CS583 Lecture 04

Jana Kosecka

Linear Time Sorting, Median, Order Statistics

Many slides here are based on E. Demaine , D. Luebke slides

Sorting So Far

- Insertion sort:
- Easy to code

  Fast on small inputs (less than ~50 elements)

  Fast on nearly-sorted inputs

  $O(n^2)$ worst case

  $O(n^2)$ average (equally-likely inputs) case

  $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
- Divide-and-conquer:
  - Split array in half
  - Recursively sort subarrays
  - Linear-time merge step
- O(n lg n) worst case
- Doesn't sort in place

Sorting So Far

- Heap sort:
- Uses the very useful heap data structure
- Complete binary tree
- Heap property: parent key > children's keys
- O(n lg n) worst case
- Sorts in place
- Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
- Divide-and-conquer:
- Partition array into two subarrays, recursively sort
  All of first subarray < all of second subarray
  No merge step needed!
- O(n lg n) average case, fast in practice
- $O(n^2)$ worst case
- Naïve implementation: worst case on sorted input
- Address this with randomized quicksort

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  How do you suppose we'll beat it?
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
- The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
- We have seen sorting algorithms O(n lg n)
- Can we do better ?
- Theorem: all comparison sorts are Ω(n lg n)

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
- A decision tree represents the comparisons made by a comparison sort. Every thing else ignored
  (Draw examples on board)
- *What do the leaves represent?*
- *How many leaves must there be?*

Decision Trees

- Decision trees can model comparison sorts. For a given algorithm:
- One tree for each n
- Tree paths are all possible execution traces
  What's the longest path in a decision tree for insertion sort? For merge sort?
- *What is the asymptotic height of any decision tree for sorting n elements?*
- Answer: $\Omega(n \lg n)$ (now let's prove it…)

Lower Bound - Comparison Sorting

- Thm: Any decision tree that sorts n elements has height $\Omega(n \lg n)$
- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height h?*
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

Lower Bound - Comparison Sorting

- So we have…
  $n! \leq 2^h$
- Taking logarithms:
  $\lg(n!) \leq h$
- Stirling's approximation tells us:

- Thus:

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \lg\left(\frac{n}{e}\right)^n$$

Lower Bound - Comparison Sorting

- So we have

$$
\begin{aligned}
h &\geq \lg\left(\frac{n}{e}\right)^n \\
&= n\lg n - n\lg e \\
&= \Omega(n\lg n)
\end{aligned}
$$

- Thus the minimum height of a decision tree is $\Omega(n \lg n)$

Lower Bound - Comparison Sorts

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But the name of this lecture is "Sorting in linear time"!

  How can we do better than $\Omega(n \lg n)$*?*

Sorting In Linear Time

- Counting sort
- No comparisons between elements!
- ***But***…depends on assumption about the numbers being sorted
- We assume numbers are in the range *1*.. *k*

  The algorithm:

Input: A[1..n], where A[j] $\in \{1, 2, 3, \ldots, k\}$
Output: B[1..n], sorted (notice: not sorting in place)
Also: Array C[1..k] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                        C[A[j]] -= 1;
```

Work through example: A={4 1 3 4 3}, k = 4

Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                        C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

Counting Sort Example

- Loop 1

Array C

4 1 3 4 3

0 0 0 0

Counting Sort Example

- Loop 2

Array C

4 1 3 4 3

0 0 0 0

Counting Sort Example

- Loop 3 Array C Array C’

4 1 3 4 3 1 0 2 2

Counting Sort Example

- Loop 4

Array C	Array B
1 1 3 5	

4 1 3 4 3

Counting Sort

- Total time: $O(n + k)$
  Usually, $k = O(n)$
  Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$!
- No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
- Stable algorithm – the numbers with the same value appear in the same order in the output array as they to in the input array

Notice that this algorithm is *stable*

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large (2^{32} = 4,294,967,296)

- How to sort n integers in range $1 \cdots n^2$ in O(n) time ?
- Counting Sort $O(n+k) = O(n+n^2) = O(n^2)$

Counting Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
- In particular, a *card sorter* that could sort cards into different bins
- Each column can be punched in 12 places
- Decimal digits use 10 places
- Problem: only one column can be sorted on at a time

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

Example: Fig 9.3

Radix Sort

329	720	720	329
458	355	329	355
659	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
- Assume lower-order digits {j: j<i}are sorted
- Show that sorting next digit i leaves array correctly sorted
- If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
- If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

329	720	720	329
458	355	329	355
659	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- Two digits are the same
- Two digits are different

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  Sort n numbers on digits that range from $1..k$
  Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n + k)$, so total time $O(dn+dk)$
- When d is constant and $k=O(n)$, takes $O(n)$ time
- Here the analysis is done on digits ? What about bits ?

- *How many bits in a computer word?*

Radix Sort

- Given n b-bit numbers how long will it take ?
- Suppose each digit is r-bits long $2^r - 1$
- Each pass takes $O(n + 2^r - 1)$
- There are d-passes $O(d(n + 2^r - 1))$ $O\left(\frac{b}{r}(n + 2^r - 1)\right)$
- How to choose *r* to be able to sort in linear time ?

Radix sort

- How to choose r so the running time is still linear

$$O\left(\frac{b}{r}(n+2^r)\right)$$

- If $b = \lg n$ then using
- Radix sort is a good idea
- Since the running time is linear

$$n \approx 2^r$$

$$\lg n \approx r$$

$$O\left(\frac{bn}{\lg n}\right)$$

- Hidden constant factors in the notation can influence
- the choice

Radix Sort

- Given n b-bit numbers how long will it take ?

- Problem: Sort 1 million 64-bit numbers
  Treat as four-16-digit numbers radix 2^{16} numbers
  Can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  Requires approx lg n = 20 operations per number being sorted
- *So why would we ever use anything but radix sort?*

Radix Sort

- In general, radix sort based on counting sort is
  Fast
  Asymptotically fast (i.e., $O(n)$)
  Simple to code
  A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

Review: Comparison Sorts

- Comparison sorts: $O(n \lg n)$ at best
- Model sort with decision tree
- Path down tree = execution trace of algorithm
- Leaves of tree = possible permutations of input
- Tree must have n! leaves, so $O(n \lg n)$ height

Review: Counting Sort

- Counting sort:

  Assumption: input is in the range 1..k

  - Basic idea:

    Count number of elements $k \leq$ each element i

    Use that number to place i in position k of sorted array

  - No comparisons! Runs in time O(n + k)

  Stable sort

  Does not sort in place:

    O(n) array to hold sorted output

    O(k) array for scratch storage

Review: Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                        C[A[j]] -= 1;
```

Summary: Radix Sort

- Radix sort:

  Assumption: input has d digits ranging from 0 to k

  - Basic idea:

    Sort elements by digit starting with *least* significant

    Use a stable sort (like counting sort) for each stage

  - Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time

  Fast! Stable! Simple!

  Doesn't sort in place

Bucket Sort

- Bucket sort

  Assumption: input is n reals from [0, 1)

  - Basic idea:

    Create n linked lists (*buckets*) to divide interval [0,1) into subintervals of size $1/n$
  - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
  - Therefore the expected total time is O(n)

  These ideas will return when we study *hash tables*

Order Statistics

- The ith *order statistic* in a set of n elements is the ith smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the nth order statistic
- The *median* is the $n/2$ order statistic
  If n is even, there are 2 medians
- *How can we calculate order statistics?*
- *What is the running time?*

Order Statistics

- *How many comparisons are needed to find the minimum element in a set? The maximum?*
- *Can we find the minimum and maximum with less than twice the cost?*
- Yes:

  Walk through elements by pairs

  Compare each element in pair to the other

  Compare the largest to maximum, smallest to Minimum

  Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding Order Statistics: The Selection Problem

- A more interesting problem is *selection*: finding the *i*th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

Randomized Selection

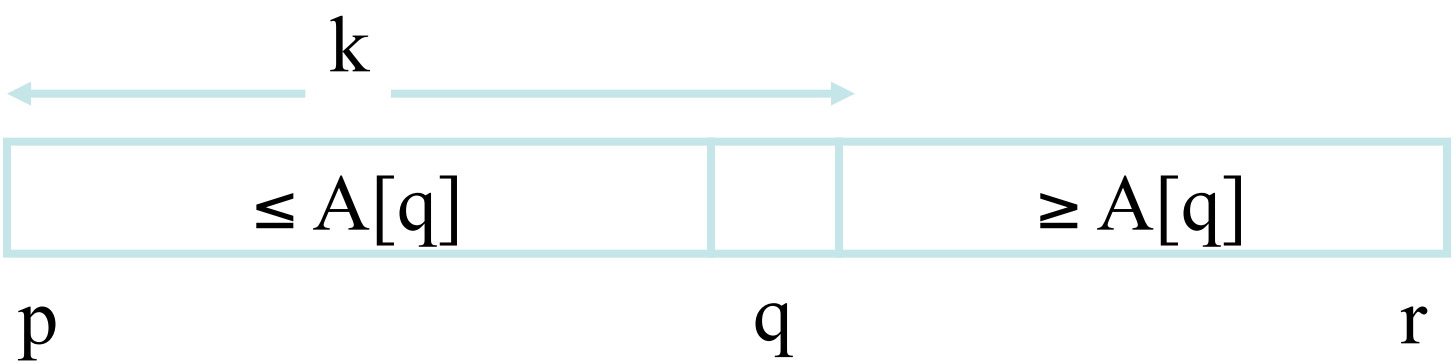
- Key idea: use partition() from quicksort
  But, only need to examine one subarray
  This savings shows up in running time: O(n)
- We will again use a slightly different partition than the book:
  q = RandomizedPartition(A, p, r)

≤ A[q]		≥ A[q]

p q r

Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];   // not in
  book
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Select example

6 10 13 5 8 3 2 11 i=7 looking for i-th largest

↑

2 5 3 6 8 13 10 11 i=3 looking for i-th largest

↑

7-4 = 3

Randomized Selection

- Analyzing **`RandomizedSelect()`**

  Worst case: partition always 0:n-1

  $T(n) = T(n-1) + O(n)$ *= ???*

  $= O(n^2)$ (arithmetic series)

  No better than sorting!

  "Best" case: suppose a 9:1 partition

  $T(n) = T(9n/10) + O(n)$ *= ???*

  $= O(n)$ (Master Theorem, case 3)

  Better than sorting!

  What if this had been a 99:1 split?

Randomized Selection

- For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \quad \leq \quad \frac{1}{n}\sum_{k=0}^{n-1} T\big(\max(k, n-k-1)\big) + \Theta(n)$$

Randomized Selection

- Average case
- For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

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

What happened here?

Let's show that T(n) = O(n) by substitution

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$ The recurrence we started with

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$ Substitute T(n) ≤ cn for T(k)

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$$ "Split" the recurrence

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$$ Expand arithmetic series

$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$ Multiply it out

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$

The recurrence so far

$$= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n)$$

Multiply it out

$$= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n)$$

Subtract c/2

$$= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right)$$

Rearrange the arithmetic

$$\le cn \quad \text{(if c is big enough)}$$

What we set out to prove

$$\frac{cn}{4} \quad \text{dominates } \Theta(\text{n})$$

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:

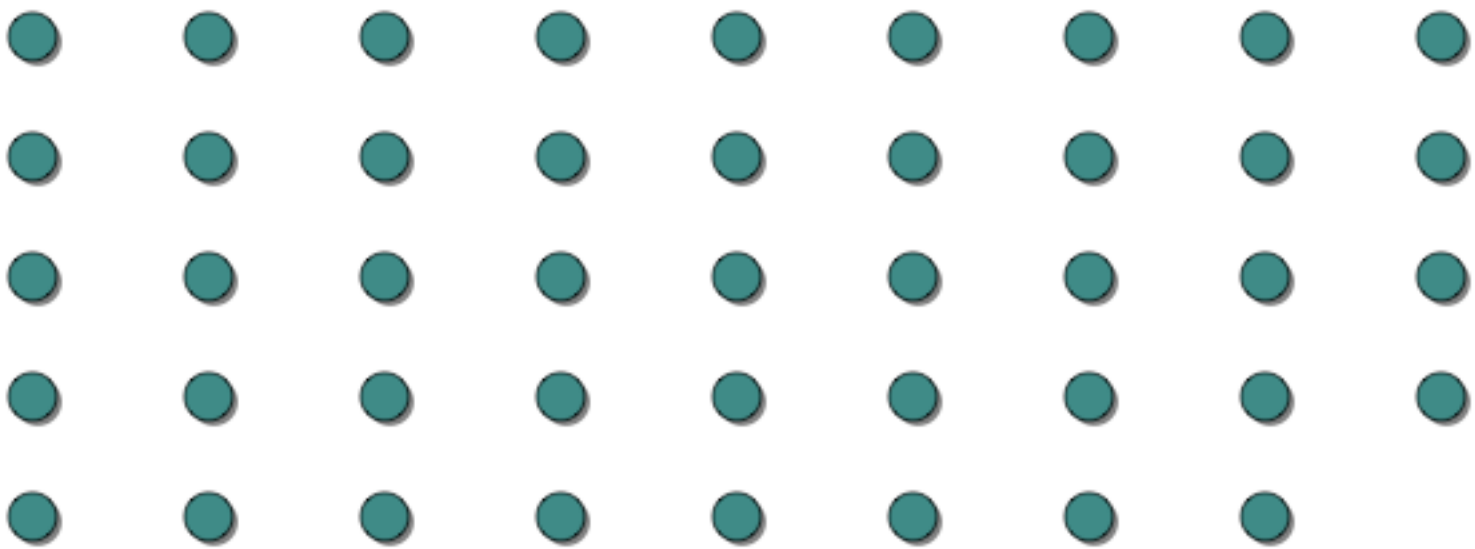
  Generate a good partitioning element

  Call this element x

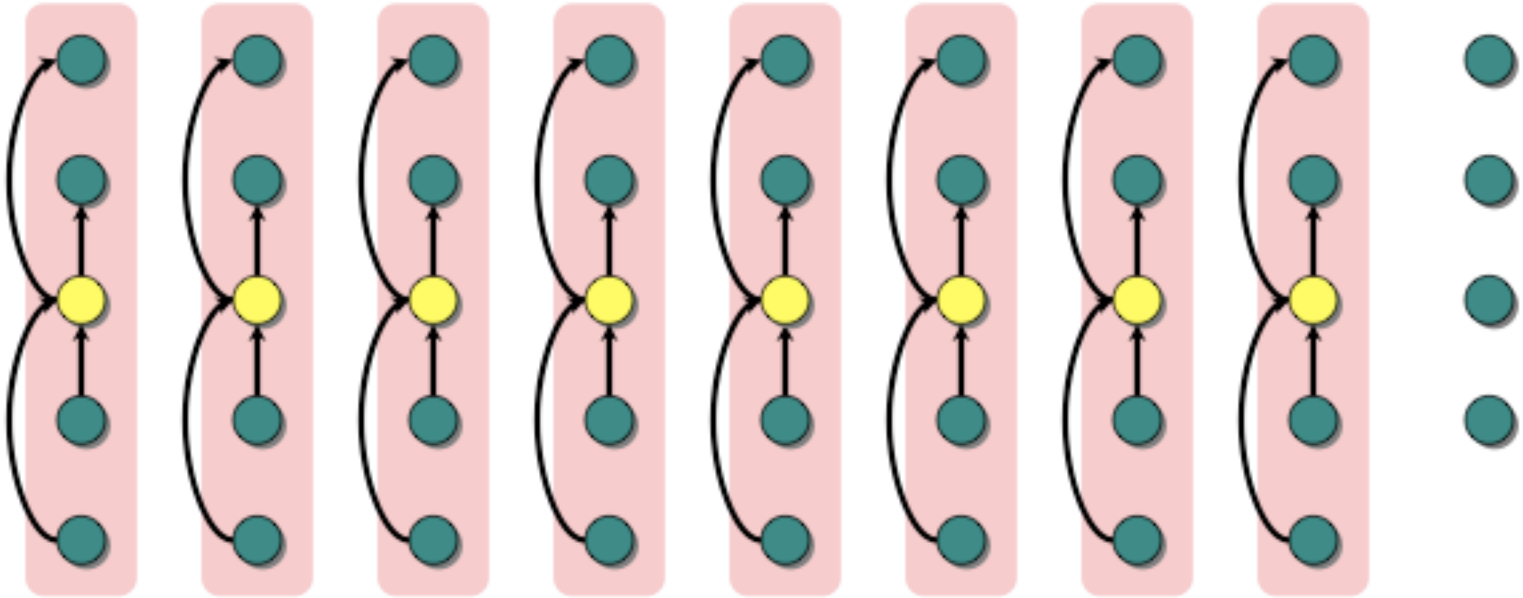
Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide n elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median x of the $\lfloor n/5 \rfloor$ medians
  4. Partition the n elements around x. Let $k = \text{rank}(x)$
  5. **if** (i == k) **then** return x
     if (i < k) **then** use Select() recursively to find ith smallest element in first partition
     else (i > k) use Select() recursively to find $(i\text{-}k)$th smallest element in last partition

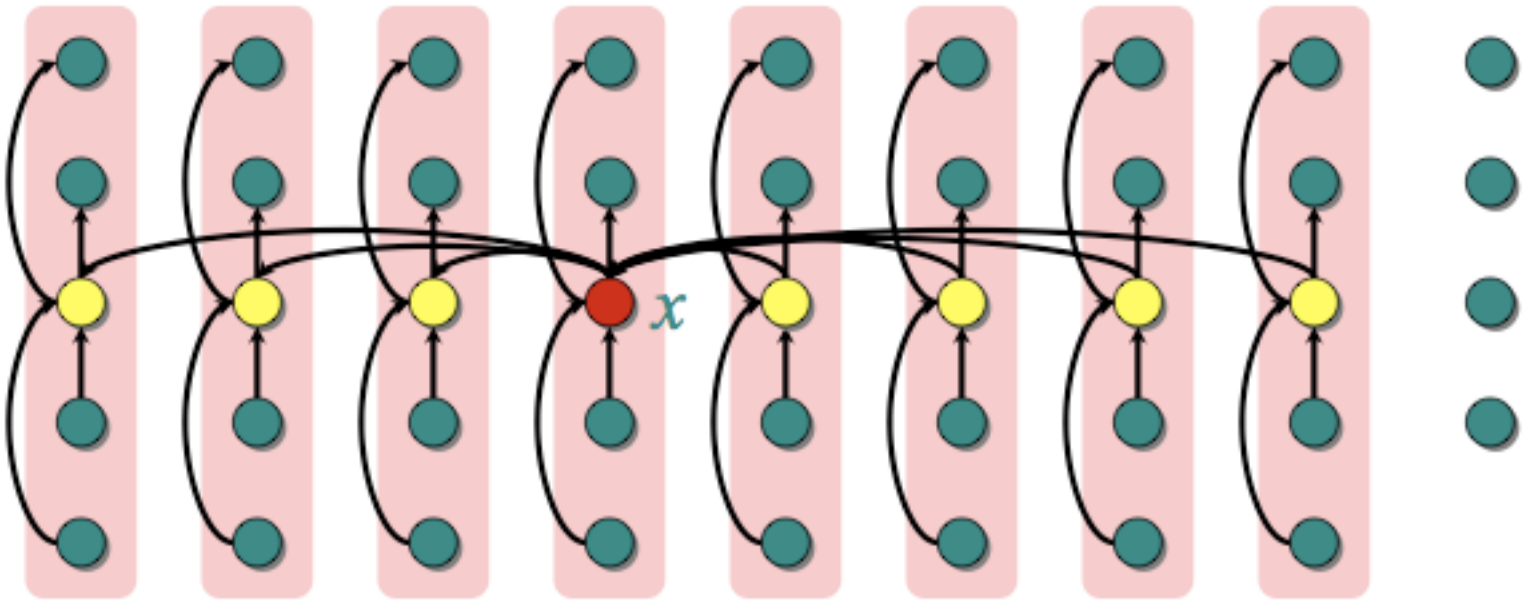
Worst-Case Linear-Time Selection



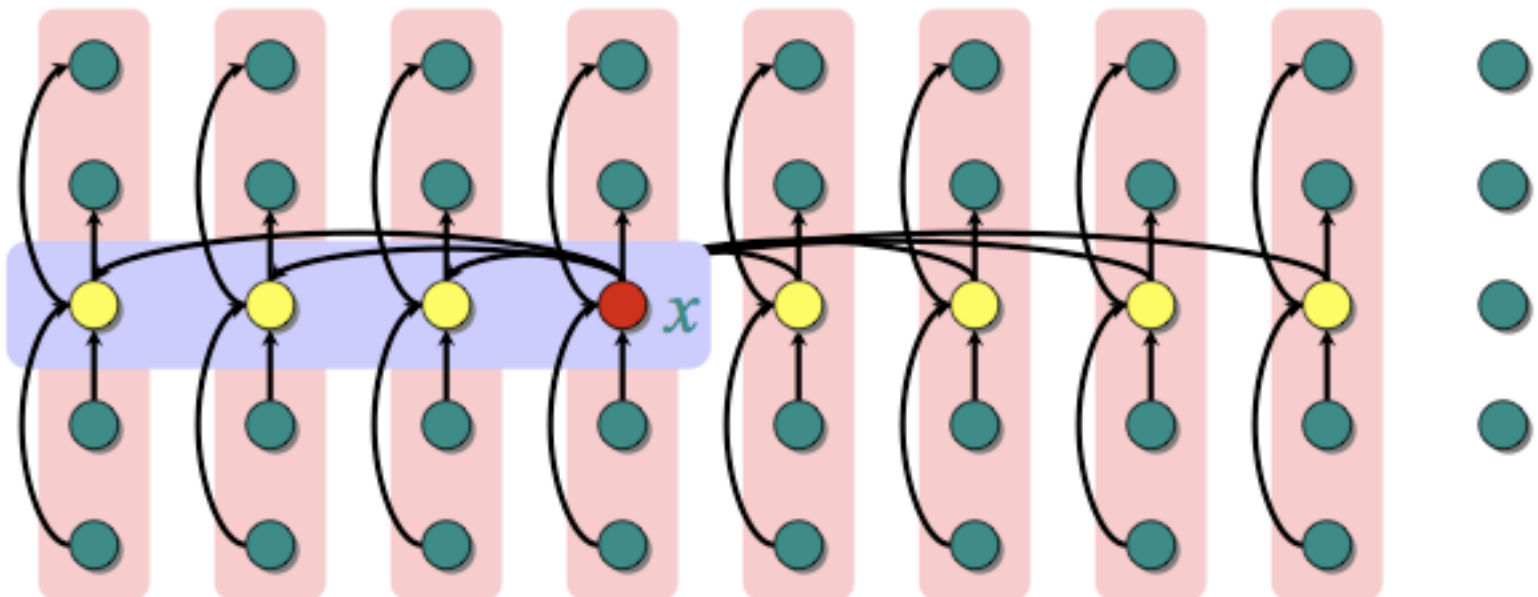
Worst-Case Linear-Time Selection



Worst-Case Linear-Time Selection



Worst-Case Linear-Time Selection



At least half the group medians are $\leq x$, 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 $\leq x$.
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$.

Worst-Case Linear-Time Selection

- (Sketch situation on the board)
- *How many of the 5-element medians are ≤ x?*

  At least 1/2 of the medians = $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are ≤ x?*

  At least $3 \lfloor n/10 \rfloor$ elements
- For large n, $3 \lfloor n/10 \rfloor \geq n/4$ *(How large?)*
- So at least $n/4$ elements $\leq x$
- Similarly: at least $n/4$ elements $\geq x$

Worst-Case Linear-Time Selection

$T(n)$	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

Worst-Case Linear-Time Selection

- Thus after partitioning around x, step 5 will call Select() on at most $3n/4$ elements
- The recurrence is therefore:

$$
\begin{aligned}
T(n) &\le T\left(\lfloor n/5 \rfloor\right) + T(3n/4) + \Theta(n) \\
&\le T(n/5) + T(3n/4) + \Theta(n) && \lfloor n/5 \rfloor \le n/5 \\
&\le cn/5 + 3cn/4 + \Theta(n) && \text{Substitute } T(n) = cn \\
&= 19cn/20 + \Theta(n) && \text{Combine fractions} \\
&= cn - \left(cn/20 - \Theta(n)\right) && \text{Express in desired form} \\
&\le cn \quad \text{if } c \text{ is big enough} && \text{What we set out to prove}
\end{aligned}
$$

Linear-Time Median Selection

- Given a "black box" O(n) median algorithm, what can we do?
- *i*th order statistic:

  Find median x

  Partition input around x

  if ($i \leq (n+1)/2$) recursively find *i*th element of first half

  else find (i - (n+1)/2)th element in second half

  $T(n) = T(n/2) + O(n) = O(n)$

Can you think of an application to sorting?

Linear-Time Median Selection

- Worst-case O(n lg n) quicksort
  Find median x and partition around it
  Recursively quicksort two halves
  $T(n) = 2T(n/2) + O(n) = O(n \lg n)$

Linear-Time Median Selection

- Worst-case O(n lg n) quicksort
  Find median *x* and partition around it
  Recursively quicksort two halves
  $T(n) = 2T(n/2) + O(n) = O(n \lg n)$

The End