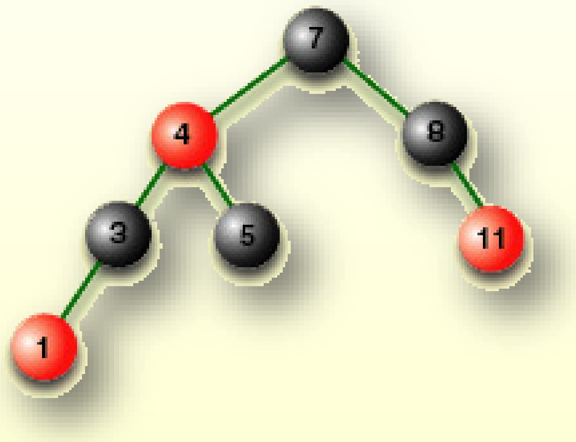
CS161:
Design and Analysis of Algorithms



Lecture 5
Leonidas Guibas

Outline

- Review of last lecture: QuickSort and its analysis

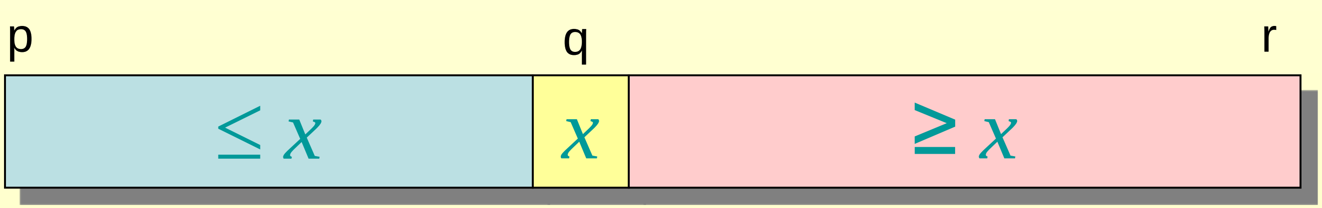
- Today: Medians and order statistics
  - Minimum, maximum, median, …
  - A randomized O(n) median algorithm
  - A worst-case O(n) median algorithm

Slides modified from
- http://www.cs.virginia.edu/~luebke/cs332/
- http://www.cs.unc.edu/~plaisted/

Review: Pseudocode for QuickSort

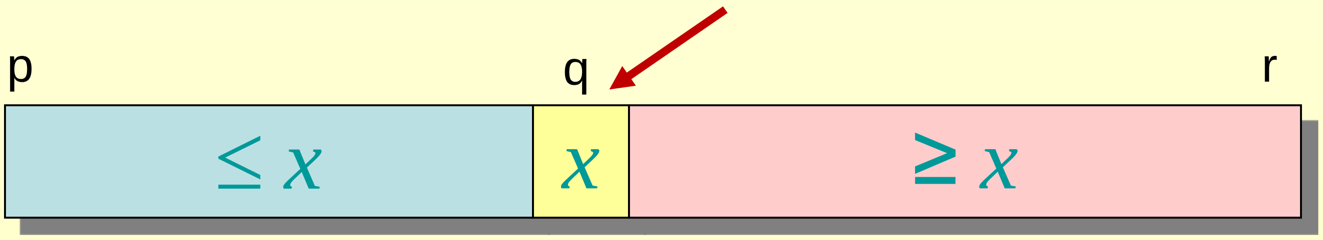
QUICKSORT(A, p, r)
    if $p < r$
        then $q \leftarrow$ PARTITION(A, p, r)
            QUICKSORT$(A, p, q–1)$
            QUICKSORT$(A, q+1, r)$

Initial call: QUICKSORT$(A, 1, n)$



Key: The Partition Subroutine

- All the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result: two subarrays
    - All values in first subarray $\leq$ all values in second
  - Returns the index of the "pivot" element separating the two subarrays



QuickSort Runtime

- Best-case runtime $T_{best}(n) \in \Theta(n \log n)$
- Worst-case runtime $T_{worst}(n) \in \Theta(n^2)$
- Average runtime $T_{avg}(n) \in \Theta(n \log n)$

- Better even, the expected runtime of **randomized QuickSort** is $\Theta(n \log n)$
- Great in practice

Randomized Algorithms

Randomized QuickSort

- Randomly choose an element as pivot
  - Every time need to do a partition, throw a die to decide which element to use as the pivot
  - Each of the n elements has 1/n probability to be selected

```
Rand-Partition(A, p, r)
    d = random();    // a random number between 0 and 1
    index = p + floor((r-p+1) * d);  // p<=index<=r
    swap(A[p], A[index]);
    Partition(A, p, r);  // now do partition using A[p] as pivot
```

Randomized Analysis

- Assume each of the pivot is equally likely and hence probability is 1/N.

$$T(N) = \frac{1}{N}\sum_{i=0}^{N-1}\left(T(i) + T(N-i-1) + cN\right)$$

$$NT(N) = 2\sum_{i=0}^{N-1}\left(T(i)\right) + cN^2 \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

$$(N-1)T(N-1) = 2\sum_{i=0}^{N-2}T(i) + c(N-1)^2 \ldots(2)$$

- Subtract (2) from (1)

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$$

$$NT(N) = (N+1)T(N-1) + 2cN$$

- Divide both sides by N(N+1)

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1}$$

- Now we can iterate

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1}$$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

- adding all equations

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{N+1}\frac{1}{i}$$

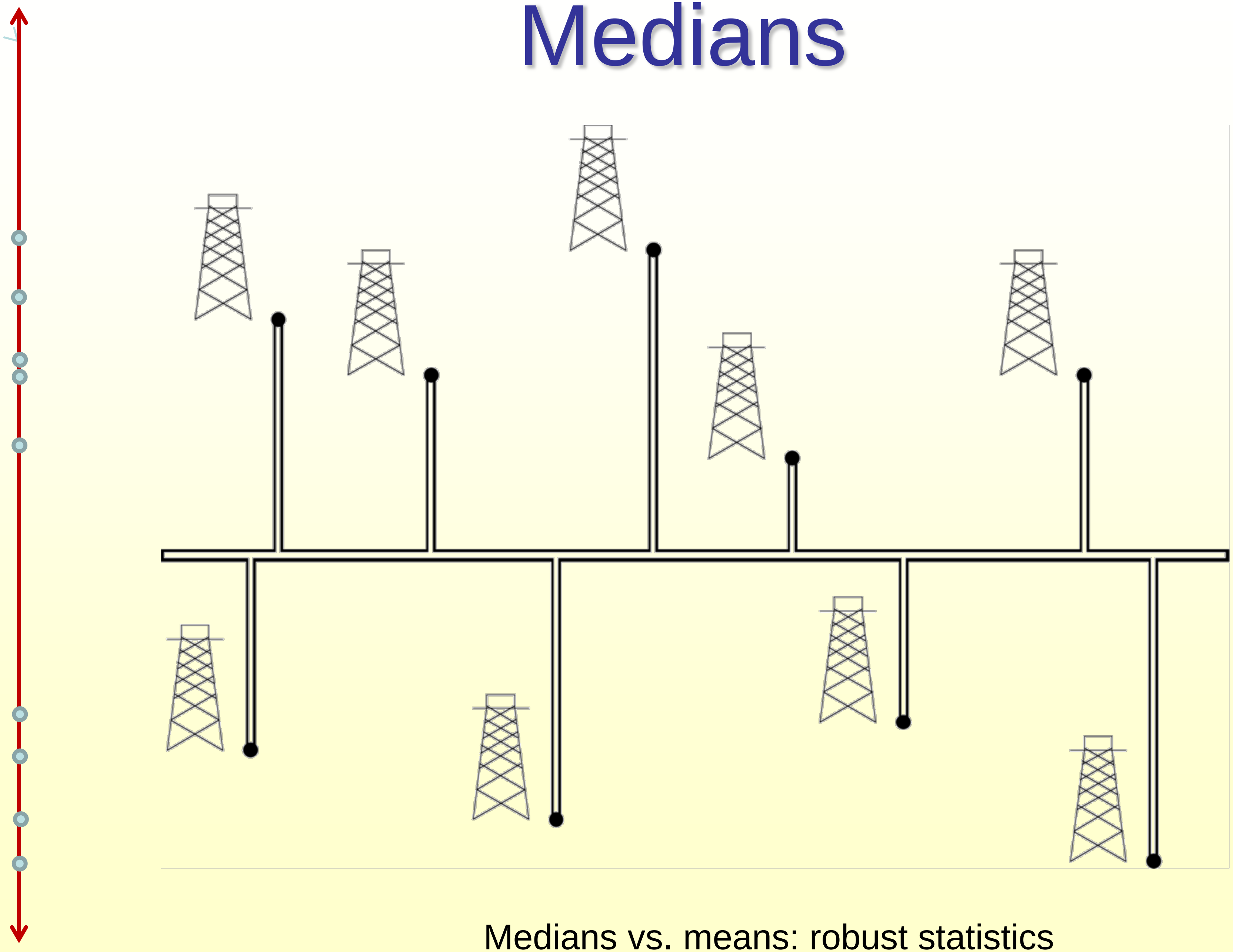
$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c(\log_e(N+1) + \gamma - 3/2)$$

$$T(N) = O(N\log N)$$

Today: Order Statistics

- i^{th} order statistic: i^{th} smallest element of a set of n elements.
- Minimum: 1^{st} order statistic.
- Maximum: n^{th} order statistic.
- Median: $(n/2)^{th}$ order statistic -- "half-way point" of the set.
  - Unique, when n is odd – occurs at $i = (n+1)/2$.
  - Two medians when n is even.
    - Lower median, at $i = n/2$.
    - Upper median, at $i = n/2+1$.
    - For consistency, "median" will refer to the lower median.

Medians



Medians vs. means: robust statistics

Selection Problem

- The selection problem:
  - Input: A set A of n **distinct** numbers and an index i, with $1 \leq i \leq n$.
  - Output: the element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

Minimum (Maximum)

<u>*Minimum* (*A*)</u>

1. *min* ← *A*[1]
2. **for** *i* ← *2* **to** *length*[*A*]
3. **do if** *min* > *A*[*i*]
4. **then** *min* ← *A*[*i*]
5. **return** *min*

Maximum can be determined similarly.

- $T(n) = \Theta(n)$.
- No. of comparisons: $n - 1$.
- Can we do better? <u>Why not?</u>
- Minimum(A) has *worst-case optimal* # of comparisons.

Problem

```
Minimum (A)
1. min ← A[1]
2. for i ← 2 to length[A]
3.     do if min > A[i]
4.         then min ← A[i]
5. return min
```

- Average for random input: How many times do we expect line 4 to be executed?
  - X = RV for # of executions of line 4.
  - X_i = Indicator RV for the event that line 4 is executed on the i^{th} iteration.
  - $X = \Sigma_{i=2..n} X_i$
  - $E[X_i] = 1/i$. Why?
  - Hence, $E(X) = \sum_{i=2}^{n} \frac{1}{i} = H_n - 1 = \Theta(\ln n) = \Theta(\log n)$

Simultaneous Min and Max

- Some applications need to determine both the maximum and minimum of a set of elements.
  - Example: Graphics program trying to fit a set of points onto a rectangular display.
- Independent determination of maximum and minimum requires $2n - 2$ comparisons.
- Can we reduce this number?
  - Yes.

Simultaneous Min and Max

- Maintain *minimum* and *maximum* elements seen so far.
- Process elements in pairs.
  - Compare the smaller to the current minimum and the larger to the current maximum.
  - Update current minimum and maximum based on the outcomes.
- No. of comparisons per pair = 3. How?
- No. of pairs $\leq \lfloor n/2 \rfloor$.
  - For odd n: initialize min and max to $A[1]$. Pair the remaining elements. So, no. of pairs = $\lfloor n/2 \rfloor$.
  - For even n: initialize min to the smaller of the first pair and max to the larger. So, remaining no. of pairs = $(n - 2)/2 < \lfloor n/2 \rfloor$.

Simultaneous Min and Max

- Total no. of comparisons, $C \leq 3\lfloor n/2 \rfloor$.
  - For odd n: $C = 3\lfloor n/2 \rfloor$.
  - For even n: $C = 3(n - 2)/2 + 1$ (For the initial comparison).

$$= 3n/2 - 2 < 3\lfloor n/2 \rfloor.$$

Order Statistics

- The i^{th} order statistic in a set of n elements is the i^{th} smallest element
- The minimum is thus the 1st order statistic
- The maximum is the n^{th} order statistic
- The median is the $n/2$ order statistic
  - If *n* is even, there are 2 medians
- *How can we calculate general order statistics?*
- *What is the running time?*

The General Selection Problem

- Select the i^{th} smallest of n elements
- ***Naive algorithm***: Sort.
  - Worst-case running time $\Theta(n \log n)$

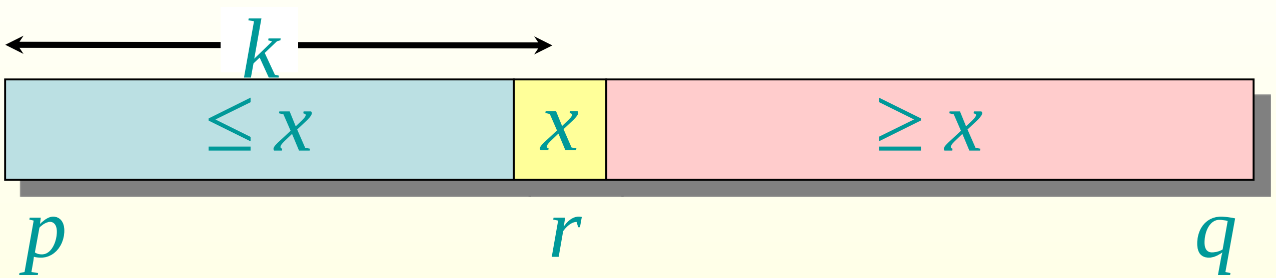
    using MergeSort (*not* InsertionSort or QuickSort).

General Selection Problem

- Seems more difficult than Minimum or Maximum.
  - Yet, has solutions with same asymptotic complexity as Minimum and Maximum.
- We will study two algorithms for the general problem.
  - One with *expected* linear-time complexity.
  - A second, whose *worst-case complexity* is linear.

Recall: QuickSort

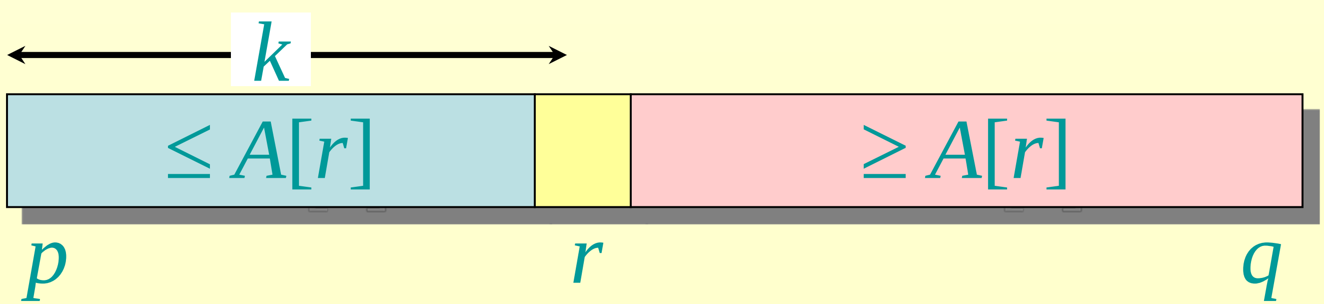
- The function *Partition* gives us the rank of the pivot



- If we are lucky, $k = i$. *done!*
- If not, at least get a smaller subarray to work with
  - $k > i$: i^{th} smallest is on the left subarray
  - $k < i$: i^{th} smallest is on the right subarray
- Divide and conquer
  - If we are lucky, *k* close to *n*/2, or desired # is in smaller subarray
  - If unlucky, desired # is in larger subarray (possible size *n*-1)

Randomized D&C Selection

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



Randomized Partition

- Randomly choose an element as pivot
  - Every time need to do a partition, throw a die to decide which element to use as the pivot
  - Each element has 1/n probability to be selected

```
Rand-Partition(A, p, q){
    d = random();   // draw a random number between 0 and 1
    index = p + floor((q-p+1) * d);  // p<=index<=q
    swap(A[p], A[index]);
    Partition(A, p, q);  // now use A[p] as pivot
}
```

Example

Select the $i = 6$-th smallest:

7	10	5	8	11	3	2	13	$i = 6$

pivot

Partition: $k = 4$

3	2	5	7	11	8	10	13

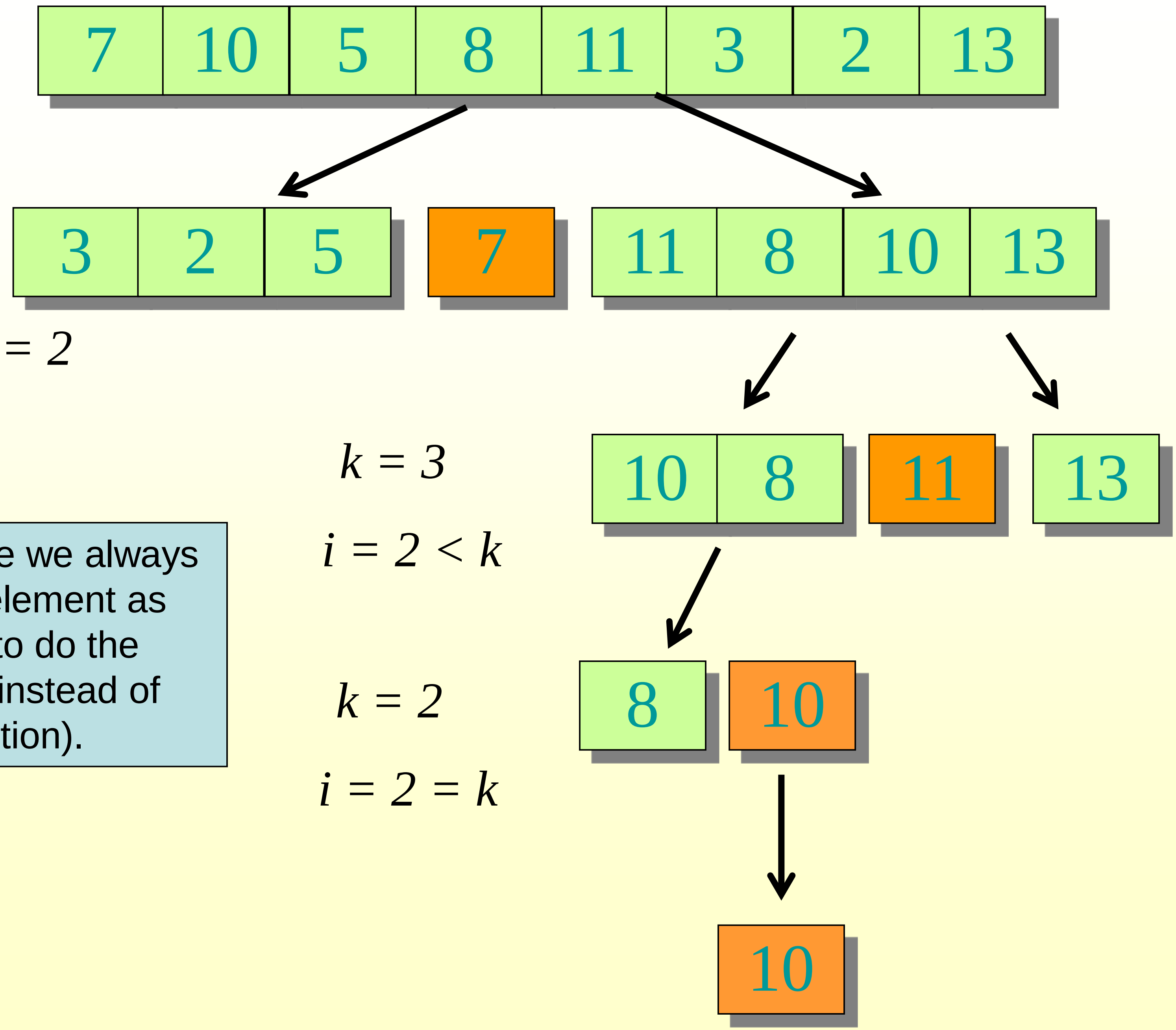
Select the $6 - 4 = 2$-nd smallest recursively.

Complete example: select the 6th smallest element.

$i = 6$

$k = 4$

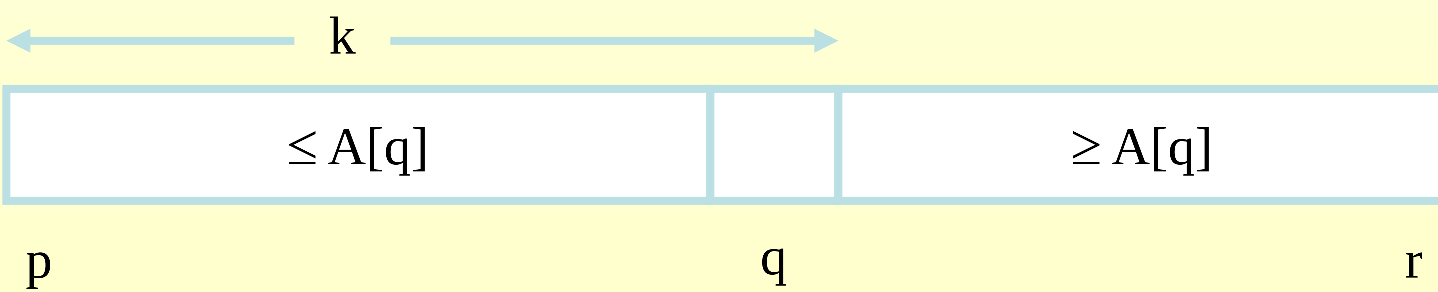
$i = 6 - 4 = 2$



Note: here we always use first element as the pivot to do the partition (instead of rand-partition).

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);
```



Intuition for Analysis

(Our analyses assume that all elements are distinct.) Like QuickSort – but now only ONE recursive call.

Lucky:

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

CASE 3

Unlucky:

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

arithmetic series

Worse than sorting!

Running Time of Randomized Selection

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

- For upper bound, assume i^{th} element always falls in larger side of partition
- The expected running time is an average of all cases

Expectation

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

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

- 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:

$$
\begin{aligned}
T(n) &\le c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n) && \textit{\textbf{The recurrence so far}} \\
&= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n) && \textit{\textbf{Multiply it out}} \\
&= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n) && \textit{\textbf{Subtract c/2}} \\
&= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right) && \textit{\textbf{Rearrange the arithmetic}} \\
&\le cn \quad \text{(if c is big enough)} && \textit{\textbf{What we set out to prove}}
\end{aligned}
$$

Different Probabilistic Analysis

- Assume each of n! permutations is equally likely
- Modify earlier indicator variable analysis of quicksort (method 2) to handle this k-selection problem
- What is probability i-th smallest item is compared to j-th smallest item (assume $i<j$)?
  - If k is contained in (i..j)?
  - If $k \leq i$?
  - If $k \geq j$?

Now the Probabilities of Comparison Get Smaller

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E\left[X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\left\{z_i \text{ is compared to } z_j\right\}.$$

- Before $\Pr\left\{z_i \text{ is compared to } z_j\right\} = \dfrac{2}{j-i+1}$

Selection, say i<j<k

- So now $\Pr\left\{z_i \text{ is compared to } z_j\right\} = \dfrac{2}{k-i+1}$

Case: (i..j) Does Not Contain k

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

- Case k ≥ j:
  - $\Sigma_{(i=1 \text{ to } k-1)} \Sigma_{j = i+1 \text{ to } k}\ 2/(k-i+1) = \Sigma_{i=1 \text{ to } k-1}\ (k-i)\ 2/(k-i+1)$
    $= \Sigma_{i=1 \text{ to } k-1}\ 2i/(i+1)$ [replace k-i with i]
    $= 2\ \Sigma_{i=1 \text{ to } k-1}\ i/(i+1)$
    $\leq 2(k-1)$
- Case k ≤ i:
  - $\Sigma_{(j=k+1 \text{ to } n)} \Sigma_{i = k \text{ to } j-1}\ 2/(j-k+1) = \Sigma_{j=k+1 \text{ to } n}\ (j-k)\ 2/(j-k+1)$
    $= \Sigma_{j = 1 \text{ to } n-k}\ 2j/(j+1)$
    [replace j-k with j and change bounds]
    $= 2\ \Sigma_{j=1 \text{ to } n-k}\ j/(j+1)$
    $\geq 2(n-k)$
- Total for both cases is ≤ 2n-2

Case: (i..j) contains k

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

- At most 1 interval of size 3 contains k
  - i=k-1, j=k+1
- At most 2 intervals of size 4 contain k
  - i=k-1, j=k+2 and i=k-2, j= k+1
- In general, at most q-2 intervals of size q contain k
- Thus we get $\Sigma_{(q=3 \text{ to } n)}$ (q-2)2/q ≤ $\Sigma_{(q=3 \text{ to } n)}$ 2 = 2(n-2)
- Summing together all cases we see the expected number of comparisons is less than 4n

Summary of Randomized 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.

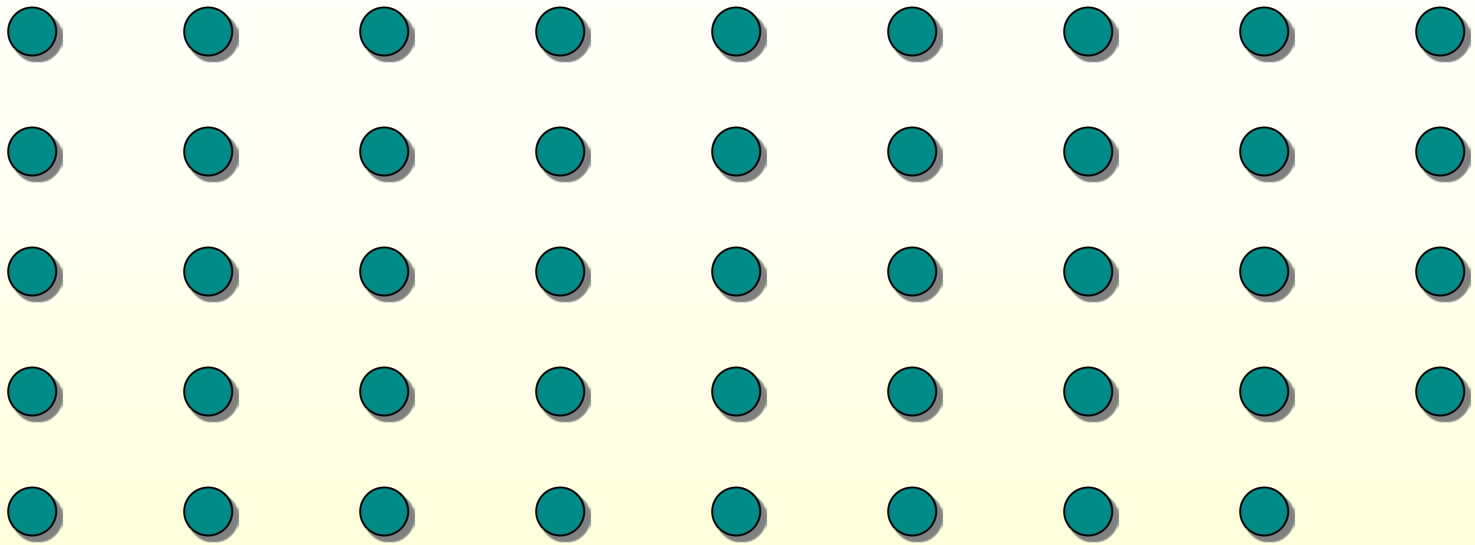
Worst-Case Linear-Time Selection

SELECT(i, n)

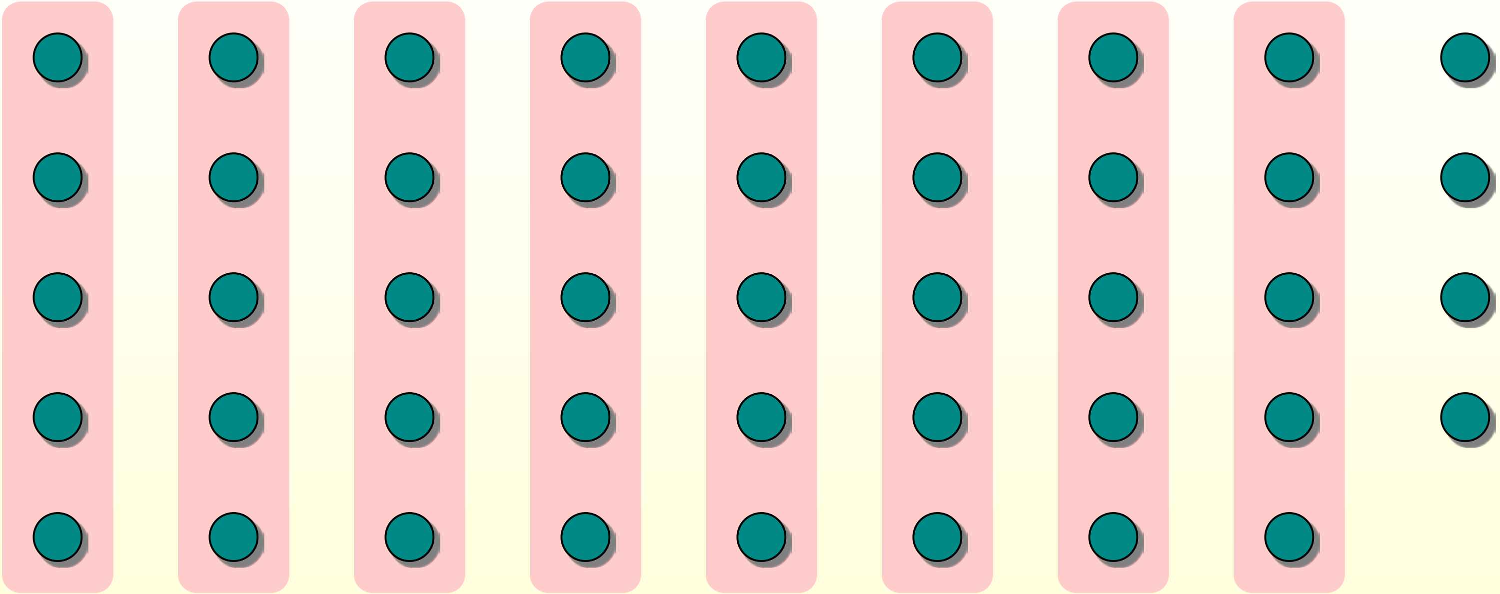
1. Divide the n elements into groups of 5. Find the median of each 5-element group by brute force.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let $k = \text{rank}(x)$.
4. **if** $i = k$ **then return** x
   elseif $i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

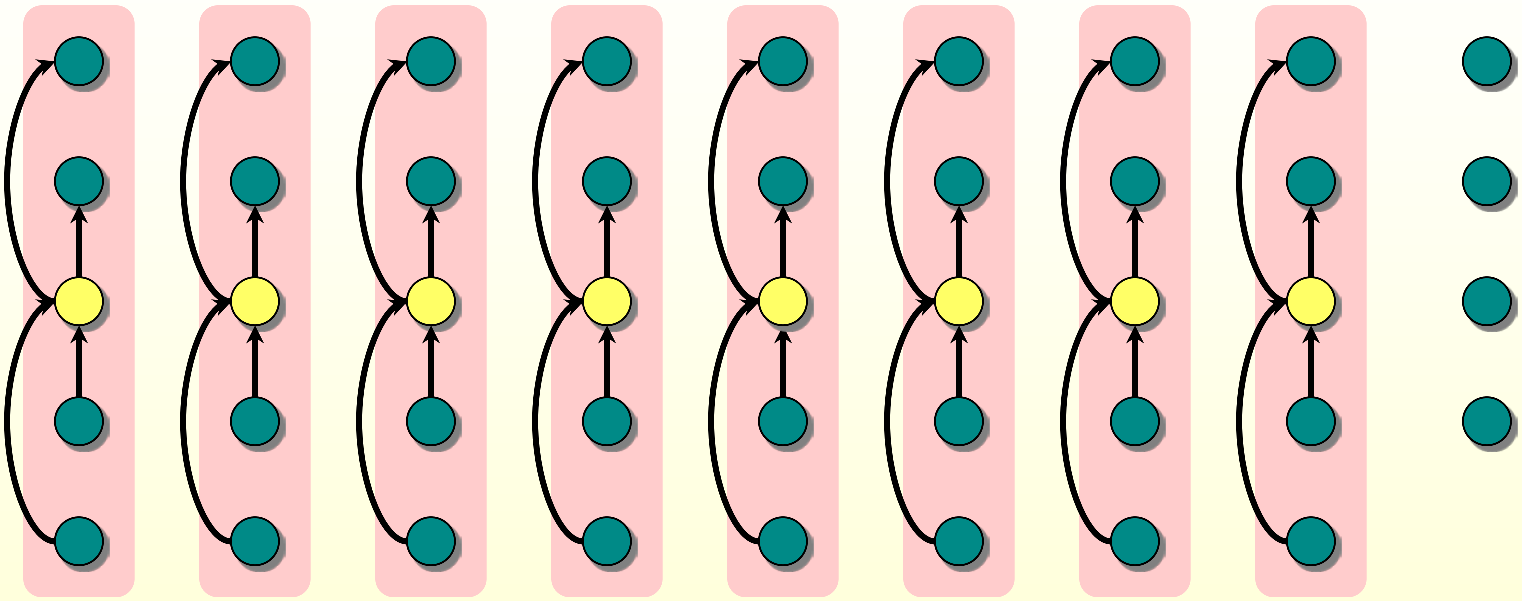


Choosing the Pivot



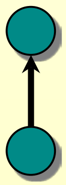
1. Divide the n elements into groups of 5.

Choosing the Pivot



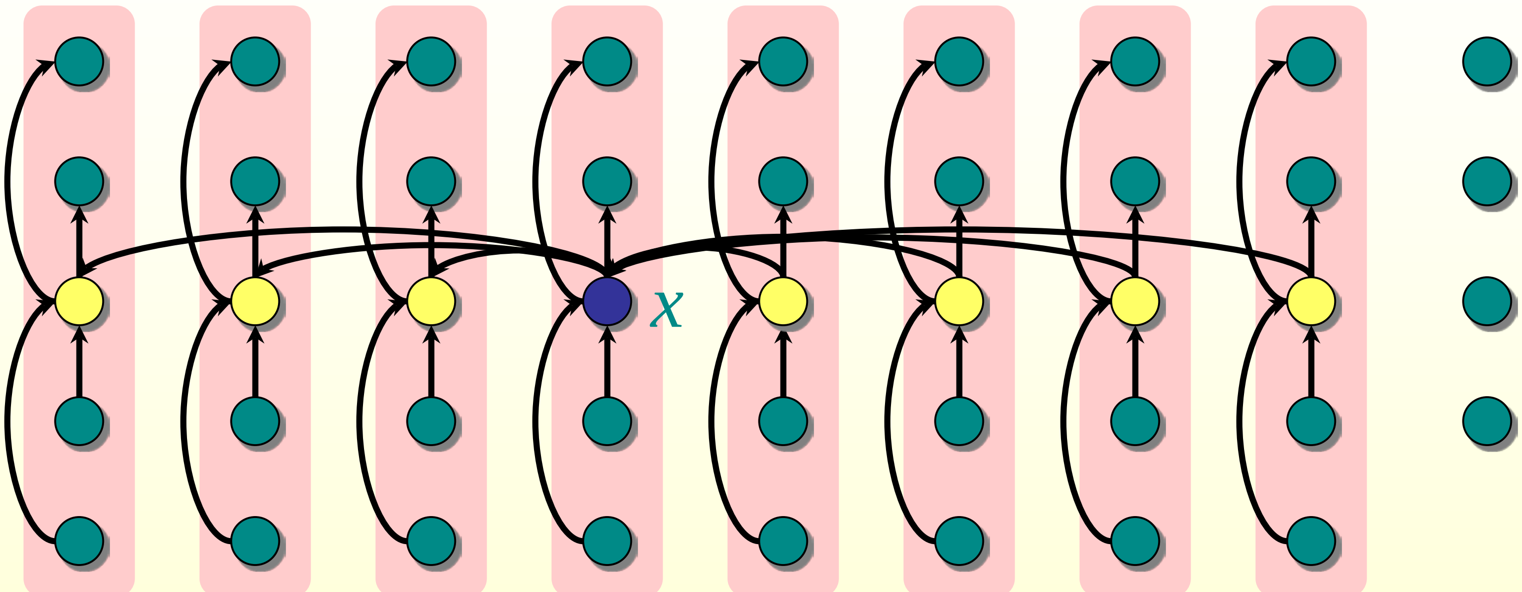
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

lesser



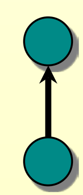
greater

Choosing the Pivot



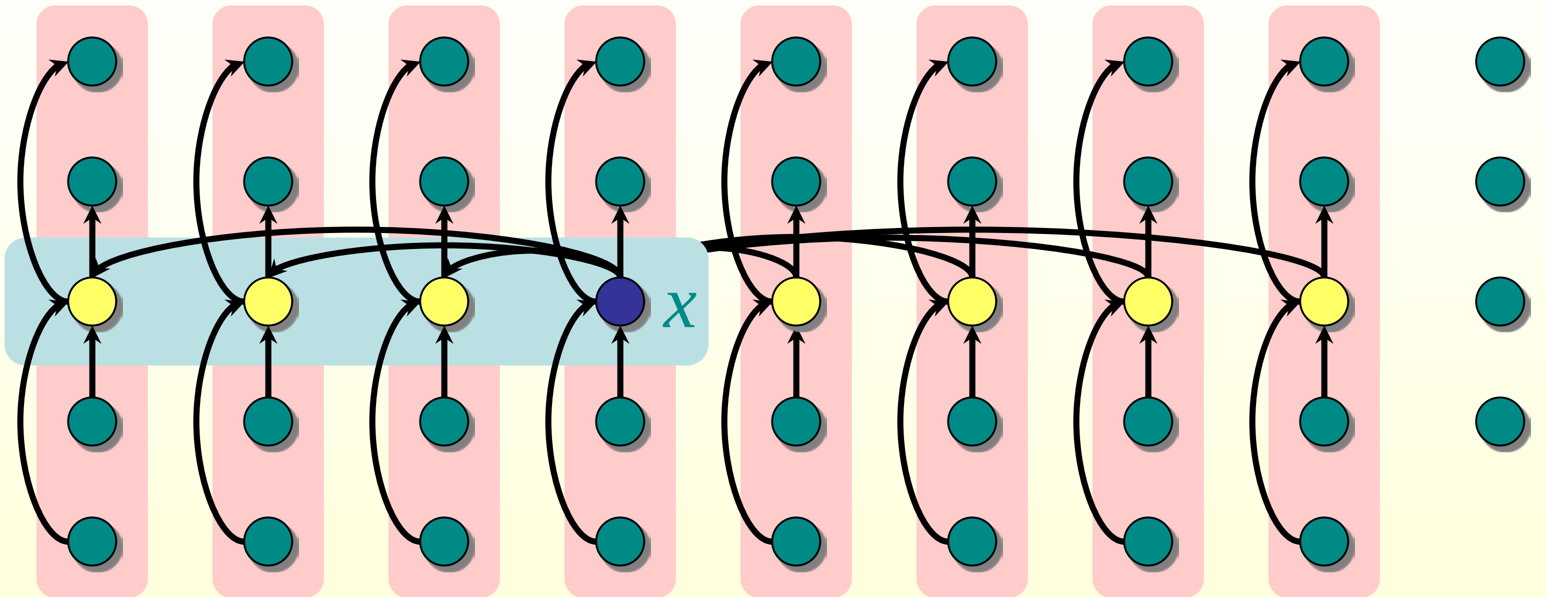
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

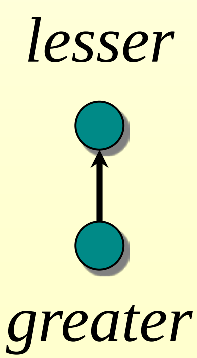
lesser



greater

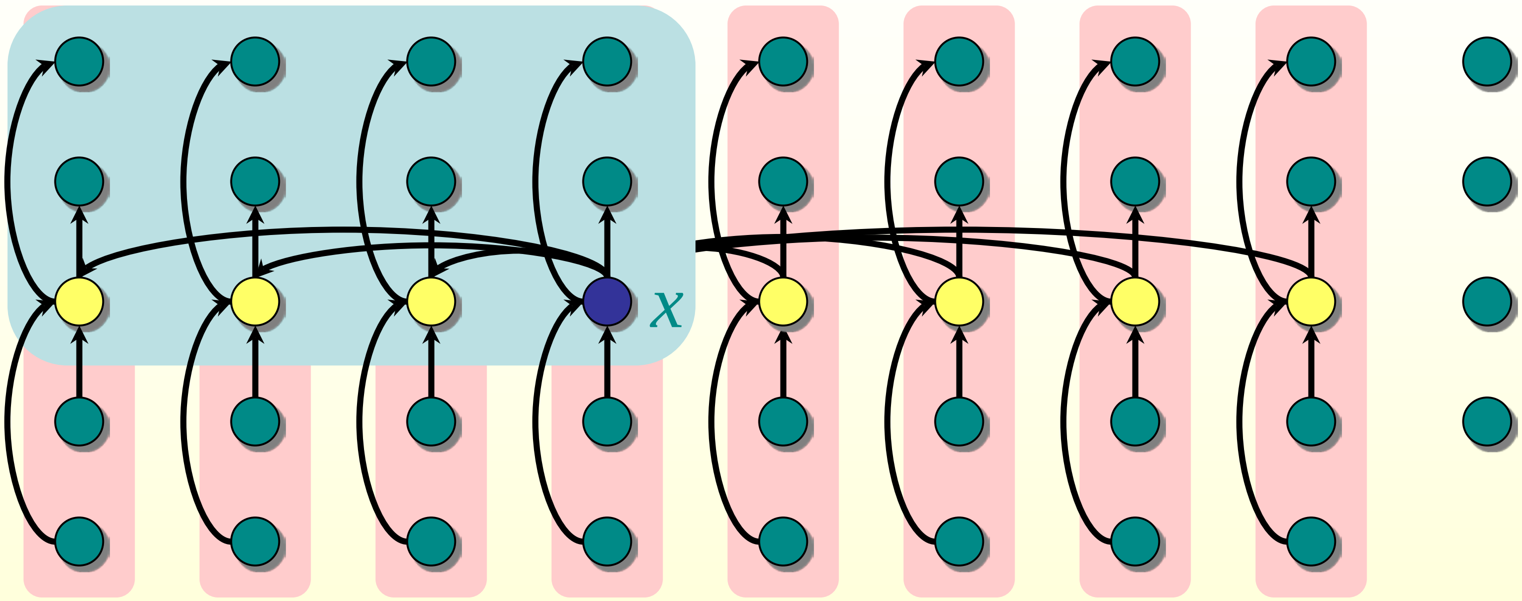
Analysis





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

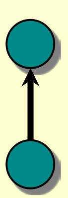
Analysis



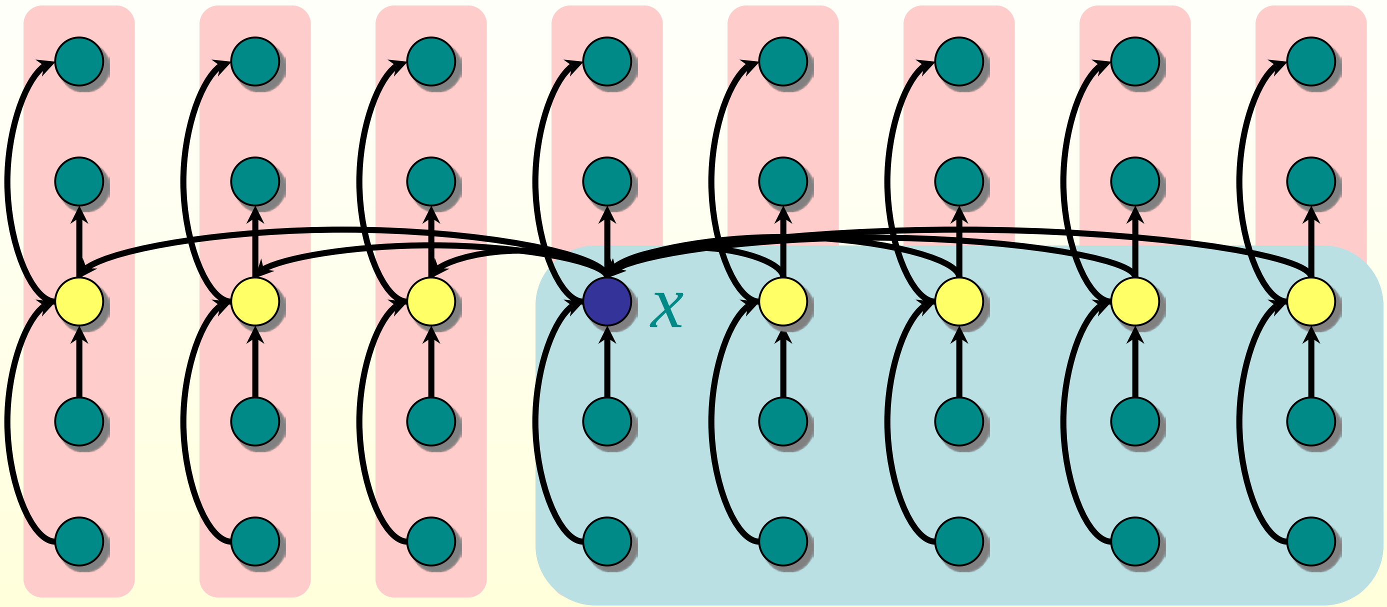
At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

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

(Assume all elements are distinct.)

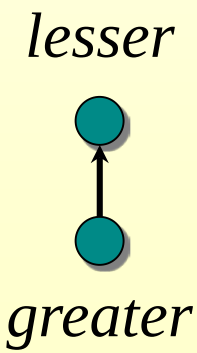


Analysis



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.

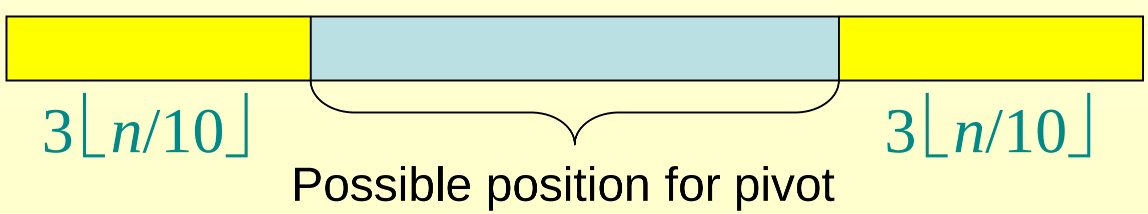
- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\le x$.
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\ge x$.



Analysis

Need “at most” for worst-case runtime

- At least $3\lfloor n/10 \rfloor$ elements are $\leq x$
  $\Rightarrow$ at most $n\text{-}3\lfloor n/10 \rfloor$ elements are $\geq x$
- At least $3\lfloor n/10 \rfloor$ elements are $\geq x$
  $\Rightarrow$ at most $n\text{-}3\lfloor n/10 \rfloor$ elements are $\leq x$
- The recursive call to SELECT in Step 4 is executed recursively on at most $n\text{-}3\lfloor n/10 \rfloor$ elements.



Analysis

- Use fact that $\lfloor a/b \rfloor > a/b-1$
- $n-3\lfloor n/10 \rfloor < n-3(n/10-1) \leq 7n/10 + 3$

  $[\ \leq 3n/4 \text{ if } n \geq 60\ ]$
- The recursive call to SELECT in Step 4 is executed recursively on at most $7n/10+3$ elements.

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 $k = \mathrm{rank}(x)$.
$T(7n/10 +3)$	4. **if** $i = k$ **then return** x **elseif** $i < k$ **then** recursively SELECT the i-th smallest element in the lower part **else** recursively SELECT the $(i–k)$-th smallest element in the upper part

Solving the Recurrence

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n + 3\right) + n$$

Assumption: $T(k) \le ck$ *for all* $k < n$

$$\begin{aligned} T(n) &\le c(n/5) + c(7n/10 + 3) + n \\ &\le cn/5 + 3cn/4 + n \quad \text{if } n \ge 60 \\ &= 19cn/20 + n \\ &\le cn - (cn/20 - n) \\ &\le cn \quad \text{if } c \ge 20 \text{ and } n \ge 60 \end{aligned}$$

Worst-Case Linear-Time Selection

- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller as we go down the tree
    - Geometric progression!
  - Thus the O(n) work at the root dominates

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
  - $T(n) = 2T(n/2) + O(n) = O(n \lg n)$

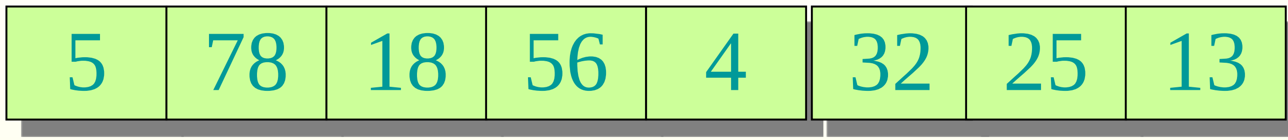
Conclusion

- In practice, the $\Theta(n)$ median algorithm runs very slowly, because the constant in front of n is large.
- The randomized algorithm is far more practical.

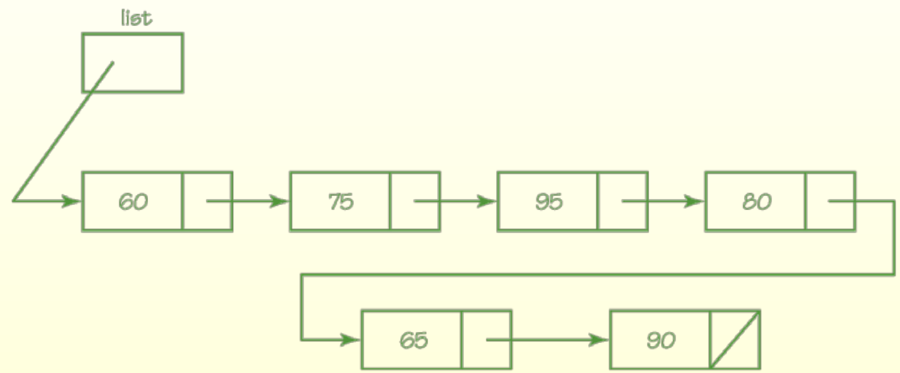
Exercise: *Try to divide into groups of 3 or 7.*

Basic Data Structures

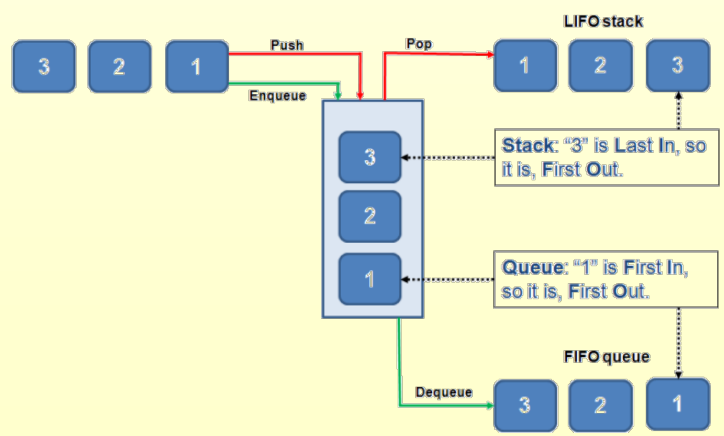
- Arrays



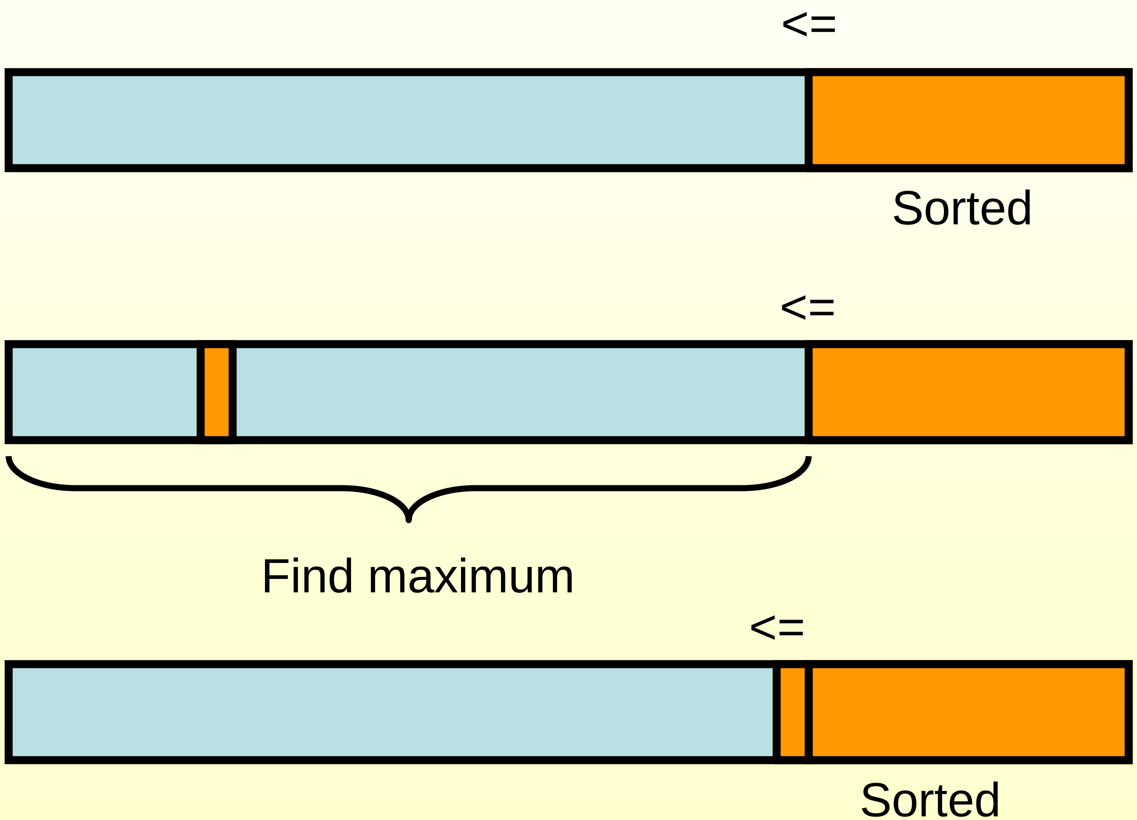
- Linked lists



- Stacks
- Queues



SelectionSort



SelectionSort

```
SelectionSort(A[1..n])
  for (i = n; i > 0; i--)
      index = max_element(A[1..i])
      swap(A[i], A[index]);
  end
```

What's the time complexity?

If max_element takes Θ(n),
selection sort takes $\sum_{i=1}^{n} i = \Theta(n^2)$

HeapSort

- Another $\Theta(n \log n)$ sorting algorithm
- In practice QuickSort wins
- However, the heap data structure and its variants are very useful for many algorithms