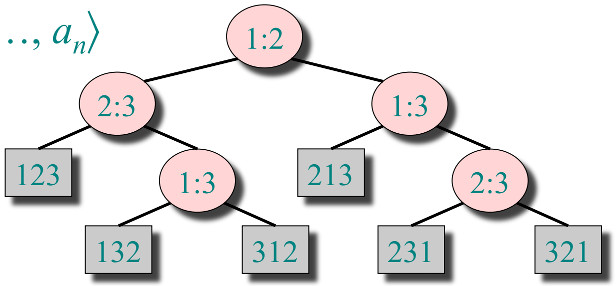
Sort $\langle a_1, a_2, a_3 \rangle$



All (6 of 6) input cases are resolved.
There are 6 = 3! possible input cases, all resolved.

Decision-tree example

Sort $\langle a_1, a_2, \ldots, a_n \rangle$

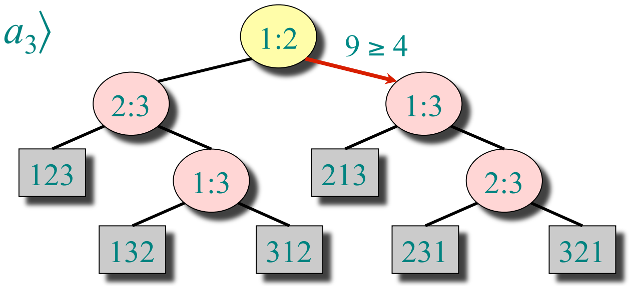


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

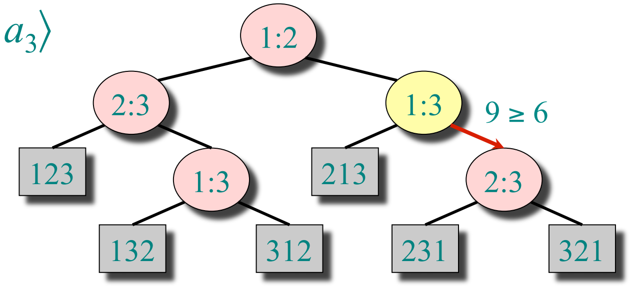


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

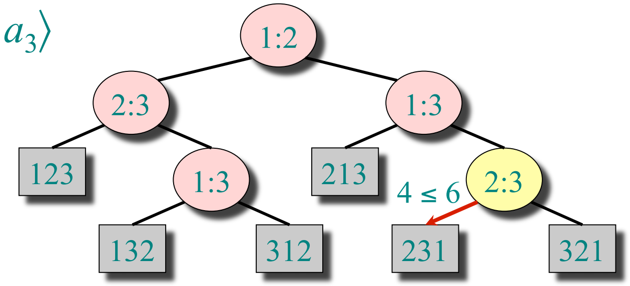


Each internal node is labeled i:j for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

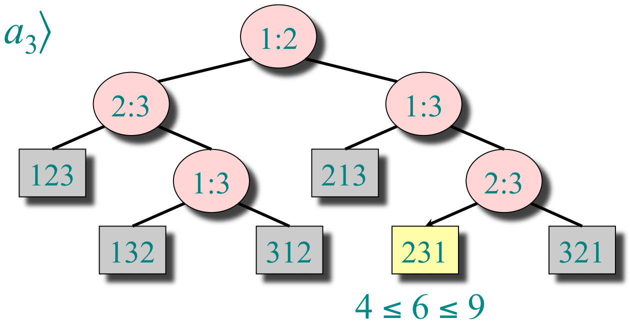


Each internal node is labeled i:j for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$ has been established.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

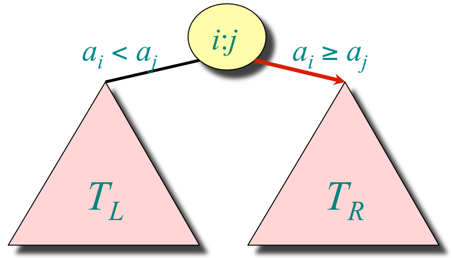
$$\begin{aligned} \therefore\ h &\geq \lg(n!) && \text{(lg is mono. increasing)} \\ &\geq \lg((n/e)^n) && \text{(Stirling's formula)} \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \quad \square \end{aligned}$$

(*Modified slightly by LeongHW, 2013/14*)

Decision-tree for 4 numbers?

Sort $\langle a_1, a_2, a_3, a_4 \rangle = \langle 9, 4, 6, 7 \rangle$:

Q: How many leaves are there in total?



If we compare a_i and a_j

- Left subtree T_L has all perms with $a_i < a_j$.
- Right subtree T_R has all perms with $a_i \geq a_j$.

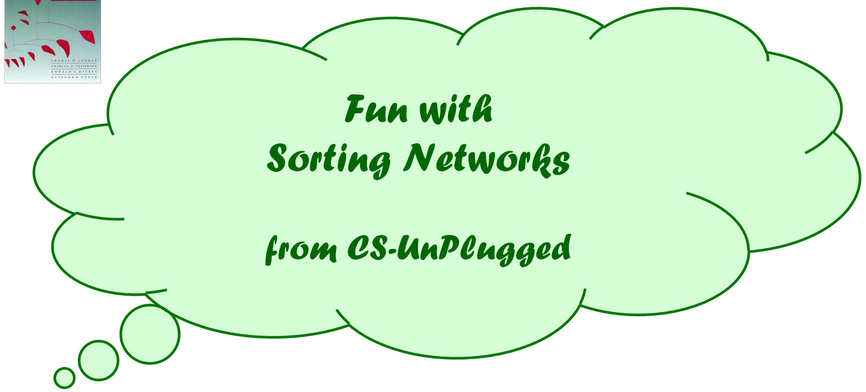
For minimum height, try to balance size of T_L and T_R

(*Modified slightly by LeongHW, 2013/14*)

Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. $\square$

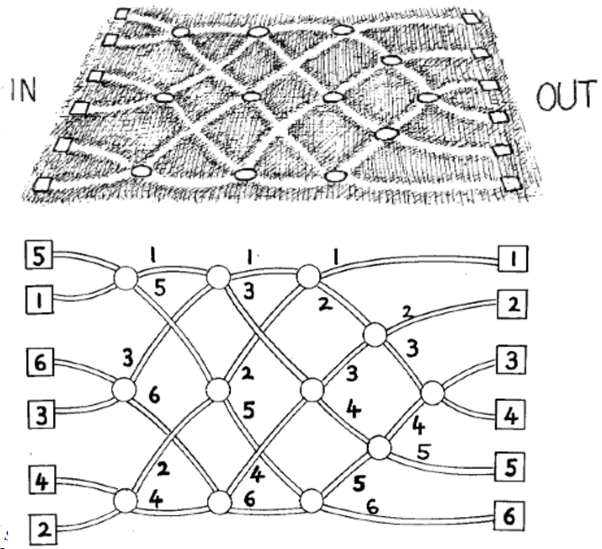
(*Modified slightly by LeongHW, 2013/14*)



Tim Bell, Mike Fellows, Ian Witten, "CS UnPlugged"
http://csunplugged.org/
Youtube: http://www.youtube.com/watch?v=30WcPnvfiKE#t=58 (1:43 min)

(*Modified slightly by LeongHW, 2013/14*)

Sorting networks…



Sorting Network (with Mike Fellows)

Youtube: http://www.youtube.com/watch?v=30WcPnvfiKE#t=58 (1:43 min)

Lessons fr. Sorting Networks

Q1. Build the fastest sorting network for sorting 4 numbers.

Q2. Build the sorting network for *bubble sort algorithm* on 4 numbers.

Q3. Modify the sorting network (for 6 #'s) so that it only *finds the largest*, i.e., moves the largest to the end.

Breaking the (n lg n) Barrier

To Linear Time Sort

Sorting in linear time

Counting sort: No comparisons between elements.

- ***Input***: $A[1\,..\,n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1\,..\,n]$, sorted.
- ***Auxiliary storage***: $C[1\,..\,k]$.

Counting sort

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright$ $C[i] = |\{\text{key} \le i\}|$
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:				

Loop 1

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	0	0	0	0

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	1

B:

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

(Modified slightly by LeongHW, 2013/14)

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	0	1

B:

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

(Modified slightly by LeongHW, 2013/14)

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	1	1

B:

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

(Modified slightly by LeongHW, 2013/14)

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	1	2

B:

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{key} = i\}|$

(Modified slightly by LeongHW, 2013/14)

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

C':

1	1	2	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ $\quad \triangleright\ C[i] = |\{\text{key} \le i\}|$

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

C':

1	1	3	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ $\quad \triangleright\ C[i] = |\{\text{key} \le i\}|$

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

C':

1	1	3	5

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ $\quad \triangleright\ C[i] = |\{\text{key} \le i\}|$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	3	5
C':	1	1	2	5

for $j \leftarrow n$ **downto** 1
 do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

	1	2	3	4
C:	1	1	2	5
C':	1	1	2	4

for $j \leftarrow n$ **downto** 1
 do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	2	4
C':	1	1	1	4

for $j \leftarrow n$ **downto** 1
 do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		4

	1	2	3	4
C:	1	1	1	4
C':	0	1	1	4

for $j \leftarrow n$ **downto** 1
 do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3	4	4

	1	2	3	4
C:	0	1	1	4
C':	0	1	1	3

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

What if we go
for $j \leftarrow 1$ **to** n ?
Will it still sort?

Analysis

$\Theta(k)$ — **for** $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

$\Theta(n)$ — **for** $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ — **for** $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ — **for** $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.

A:	4	1	3	4	3
B:	1	3	3	4	4

Exercise: What other sorts have this property?

Radix sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix ⓘ.)
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

(Modified slightly by LeongHW, 2013/14)

Operation of radix sort

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

(Modified slightly by LeongHW, 2013/14)

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t

7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

(Modified slightly by LeongHW, 2013/14)

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.

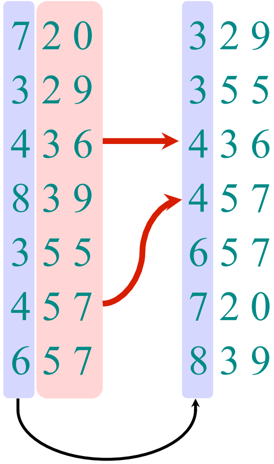
7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

(Modified slightly by LeongHW, 2013/14)

Correctness of radix sort

Induction on digit position

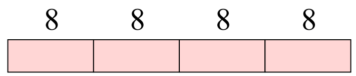
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis (continued)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.

Choosing *r*

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

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(d\,n)$ time.

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \lg 2000 \rceil =$ 11 passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

(Modified slightly by LeongHW, 2013/14)

Appendix: Punched-card technology

- Herman Hollerith (1860-1929)
- Punched cards
- Hollerith's tabulating system
- Operation of the sorter
- Origin of radix sort
- "Modern" IBM card
- Web resources on punched-card technology

Return to last slide viewed.

(Modified slightly by LeongHW, 2013/14)

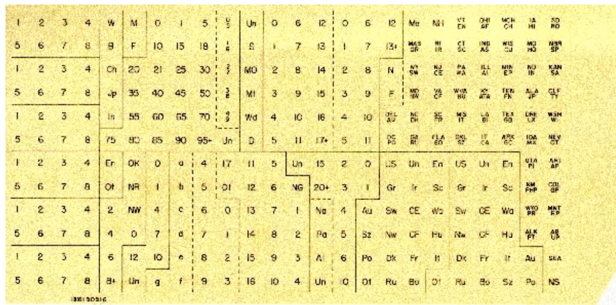
Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

(Modified slightly by LeongHW, 2013/14)

Punched cards

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.



Replica of punch card from the 1900 U.S. census. [Howells 2000]

(Modified slightly by LeongHW, 2013/14)

Hollerith's tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

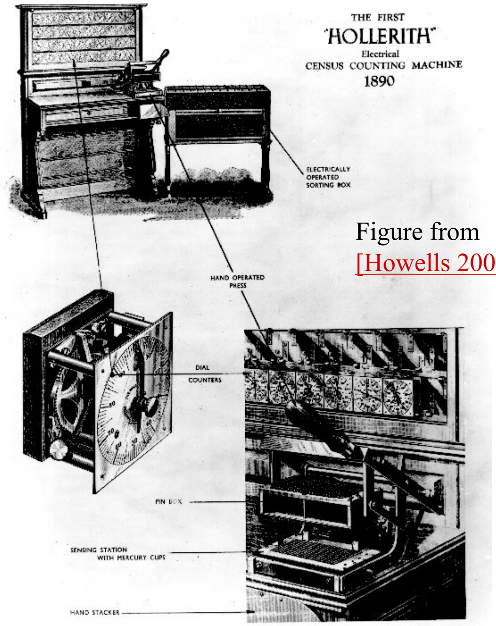
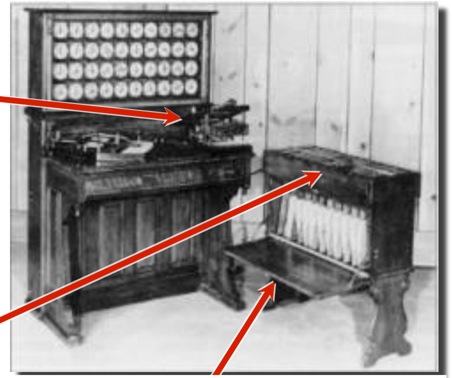


Figure from [Howells 2000].

(*Modified slightly by LeongHW, 2013/14*)

Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.



Hollerith Tabulator, Pantograph, Press, and Sorter

(*Modified slightly by LeongHW, 2013/14*)

Origin of radix sort

Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

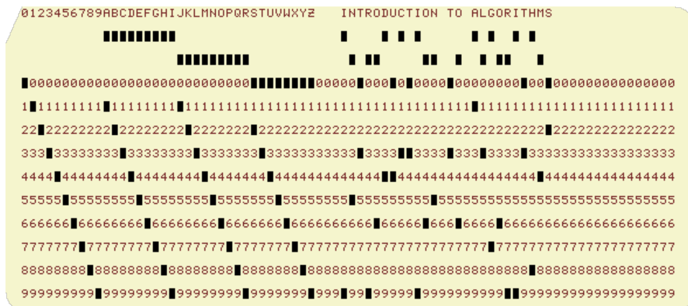
> *"The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards."*

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

(*Modified slightly by LeongHW, 2013/14*)

"Modern" IBM card

- One character per column.



Produced by the WWW Virtual Punch-Card Server.

So, that's why text windows have 80 columns!

(*Modified slightly by LeongHW, 2013/14*)

Web resources on punched-card technology

- Doug Jones's punched card index
- Biography of Herman Hollerith
- The 1890 U.S. Census
- Early history of IBM
- Pictures of Hollerith's inventions
- Holleriths patent application (borrowed from Gordon Bell's CyberMuseum)
- Impact of punched cards on U.S. history

(Modified slightly by LeongHW, 2013/14)

CS3230 Lecture 5(b)

School of Computing

"Lower Bound for Sorting, Linear-Time Sorting"

"Order Statistics, and Linear Time OS"

Lecture Topics and Readings

- **Order Statistics, Min, Max, Min-Max**
- **Randomized Divide-and-Conquer [CLRS]-C9**
- **Order Statistics in Linear Time [CLRS]-C9**

Recursive algorithms are elegant!
Balancing leads to efficient algorithms

Order statistics

Select the ith smallest of n elements (the element with ***rank*** i).

- $i = 1$: ***minimum***;
- $i = n$: ***maximum***;
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: ***median***.

Naive algorithm: Sort and index ith element.

Worst-case running time $= \Theta(n \lg n) + \Theta(1)$
$= \Theta(n \lg n)$,

using merge sort or heapsort (*not* quicksort).

Quick Revision:
We start with the humble Find-Max

Iterative Find-Max algorithm

FIND-MAX $A[1 \, .. \, n]$

1. Let $Max\text{-}sf := A[1]$;
2. **for** $k := 2$ **to** n **do**
3.   **if** $A[k] > Max\text{-}sf$ **then**
4. $Max\text{-}sf := A[k]$
5. **return** $Max\text{-}sf$

If $A=[3, 1, 5, 7]$ Let $C(x, y) = \text{Compare}(x, y)$

$Max\text{-}sf = 3$, $C(1,3)=3$, $C(5,3)=5$, $C(7,5)=7$

Iterative Find-Max algorithm

FIND-MAX $A[1 \, .. \, n]$

1. Let $Max\text{-}sf := A[1]$;
2. **for** $k := 2$ **to** n **do**
3.   **if** $A[k] > Max\text{-}sf$ **then**
4. $Max\text{-}sf := A[k]$
5. **return** $Max\text{-}sf$

A-Problem:
On average, how often is line 4 executed?

Obviously: $T(n) = \Theta(n)$
more precisely: ($n-1$ comparisons)

Making Find-Max recursive

FIND-MAX $A[1 \, .. \, n]$

1. Let $Max\text{-}sf := A[1]$;
2. **for** $k := 2$ **to** n **do**
3.   **if** $A[k] > Max\text{-}sf$ **then**
4. $Max\text{-}sf := A[k]$
5. **return** $Max\text{-}sf$

Compares $A[k]$ with $\max\{ A_1, A_2, \ldots, A_{k-1} \}$

Question: ***Can we turn this into a recursive algorithm?***

Iterative Find-Max algorithm

FIND-MAX $A[1 .. n]$

compares $A[k]$ with $\max\{A_1, A_2, \ldots, A_{k-1}\}$

1. Let *Max-sf* := $A[1]$;
2. **for** $k := 2$ **to** n **do**
3. **if** $A[k] >$ *Max-sf* **then**
4. *Max-sf* := $A[k]$
5. **return** *Max-sf*

When k=7, what is value of *Max-sf*?
Max-sf = max $\{ A[1], A[2], \ldots A[6] \}$

Iterative Find-Max algorithm

FIND-MAX $A[1 .. n]$

compares $A[k]$ with $\max\{A_1, A_2, \ldots, A_{k-1}\}$

1. Let *Max-sf* := $A[1]$;
2. **for** $k := 2$ **to** n **do**
3. **if** $A[k] >$ *Max-sf* **then**
4. *Max-sf* := $A[k]$
5. **return** *Max-sf*

When k=n, what is value of *Max-sf*?…
Max-sf = max $\{ A[1], A[2], \ldots A[n-1] \}$

Recursive Find-Max (FMR)

Recursion Schematic:

$\boxed{\text{FMR}\{A[1..n]\}}$ = max$\{\boxed{\text{FMR}\{A[1..(n-1)]\}}, A[n]\}$

Find-Max-R $A[1 .. n]$

Find-Max-R = FMR

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1 .. n-1]$
3. return Max $\{A[n], M1\}$

(Try the worked example in next few slides yourself)

Recursive Find-Max (FMR)

Find-Max-R $A[1 .. n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1 .. n-1]$
3. return Max $\{A[n], M1\}$

$\boxed{\text{FMR}\{[3,1,5,7]\}}$

If A=[3, 1, 5, 7] }

Max$\{7, \boxed{\text{FMR}\{[3,1,5]\}}\}$

(Finish this worked example in next few slides yourself)

Recursive Find-Max (FMR)

Find-Max-R $A[1\ ..\ n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1\ ..\ n{-}1]$
3. return Max $\{A[n], M1\}$

FMR{[3,1,5,7]}

Max{7, FMR{[3,1,5]} }

Max{5, FMR{[3,1]} }

If A=[3, 1, 5, 7] }

Recursive Find-Max (FMR)

Find-Max-R $A[1\ ..\ n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1\ ..\ n{-}1]$
3. return Max $\{A[n], M1\}$

FMR{[3,1,5,7]}

Max{7, FMR{[3,1,5]} }

Max{5, FMR{[3,1]} }

Max{1, FMR{[3]} }

If A=[3, 1, 5, 7] }

Recursive Find-Max (FMR)

Find-Max-R $A[1\ ..\ n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1\ ..\ n{-}1]$
3. return Max $\{A[n], M1\}$

FMR{[3,1,5,7]}

Max{7, FMR{[3,1,5]} }

Max{5, FMR{[3,1]} }

Max{1, FMR{[3]} }

If A=[3, 1, 5, 7] }

Recursive Find-Max (FMR)

Find-Max-R $A[1\ ..\ n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1\ ..\ n{-}1]$
3. return Max $\{A[n], M1\}$

FMR{[3,1,5,7]}

Max{7, FMR{[3,1,5]} }

Max{5, FMR{[3,1]} }

Max{1, 3 } **= 3**

If A=[3, 1, 5, 7] }

Recursive Find-Max (FMR)

Find-Max-R $A[1 \mathrel{..} n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1 \mathrel{..} n-1]$
3. return Max $\{A[n], M1\}$

If A=[3, 1, 5, 7] }

FMR{[3,1,5,7]}

Max{7, FMR{[3,1,5]} }

Max{5, 3 } = 5

Max{1, 3 } = 3

Recursive Find-Max (FMR)

Find-Max-R $A[1 \mathrel{..} n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1 \mathrel{..} n-1]$
3. return Max $\{A[n], M1\}$

If A=[3, 1, 5, 7] }

FMR{[3,1,5,7]}

Max{7, 5 } = 7

Max{5, 3 } = 5

Max{1, 3 } = 3

Recursive Find-Max (FMR)

Find-Max-R $A[1 \mathrel{..} n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1 \mathrel{..} n-1]$
3. return Max $\{A[n], M1\}$

If A=[3, 1, 5, 7] }

7

Max{7, 5 } = 7

Max{5, 3 } = 5

Max{1, 3 } = 3

Recursive Find-Max (FMR)

Find-Max-R $A[1 \mathrel{..} n]$

1. If $n = 1$, return $A[1]$
2. $M1$:= Find-Max-R $A[1 \mathrel{..} n-1]$
3. return Max $\{A[n], M1\}$

If A=[3, 1, 5, 7] }

7

Max{7, 5 } = 7

Max{5, 3 } = 5

Max{1, 3 } = 3

FMR does exactly the ***same computations*** as Find-Max!

Find-Max and Find-Max-R

Find-Max-R does exactly
the ***same computations*** as Find-Max!

Have the same asymptotic
$\Theta(n)$ worst-case time.

Let's explore the
Recursion Schematic
some more

Recursion Schematics

Recursion Schematic:

$$\boxed{\mathrm{FMR}\{A[1..n]\}} = \max\{\boxed{\mathrm{FMR}\{A[1..(n-1)]\}},\ A[n]\ \}$$

Recursion Schematics

Recursion Schematic:

$$\boxed{\mathrm{FMR}\{A[1..n]\}} = \max\{\underbrace{\boxed{\mathrm{FMR}\{A[1..(n-1)]\}}}_{(n-1)},\ \underbrace{A[n]}_{1}\ \}$$

Extreme imbalance

Recursion Schematics

Recursion Schematic:

$\boxed{\text{FMR}\{A[1..n]\}} = \max\{\boxed{\text{FMR}\{A[1..(n-1)]\}}, A[n]\}$

$(n-1)$ 1

Extreme imbalance

Balanced Recursion Schematic:

$\text{BFM}\{A[1..n]\}$
$= \max\{\text{BFM}\{A[1..\,n/2]\}, \text{BFM}\{A[n/2+1..\,n]\}\}$

Balanced Recursive Find-Max

BFM $A[1\,..\,n]$

1. **if** $n = 1$, done.
2. $M1 :=$ BFM $A[\,1\,..\,\lceil n/2\rceil\,]$
   $M2 :=$ BFM $A[\,\lceil n/2\rceil{+}1\,..\,n\,]$.
3. **return** max $\{M2, M2\}$

Don't this remind you of Merge-Sort ?

Recall: Merge sort

MERGE-SORT $A[1\,..\,n]$

1. If $n = 1$, done.
2. MERGE-SORT $A[\,1\,..\,\lceil n/2\rceil\,]$
   MERGE-SORT $A[\,\lceil n/2\rceil{+}1\,..\,n\,]$
3. "***Merge***" the 2 sorted lists.

Balanced Recursive Find-Max

BFM $A[1\,..\,n]$

1. **if** $n = 1$, done.
2. $M1 :=$ BFM $A[\,1\,..\,\lceil n/2\rceil\,]$
   $M2 :=$ BFM $A[\,\lceil n/2\rceil{+}1\,..\,n\,]$.
3. **return** max $\{M2, M2\}$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(1) \text{ if } n > 1. \end{cases}$$

Balanced Recursive Find-Max

BFM $A[1 \, . \, . \, n]$

1. **if** $n = 1$, done.
2. $M1$:= BFM $A[\, 1 \, . \, . \, \lceil n/2 \rceil \,]$
   $M2$:= BFM $A[\, \lceil n/2 \rceil {+} 1 \, . \, . \, n \,]$.
3. **return** max $\{M2, M2\}$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(1) \text{ if } n > 1. \end{cases}$$

BFM: $a = 2$, $b = 2$ $\Rightarrow$ $n^{\log_b a} = n^{\log_2 2} = n$
$f(n) = O(n^{1-\varepsilon})$ for $\varepsilon = 0.5$ $\Rightarrow$ CASE 1: $T(n) = O(n)$.

Recursion tree

Solve $T(n) = 2T(n/2) + 1$.

Recursion tree

Solve $T(n) = 2T(n/2) + 1$.

$T(n)$

Recursion tree

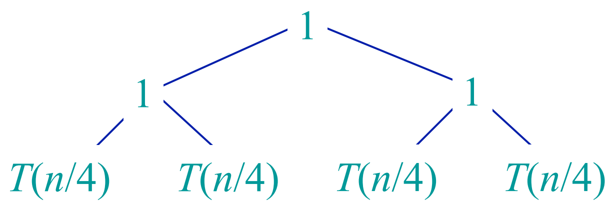
Solve $T(n) = 2T(n/2) + 1$.

1

$T(n/2)$ $T(n/2)$

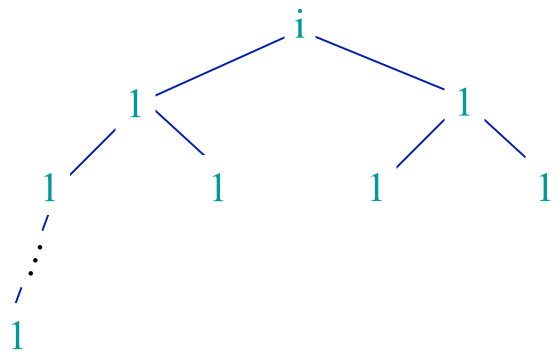
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



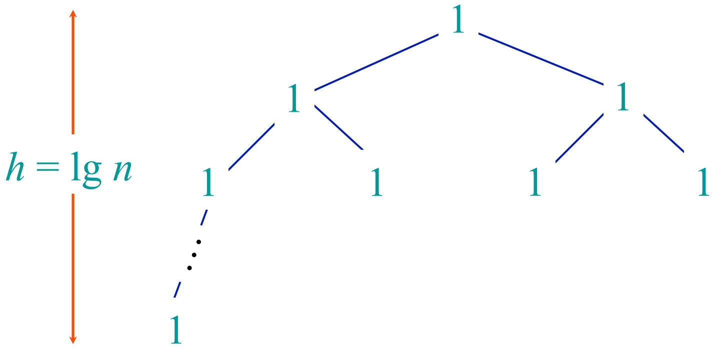
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



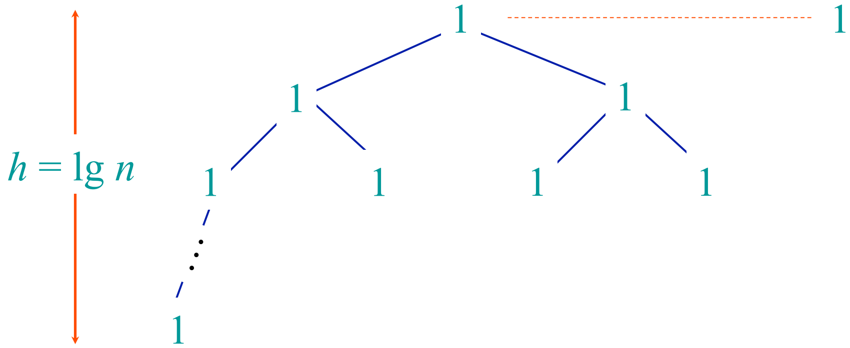
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



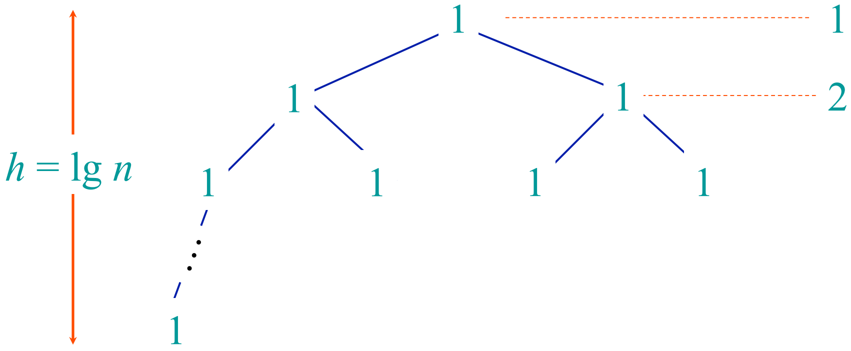
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



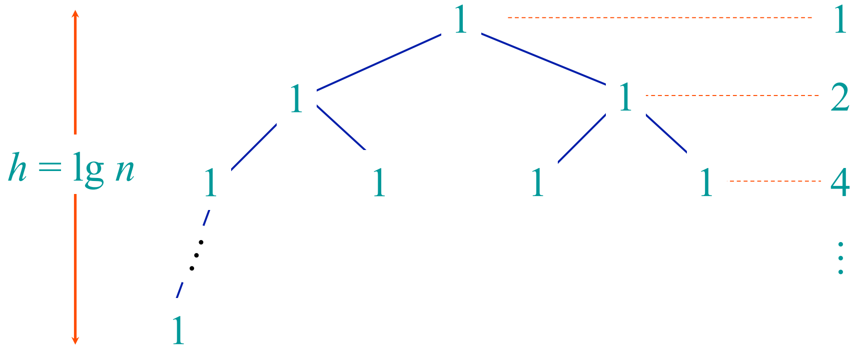
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



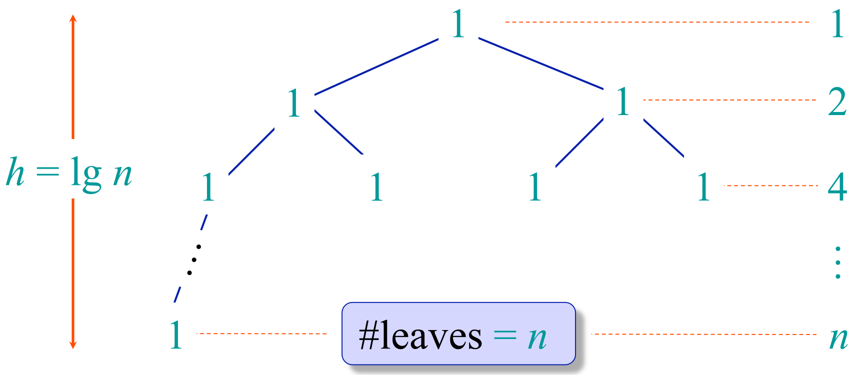
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



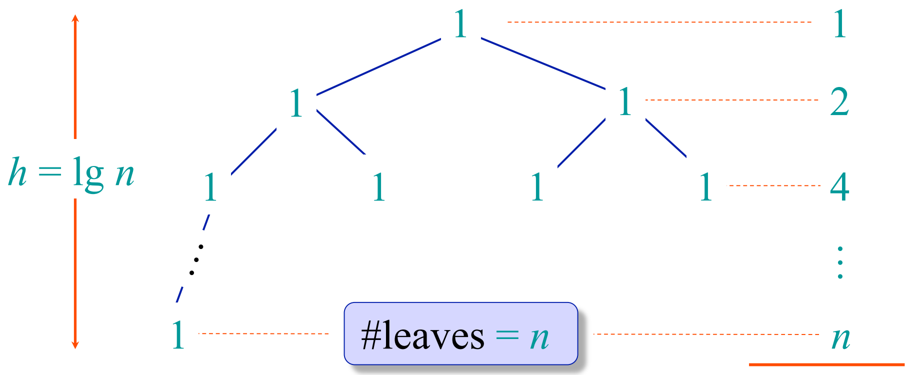
Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



Recursion tree

Solve $T(n) = 2T(n/2) + 1$.



Total $= \Theta(n)$

How to do the sum?

Recall

$$\sum_{k=0}^{\lg n} 2^k = 1 + 2 + 2^2 + \cdots + 2^h \le 2n$$

Or equivalently,

$$\sum_{k=0}^{\lg n} n/2^k = \left(n + n/2 + n/2^2 + \cdots + n/2^{\lg n}\right)$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{\lg n}}\right) \le 2n$$

Isn't this the game
in HW0-S1(b)?

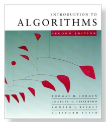
Modification of HW0-S1(b)

Algorithm (from HW0-S1(b))

1. Each student k stand up, given number A[k]
2. Pair up with someone standing, the one with *smaller number* sits down
3. Go back to Step 2

What's the similarity?

What the difference?



How about finding
Max-and-Min

Thank you.

Q & A

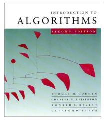
CS3230 Lecture 5(b)

"Lower Bound for Sorting, Linear-Time Sorting"

"Order Statistics, and Linear Time OS"

- **Lecture Topics and Readings**
  - **Order Statistics, Max, Min-Max** [CLRS]-C9.1
  - **Randomized Divide-and-Conquer** [CLRS]-C9.2
  - **Order Statistics in Linear Time** [CLRS]-C9.3

Recursive algorithms are elegant!
Balancing leads to efficient algorithms



Order statistics

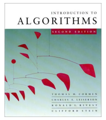
Select the ith smallest of n elements (the element with ***rank*** i).

- $i = 1$: ***minimum***;
- $i = n$: ***maximum***;
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: ***median***.

Naive algorithm: Sort and index ith element.

Worst-case running time $= \Theta(n \lg n) + \Theta(1)$
$= \Theta(n \lg n)$,

using merge sort or heapsort (*not* quicksort).



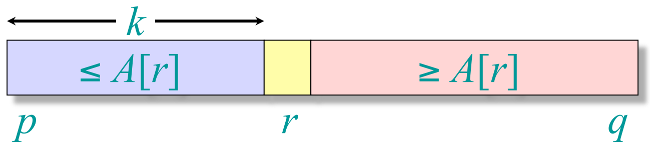
Randomized Divide-and-Conquer Algorithm

Modified from Quicksort

Also by **C. A. R. (Tony) Hoare**, who invented Quicksort.

Randomized divide-and-conquer algorithm

RAND-SELECT(A, p, q, i) ▹ ith smallest of $A[p \,..\, q]$
  if $p = q$ **then return** $A[p]$
  $r \leftarrow$ RAND-PARTITION(A, p, q)
  $k \leftarrow r - p + 1$ ▹ $k = \text{rank}(A[r])$
  if $i = k$ **then return** $A[r]$
  if $i < k$
    then return RAND-SELECT$(A, p, r - 1, i)$
    else return RAND-SELECT$(A, r + 1, q, i - k)$



Example

Select the $i = 7$th smallest:

6	10	13	5	8	3	2	11

$i = 7$

pivot

Partition:

2	5	3	6	8	13	10	11

$k = 4$

Select the $7 - 4 = 3$rd smallest recursively.

Intuition for analysis

(All our analyses today assume that all elements are distinct.)

Lucky:

$T(n) = T(9n/10) + \Theta(n)$ $\quad n^{\log_{10/9} 1} = n^0 = 1$
$\quad = \Theta(n)$ $\quad$ CASE 3

Unlucky:

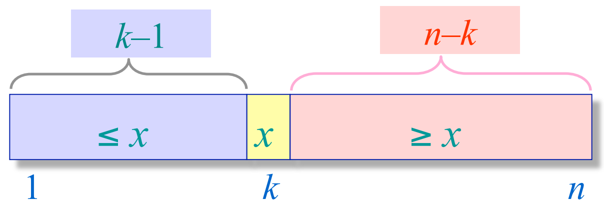
$T(n) = T(n - 1) + \Theta(n)$ $\quad$ arithmetic series
$\quad = \Theta(n^2)$

Worse than sorting!

Analysis of RAND-SELECT

Let $T(n)$ = the *expected worst-case* time taken by RAND-SELECT on input of size n.

If pivot x ends up in position k,
then $T(n) = \max\{T(k-1), T(n-k)\} + (n+1)$



Prob(pivot is at pos k) = $1/n$ *for all k*

Analysis of RAND-SELECT

Then, for expected worst-case, we have

$$T(n) = \begin{cases} \max\{T(0), T(n-1)\} + (n+1) & \text{if } 0 : n-1 \text{ split} \\ \max\{T(1), T(n-2)\} + (n+1) & \text{if } 1 : n-2 \text{ split} \\ \max\{T(2), T(n-3)\} + (n+1) & \text{if } 2 : n-3 \text{ split} \\ \vdots & \vdots \\ \max\{T(n-2), T(1)\} + (n+1) & \text{if } n-2 : 1 \text{ split} \\ \max\{T(n-1), T(0)\} + (n+1) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

Prob(pivot is at pos k) = $1/n$ *for all k*

Analysis of RAND-SELECT

Then, we have the following recurrence:

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[\max\{T(k-1),\, T(n-k)\} + (n+1)\right]$$

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} T(k) + (n+1)$$

Upper terms appear twice.

$$T(n) \le \frac{2}{n}\Big(T(\lfloor \tfrac{n}{2} \rfloor) + T(\lfloor \tfrac{n}{2} \rfloor + 1) + \ldots + T(n-1)\Big) + (n+1)$$

Substitution Method

- We will use "Substitution Method" to prove that $T(n) \le Cn$, for some C.
- **Idea in Substitution Method:**
  1. Guess the form of the solution;
  2. Use mathematical induction to prove it and find the constants
- **Optional for CS3230 (Spring 2014)**
  - **See [CLRS]-C4.3 pp.83-87 for details**

Using the Substitution Method

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} T(k) + (n+1)$$

Step 1 : Guess $T(n) \le Cn$ for constant $C > 0$.

Step 2: Prove $T(n) \le Cn$ using MI, and find the constant C

Using the Substitution Method

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} T(k) + (n+1)$$

Prove: $T(n) \le Cn$ for constant $C > 0$.

(Use mathematical induction.)

- **Base Case:** The constant C can be chosen large enough so that $T(n) \le Cn$ for the base cases (n very small).

Later, need fact: $\sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} k \le \frac{3}{8} n^2$ (exercise).

Using the Substitution Method

- **Induction Step:**

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} Ck + (n+1)$$

Substitute inductive hypothesis.
Namely, $T(k) \le Ck$ for all $k < n$.

Using the Substitution Method

- **Induction Step:**

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} Ck + (n+1)$$

$$\le \frac{2C}{n}\left(\frac{3}{8}n^2\right) + (n+1) \qquad \text{(Use fact)}$$

Using the Substitution Method

- **Induction Step:**

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} Ck + (n+1)$$

$$\le \frac{2C}{n}\left(\frac{3}{8}n^2\right) + (n+1)$$

$$= Cn - \left(\frac{Cn}{4} - (n+1)\right)$$

Express as ***desired – residual***.

Using the Substitution Method

- **Induction Step:**

$$T(n) \le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} Ck + (n+1)$$

$$\le \frac{2C}{n}\left(\frac{3}{8}n^2\right) + (n+1)$$

$$= Cn - \left(\frac{Cn}{4} - (n+1)\right)$$

$$\le Cn \qquad \text{(end of induction proof)}$$

When C=5, then
$(Cn/4 - (n+1))$
$= (n/4 - 1) \ge 0$ for $n \ge 4$.

Choose C=5, $n_0 = 4$,
then residual term ≥ 0 for $n > n_0$.

Summary of randomized order-statistic selection

- Works fast: linear expected time.
- Excellent algorithm in practice.
- But, the worst case is ***very*** bad: $\Theta(n^2)$.

Q. Is there an algorithm that runs in linear time in the worst case?

A. Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973].

IDEA: Generate a good pivot recursively.

Worst-case Linear-Time Order Statistic Algorithm

M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan,
"Time Bounds for Selection," Journal of Computer and System Sciences, (Aug 1973), **7** (4): 448–461. doi:10.1016/S0022-0000(73)80033-9

Why is CS3230 FUN?

- "Meet" many CS celebrities

(1972) (1974) **(1978)** **(1980)** (1982)

(1985) (1986) **(1986)** **(1995)** **(2002)**

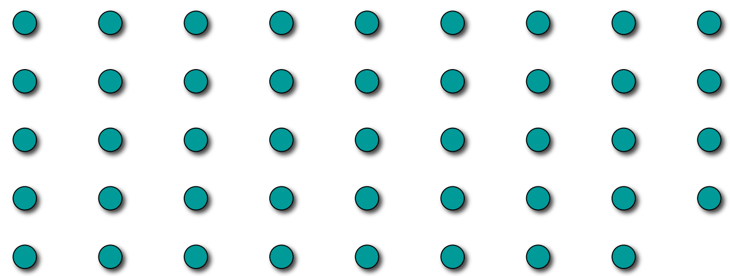
Worst-case linear-time order statistics

SELECT(i, n)

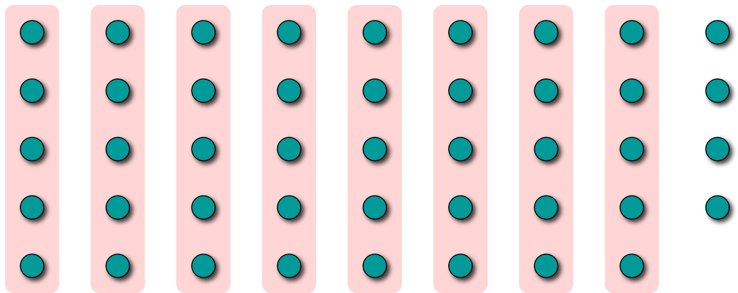
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let $k = \text{rank}(x)$.
4. **if** $i = k$ **then return** x
   elseif $i < k$
   then recursively SELECT the ith smallest element in the lower part
   else recursively SELECT the $(i–k)$th smallest element in the upper part

Same as RAND-SELECT

Choosing the pivot

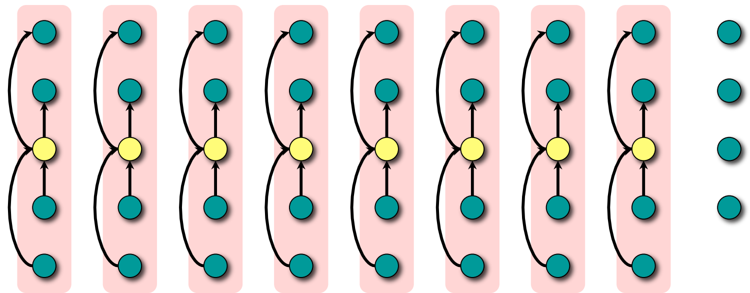


Choosing the pivot

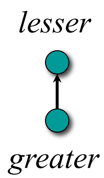


1. Divide the n elements into groups of 5.

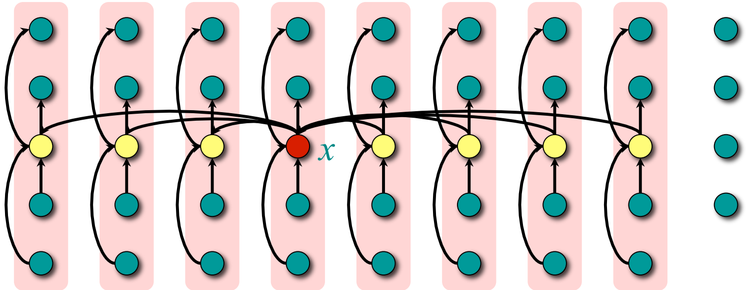
Choosing the pivot



1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

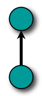


Choosing the pivot



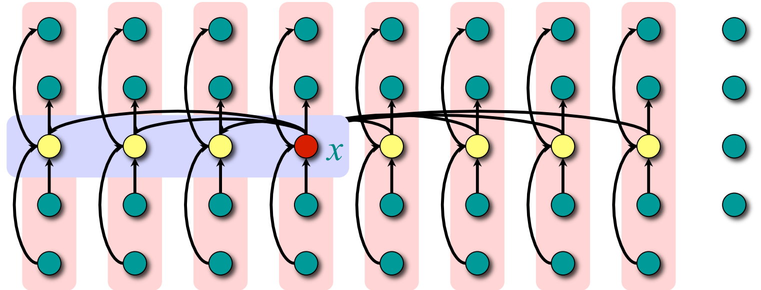
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

lesser



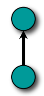
greater

Analysis



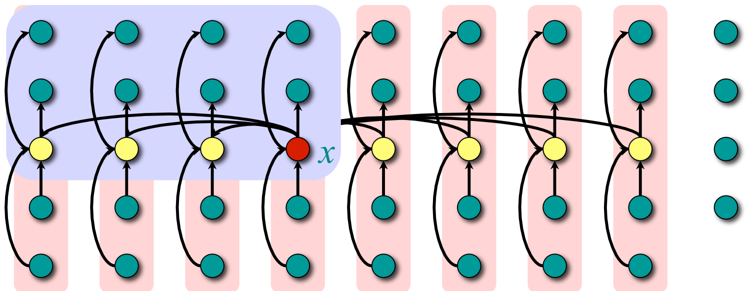
At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

lesser



greater

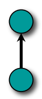
Analysis (Assume all elements are distinct.)



At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

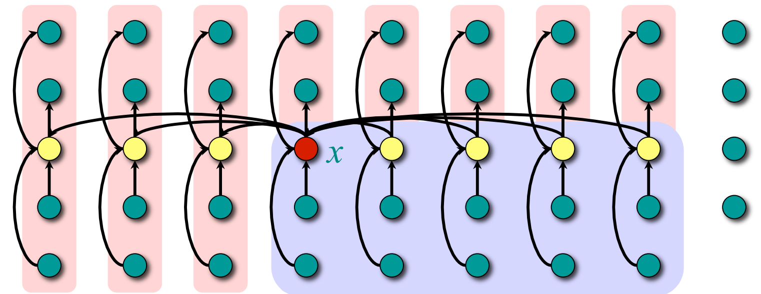
- Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\le x$.

lesser



greater

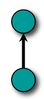
Analysis (Assume all elements are distinct.)



At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\le x$.
- Similarly, at least $3 \lfloor n/10 \rfloor$ elements are $\ge x$.

lesser



greater

Minor simplification

- For $n \geq 50$, we have $3\lfloor n/10 \rfloor \geq n/4$.
- Therefore, for $n \geq 50$ the recursive call to SELECT in Step 4 is executed recursively on $\leq 3n/4$ elements.
- Thus, the recurrence for running time can assume that Step 4 takes time $T(3n/4)$ in the worst case.
- For $n < 50$, we know that the worst-case time is $T(n) = \Theta(1)$.

Developing the recurrence

$T(n)$ SELECT(i, n)

$\Theta(n)$ 1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

$T(n/5)$ 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

$\Theta(n)$ 3. Partition around the pivot x. Let $k = \text{rank}(x)$.

$T(3n/4)$ 4. **if** $i = k$ **then return** x
elseif $i < k$
then recursively SELECT the ith smallest element in the lower part
else recursively SELECT the $(i–k)$th smallest element in the upper part

Solving the recurrence

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

Substitution:
$T(n) \leq cn$

$$\begin{aligned} T(n) &\leq \frac{1}{5}cn + \frac{3}{4}cn + \Theta(n) \\ &= \frac{19}{20}cn + \Theta(n) \\ &= cn - \left(\frac{1}{20}cn - \Theta(n)\right) \\ &\leq cn \quad , \end{aligned}$$

if c is chosen large enough to handle both the $\Theta(n)$ and the initial conditions.

Conclusions

- Since the work at each level of recursion is a constant fraction (19/20) smaller, the work per level is a geometric series dominated by the linear work at the root.
- In practice, this algorithm runs slowly, because the constant in front of n is large.
- The randomized algorithm is far more practical.

Exercise: *Why not divide into groups of 3?*

