Median and Order Statistics

Medians

The median of a set of numbers is the number such that half of the numbers are larger and half smaller

A = [50, 12, 1, 97, 30]

How might we calculate the median of a set?

Sort the numbers, then pick the n/2 element

A = [1, 12, 30, 50, 97]

runtime?

Medians

The median of a set of numbers is the number such that half of the numbers are larger and half smaller

A = [50, 12, 1, 97, $\boxed{30}$]

How might we calculate the median of a set?

Sort the numbers, then pick the n/2 element

A = [1, 12, $\boxed{30,}$ 50, 97]

$\Theta(n \log n)$

Selection

More general problem:
find the *k*-th smallest element in an array

- i.e., element where exactly k-1 things are smaller than it
- aka the "selection" problem
- can use this to find the median if we want

Can we solve this in a similar way?

- Yes, sort the data and take the kth element
- $\Theta(n \log n)$

Can we do better?

Are we doing more work than we need to?

To get the k-th element (or the median) by sorting, we're finding *all* the k-th elements at once.

We just want one!

Often when you find yourself doing more work than you need to, there is a faster way (though not always).

selection problem

Our tools

- divide and conquer
- sorting algorithms
- other functions
  - partition
  - binary search

Partition

Partition takes $\Theta(n)$ time and performs a similar operation.

given an element A[q], Partition can be seen as dividing the array into three sets:

- < A[q]
- = A[q]
- > A[q]

Ideas?

An example

We're looking for the 5^{th} smallest

5 2 34 9 17 2 1 34 18 5 3 2 1 6 5

If we called partition, what would be in three sets?

< 5:

= 5:

> 5:

An example

We're looking for the 5th smallest

5 2 34 9 17 2 1 34 18 5 3 2 1 6 5

< 5: 2 2 1 3 2 1

= 5: 5 5 5

> 5: 34 9 17 34 18 6

Does this help us?

An example

We're looking for the 5^{th} smallest

5 2 34 9 17 2 1 34 18 5 3 2 1 6 5

< 5: $\boxed{2\ 2\ 1\ 3\ 2\ 1}$

We know the 5^{th} smallest has to be in this set

= 5: 5 5 5

> 5: 34 9 17 34 18 6

Selection: divide and conquer

Call partition

- decide which of the three sets contains the answer we're looking for
- recurse

Like binary search on unsorted data

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
     Return A[q]
  else if k < relq
     Return Selection(A, k, p, q-1)
  else // k > relq
     Return Selection(A, k-relq, q+1, r)
```

?

Selection: divide and conquer

Call partition

- decide which of the three sets contains the answer we're looking for
- recurse

Like binary search on unsorted data

```
Selection(A, k, p, r)

  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Return Selection(A, k, p, q-1)
  else // k > relq
    Return Selection(A, k-relq, q+1, r)
```

As we recurse, we may update the k that we're looking for because we update the lower end

Selection: divide and conquer

Call partition

- decide which of the three sets contains the answer we're looking for
- recurse

Like binary search on unsorted data

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Return Selection(A, k, p, q-1)
  else // k > relq
    Return Selection(A, k-relq, q+1, r)
```

Partition returns the absolute index, we want an index relative to the current p (window start)

Selection(A, 3, 1, 8)

1	2	3	4	5	6	7	8
5	7	1	4	8	3	2	6

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 3, 1, 8)

1	2	3	4	5	6	7	8
5	1	4	3	2	6	8	7

$$relq = 6 - 1 + 1 = 6$$

```
Selection(A, k, p, r)

  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 3, 1, 8)

1	2	3	4	5	6	7	8
5	1	4	3	2	6	8	7

relq = 6 – 1 + 1 = 6

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 3, 1, 5)

1	2	3	4	5	6	7	8
5	1	4	3	2	6	8	7

At each call, discard
part of the array

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 3, 1, 5)

1	2	3	4	5	6	7	8
1	2	4	3	5	6	8	7

$$relq = 2 – 1 + 1 = 2$$

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 5)

1	2	3	4	5	6	7	8
1	2	4	3	5	6	8	7

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 5)

1	2	3	4	5	6	7	8
1	2	4	3	5	6	8	7

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 5)

1	2	3	4	5	6	7	8
1	2	4	3	5	6	8	7

↑

relq = 5 – 3 + 1 = 3

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 4)

1	2	3	4	5	6	7	8
1	2	4	3	5	6	8	7

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 4)

1	2	3	4	5	6	7	8
1	2	4	3	5	6	8	7

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 4)

1	2	3	4	5	6	7	8
1	2	3	4	5	6	8	7

```
Selection(A, k, p, r)
  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Selection(A, 1, 3, 4)

1	2	3	4	5	6	7	8
1	2	3	4	5	6	8	7

relq = 3 – 3 + 1 = 1

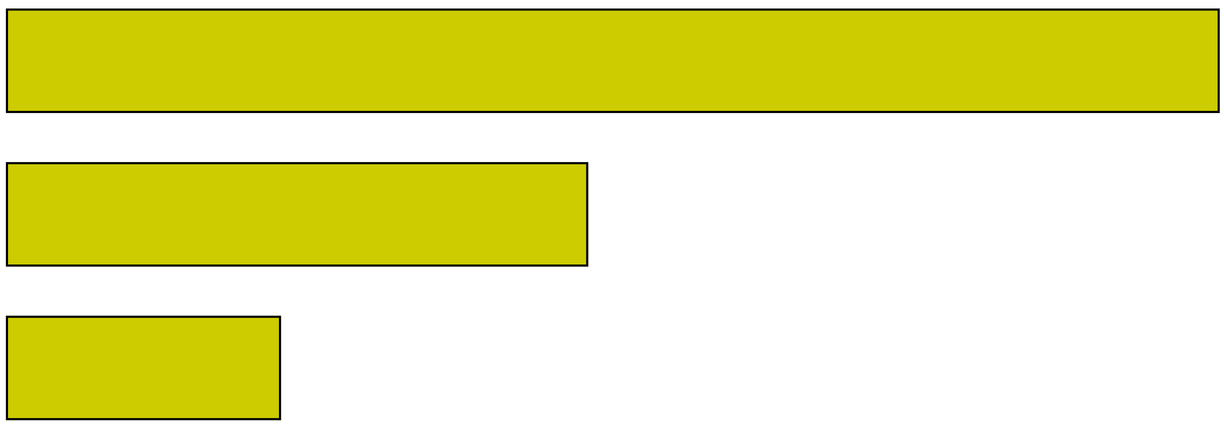
```
Selection(A, k, p, r)

  q <- Partition(A,p,r)
  relq = q-p+1
  if k = relq
    Return A[q]
  else if k < relq
    Selection(A, k, p, q-1)
  else // k > relq
    Selection(A, k-relq, q+1, r)
```

Running time of Selection?

Best case?

Each call to Partition throws away half the data



Recurrence?

$$T(n) = T(n/2) + \Theta(n)$$

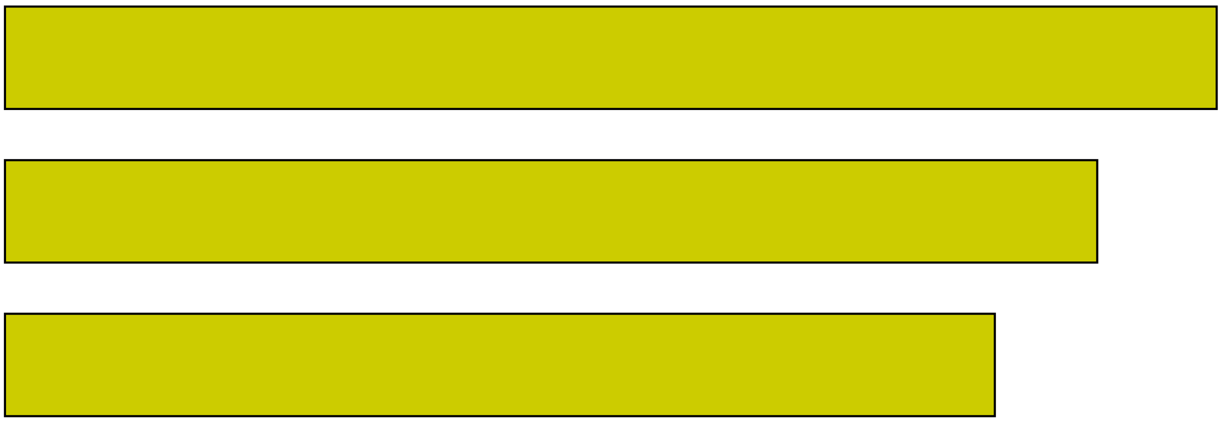
$$= n + n/2 + n/4 + n/8 + \ldots\ldots$$

$$= n(1 + 1/2 + 1/4 + \ldots\ldots\ldots) = n.2 = O(n)$$

Running time of Selection?

Worst case?

Each call to Partition only reduces our search by 1



Recurrence?

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

$O(n^2)$

Running time of Selection?

Worst case?

Each call to Partition only reduces our search by 1

When does this happen?

- sorted
- reverse sorted
- others…

How can randomness help us?

```
RSelection(A, k, p, r)
  q <- RandomizedPartition(A,p,r)
  if k = q
    Return A[q]
  else if k < q
    Return Selection(A, k, p, q-1)
  else // k > q
    Return Selection(A, k, q+1, r)
```

Running time of RSelection?

Best case

- O(n)

Worst case

- Still O(n^2)
- As with Quicksort, we can get unlucky

Average case?

O(n)

Summary of Randomized Select

- 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 Select

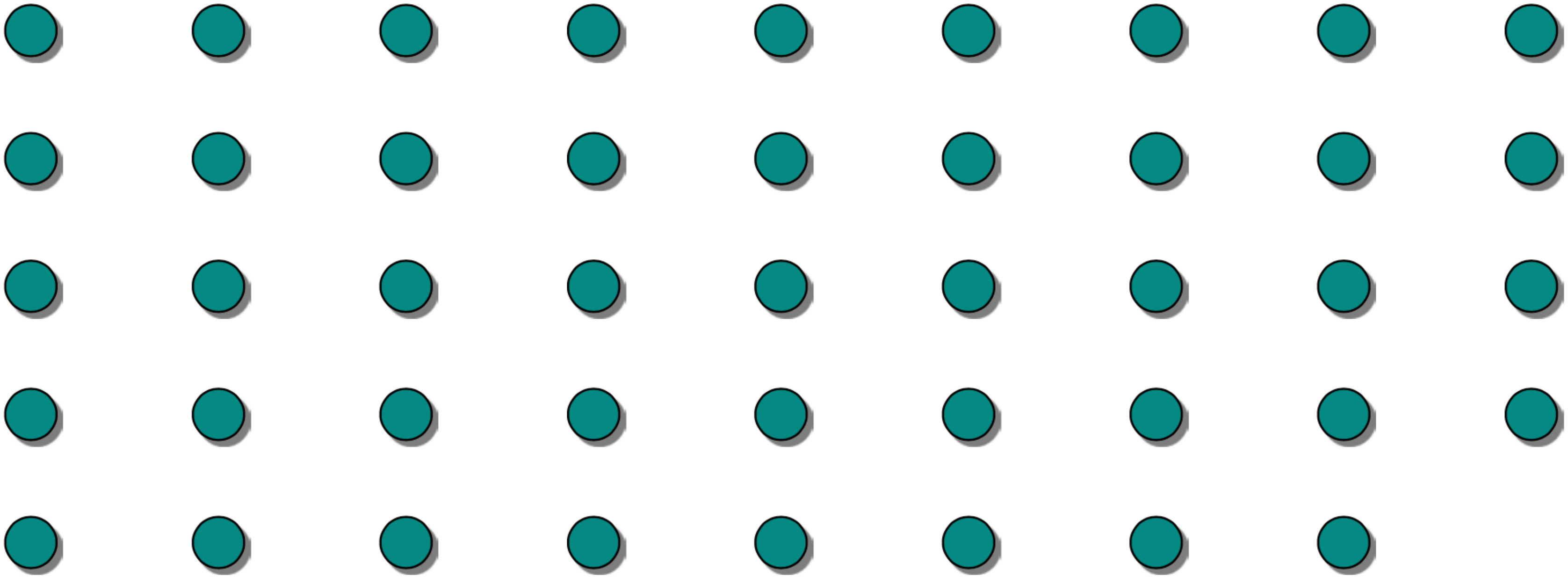
SELECT(i, n)

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

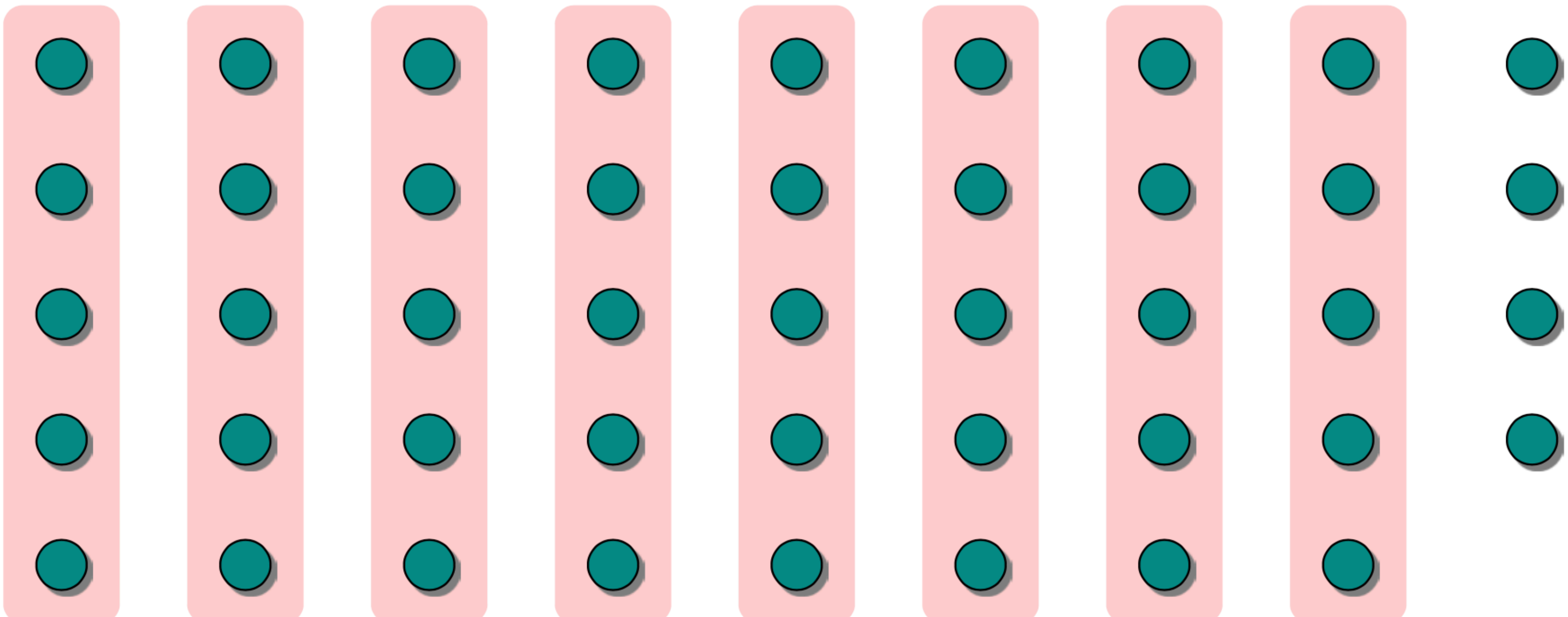
actually $\lceil n/5 \rceil$

Same as RAND-SELECT

Choosing the Pivot

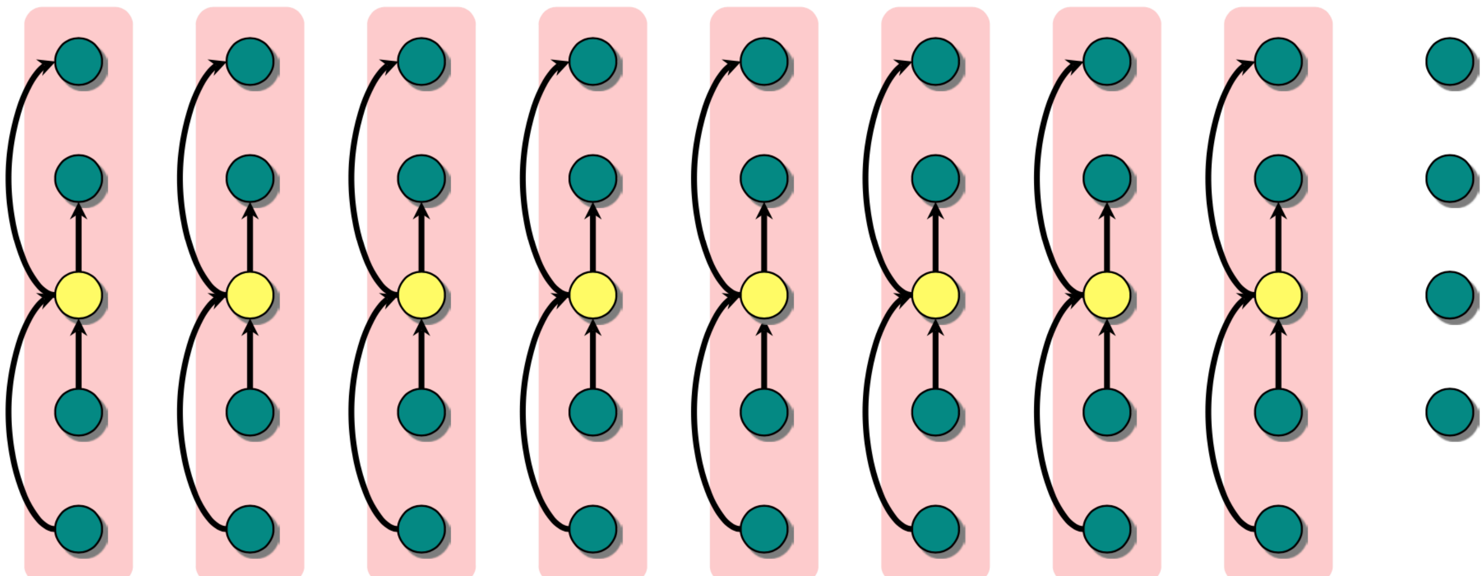


Choosing the Pivot

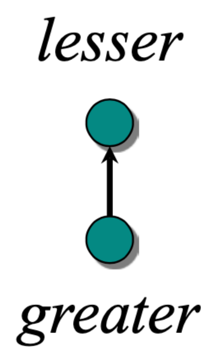


1. Divide the n elements into groups of 5.

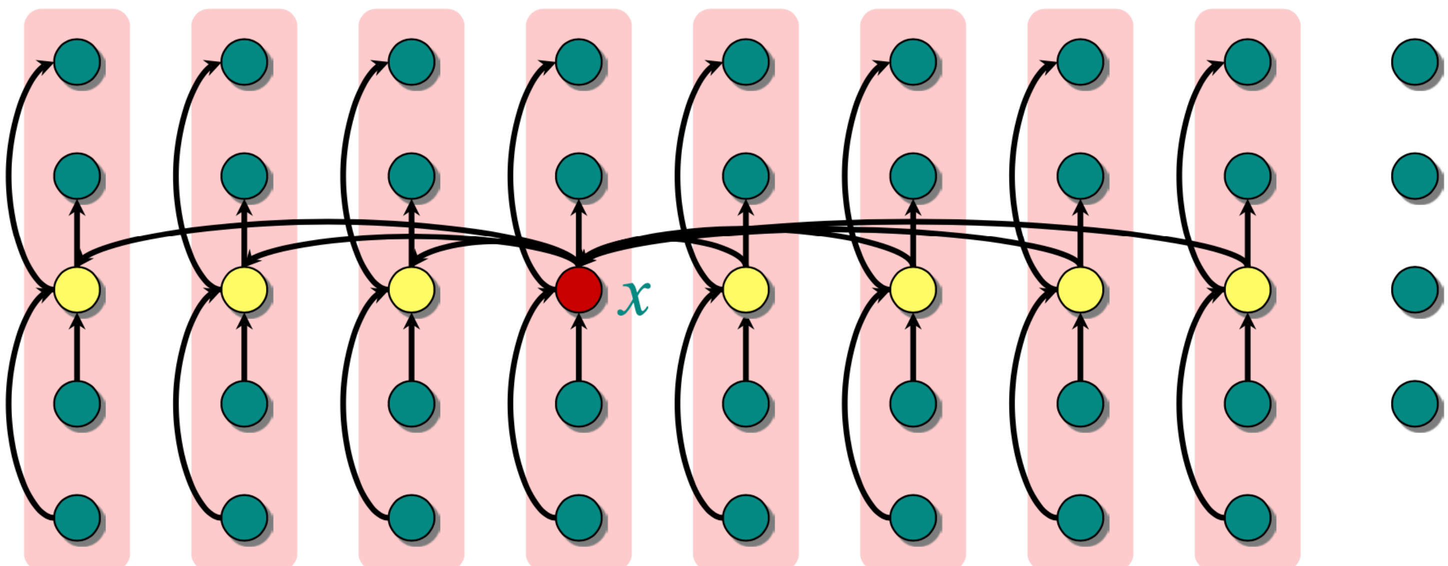
Choosing the Pivot



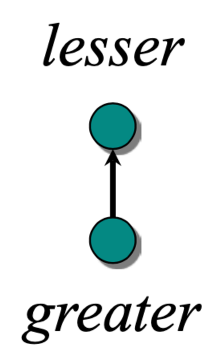
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.



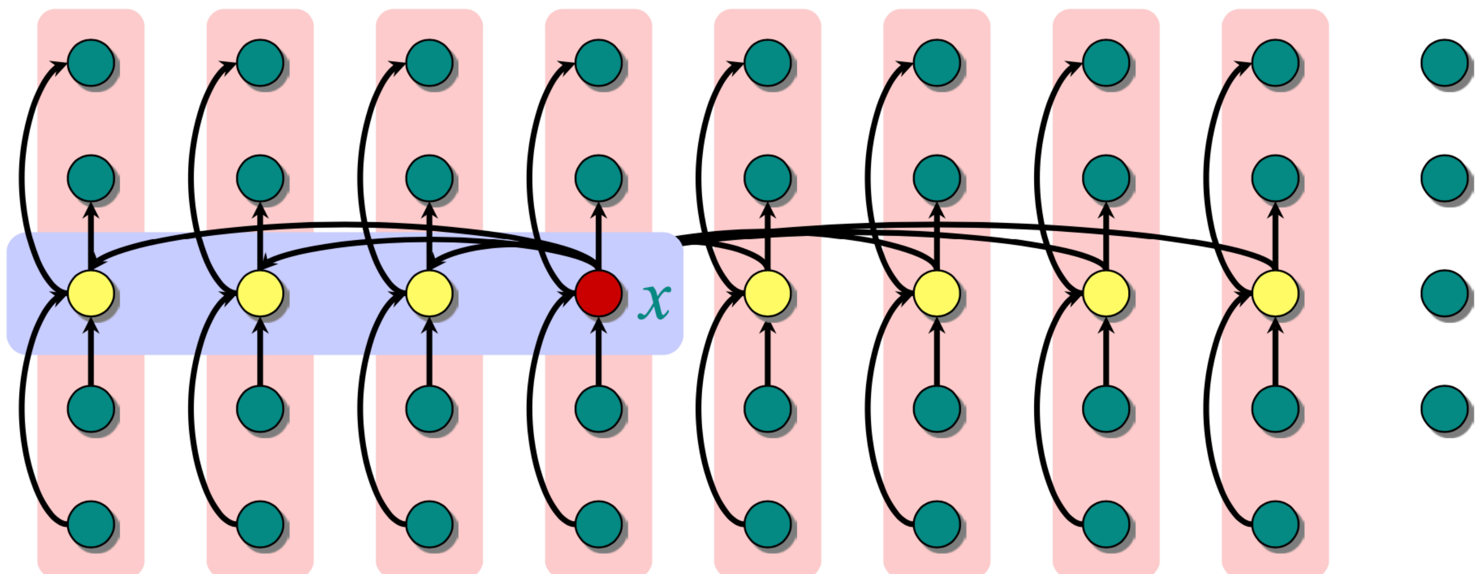
Choosing the Pivot



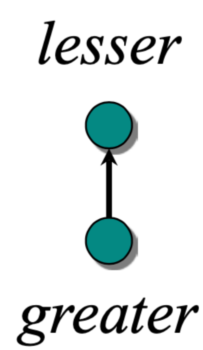
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.



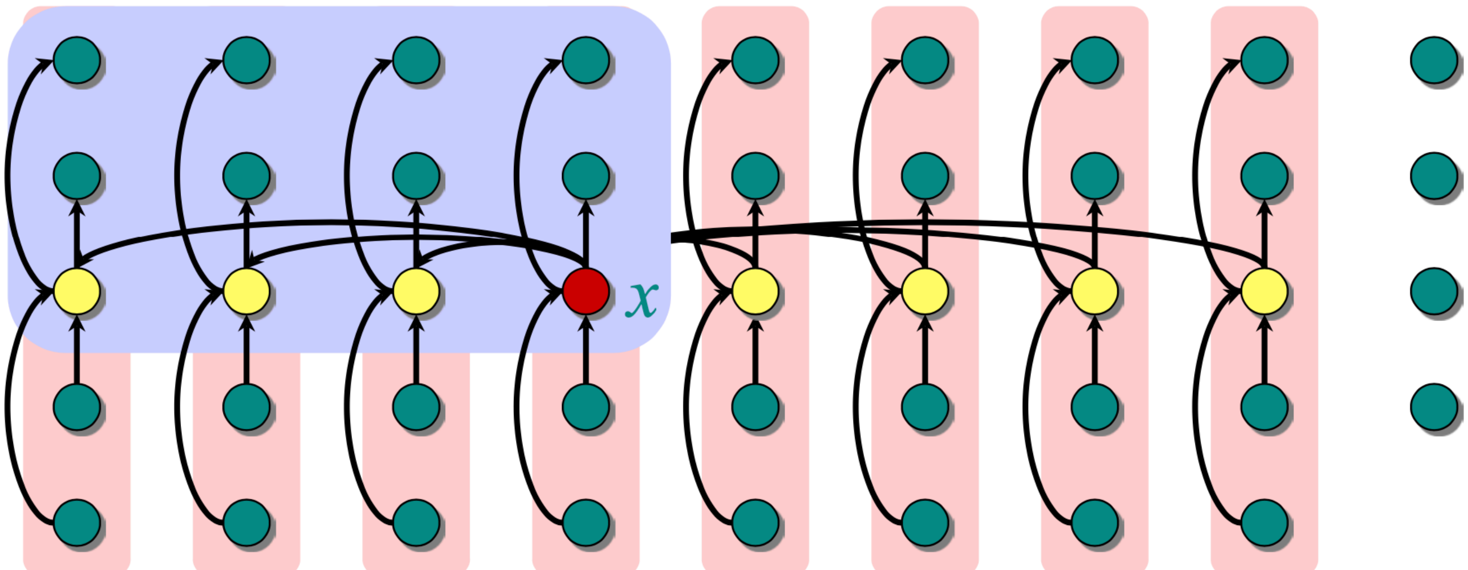
Analysis



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.

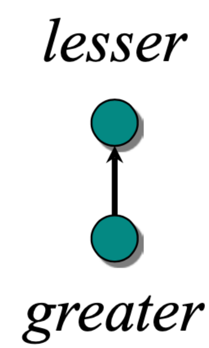


Analysis

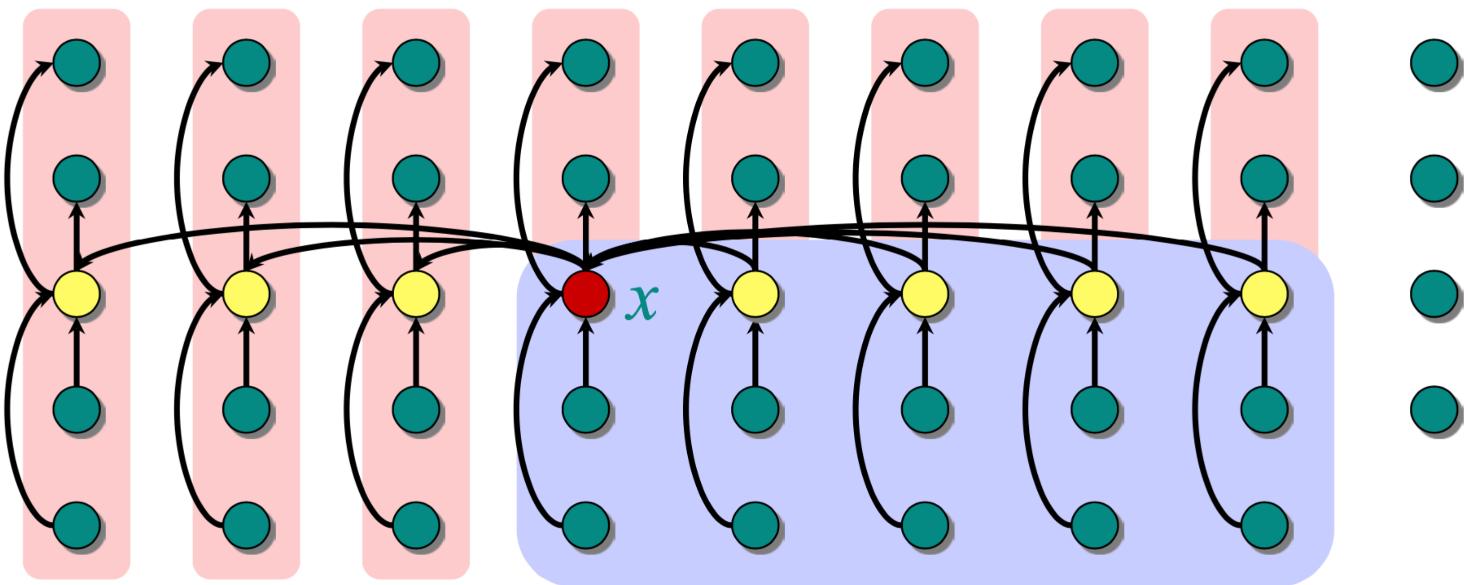


At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

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

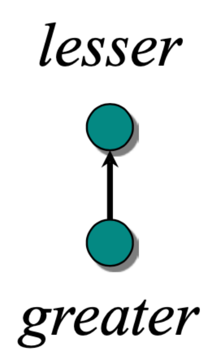


Analysis



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



Simplification Using a Magic Constant (50)

- 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 $T(n) = \Theta(1)$.

Establish Recursion

$T(n)$ — 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 Recursion

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

Substitution:

$T(n) \le cn$

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

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

Conclusion

- 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.