Randomized (Linear-Time) Selection

https://cs.pomona.edu/classes/cs140/

Selection Problem

Input: A set of n numbers and an integer i, with $1 \leq i \leq n$

Output: The element that is larger than exactly $i - 1$ other elements

- Known as the i^{th} order statistic or the i^{th} smallest number
- The minimum element is the 1^{st} order statistic ($i = 1$)
- The maximum element is the n^{th} order statistic ($i = n$)

What is "i" for the median? (an expression base on n)

- If n is even, then the medians are the $n/2$ and $n/2 + 1$ order statistics
- If n is odd, then the median is the $(n + 1)/2$ order statistic

Reduction

Find the i^{th} smallest number in an array

- Recall: it takes linear time just to read an array
- What would be a O(n lg n) algorithm for this problem?
  - Sort
  - Return the element at index i - 1

Selection Problem

- Can we do better than O(n lg n)?
- To beat O(n lg n) we cannot sort the entire array
  - Note: comparison-based sorting **cannot** be done faster than O(n lg n)
  - (you'll prove this later)

- Do we even need to perform a comparison with all the elements?
  - Yes!
  - So we know that O(n) is our lower bound on the running time
  - What is the upper bound?

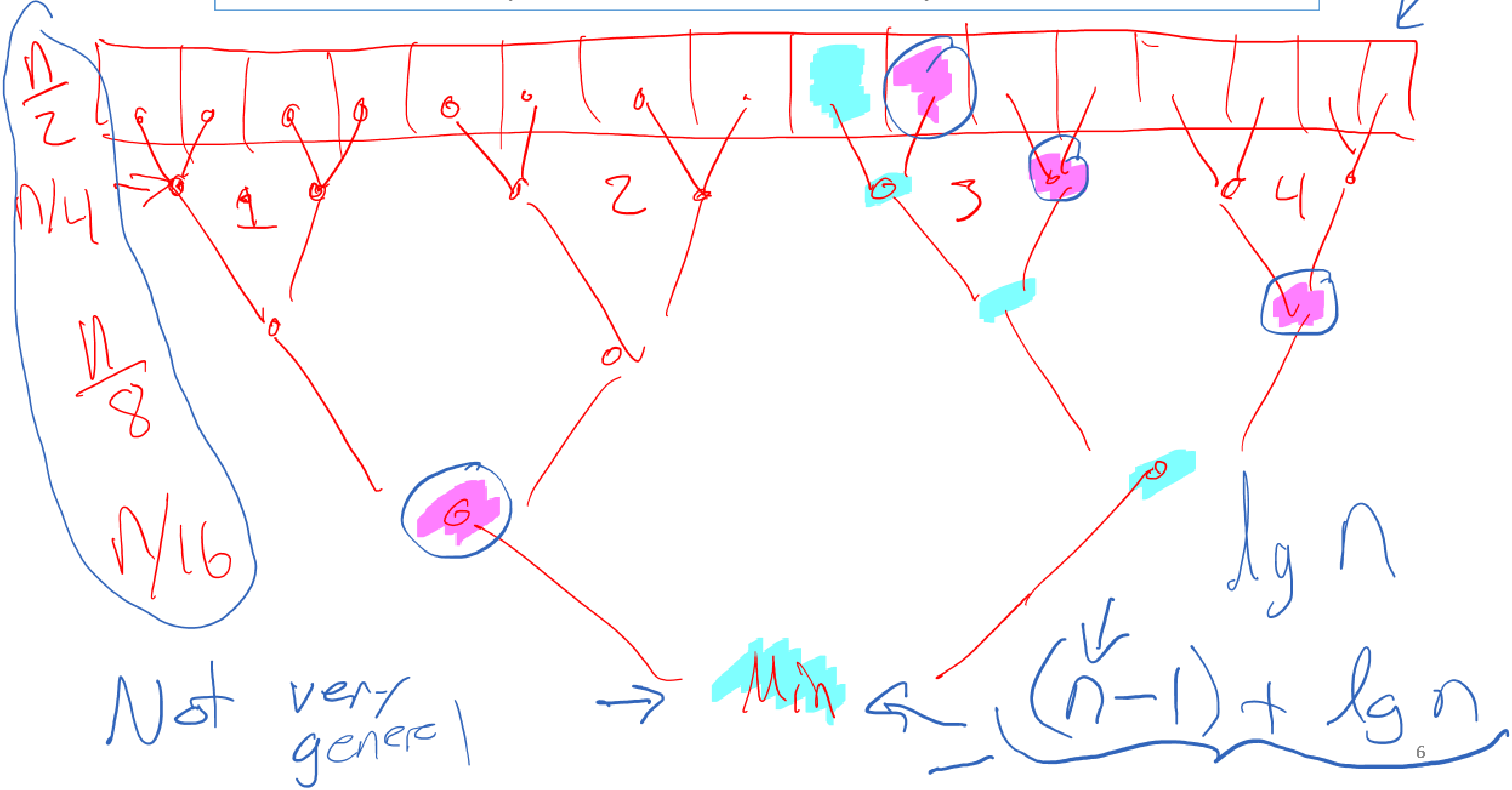
Finding the minimum (1st) and maximum (nth)

```
FUNCTION FindMinimum(array)

   n = array.length
   min_val = array[0]

   FOR val IN array[1 ..< n]
      IF val < min_val
         min_val = val

   RETURN min_val
```

```
FUNCTION FindMaximum(array)

   n = array.length
   max_val = array[0]

   FOR val IN array[1 ..< n]
      IF val > max_val
         max_val = val

   RETURN max_val
```

What would an algorithm look like for finding the 2nd order statistic?

What would an algorithm look like for finding the 2nd order statistic?



General Algorithm

- How do you find the ith order statistic in the more general case?

Randomized Selection (Quickselect)

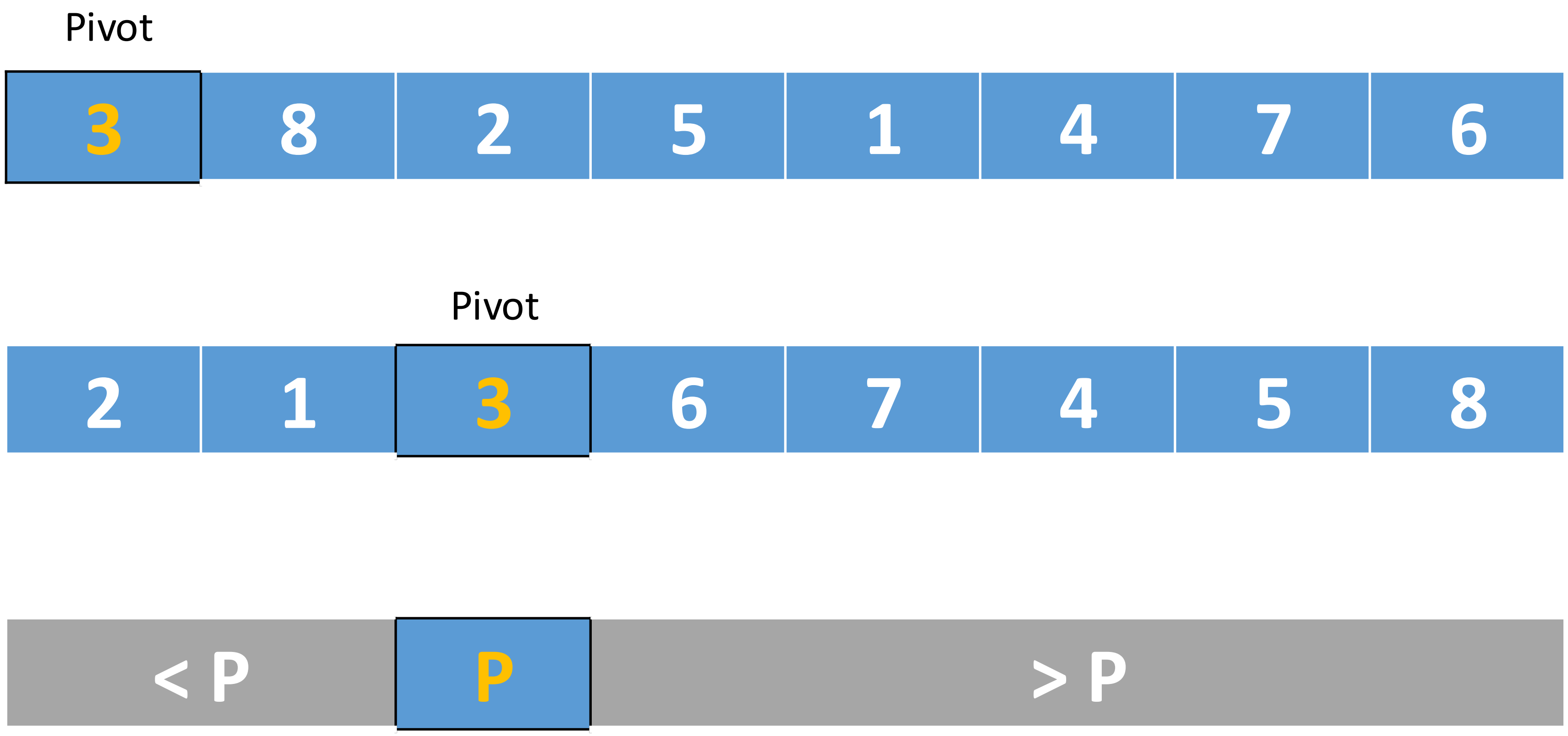
(Expected) linear-time complexity

- Same as scanning through the list once

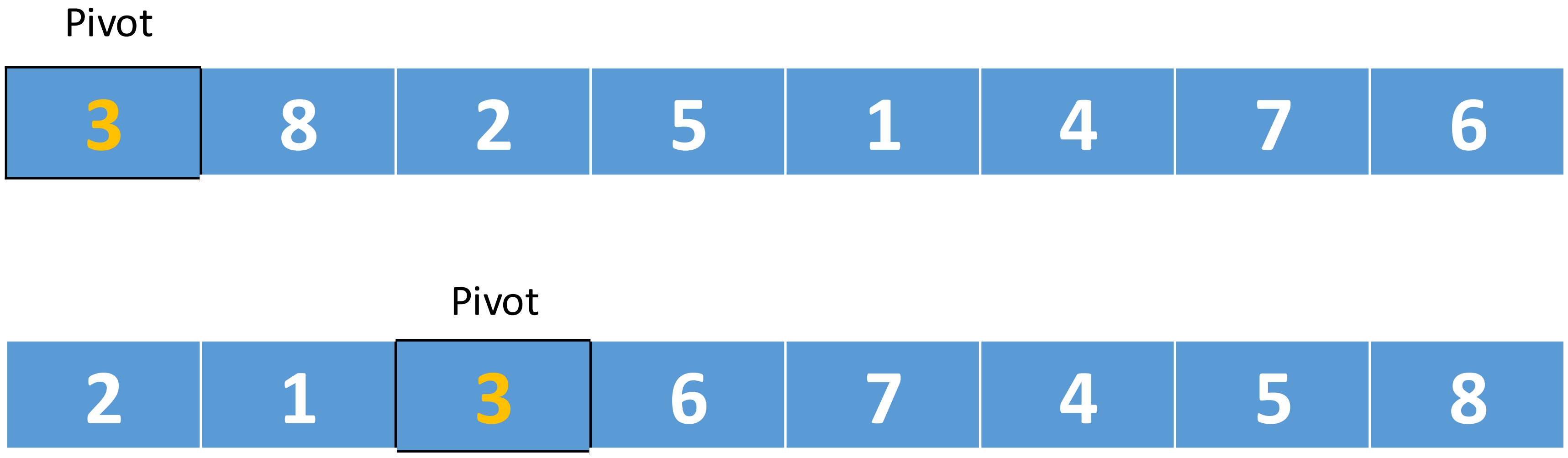
One linear algorithm for this problem is called Randomized Selection

We are going to start by modifying the only randomized algorithm that we we've seen so far: Quicksort

Key Component of Quicksort: Partitioning



Key Component of Quicksort: Partitioning



<u>What if we are looking for the 5th order statistic?</u>

- What is the fifth order statistic?
- Do we need to recursively look on both sides of the pivot?

Quicksort

How do we turn this into Quickselect?

```
FUNCTION QuickSort(array, left_index, right_index)

  IF left_index ≥ right_index
    RETURN
  MovePivotToLeft(left_index, right_index)

  pivot_index = Partition(array, left_index, right_index)

  QuickSort(array, left_index, pivot_index)
  QuickSort(array, pivot_index + 1, right_index)
```

```
FUNCTION Partition(array, left_index, right_index)

  pivot_value = array[left_index]

  i = left_index + 1
  FOR j IN [left_index + 1 ..< right_index]
    IF array[j] < pivot_value
      swap(array, i, j)
      i = i + 1

  swap(array, left_index, i - 1)
  RETURN i - 1
```

Quickselect

How do we turn this into Quickselect?

```
FUNCTION RSelect(array, left_index, right_index, ith)
  IF left_index == right_index
    RETURN array[left_index]

  MovePivotToLeft(left_index, right_index)
  pivot_index = Partition(array, left_index, right_index)

  IF ith == pivot_index + 1
    RETURN array[pivot_index]

  ELSE IF ith < pivot_index + 1
    RETURN RSelect(array, left_index, pivot_index, ith)

  ELSE
    RETURN RSelect(array, pivot_index + 1, right_index, ith)
```

```
FUNCTION Partition(array, left_index, right_index)

  pivot_value = array[left_index]

  i = left_index + 1
  FOR j IN [left_index + 1 ..< right_index]
    IF array[j] < pivot_value
      swap(array, i, j)
      i = i + 1

  swap(array, left_index, i - 1)
  RETURN i - 1
```

```
FUNCTION RSelectIter(array, left_index, right_index, ith)
    LOOP
        IF left_index == right_index
            RETURN array[left_index]

        MovePivotToLeft(left_index, right_index)
        pivot_index = Partition(array, left_index, right_index)

        IF ith == pivot_index + 1
            RETURN array[pivot_index]

        ELSE IF ith < pivot_index + 1
            right_index = pivot_index

        ELSE
            left_index = pivot_index + 1
```

Iterative Version

1. Asymptotically, do you expect any differences?
2. Practically, which do you think has better performance?

Running time of Quickselect

- What is the worst possible running time?
  - What is the worst pivot choice?

- What is the best possible running time?

- You're more likely to finish the algorithm in linear time than in quadratic (worst-case) time.

- Under normal operation, what is the best choice for a pivot?

Running time of Quickselect

- Under normal operation, what is the best choice for a pivot?
- What is the running time if we always **deterministically** pick the median?
- How would you calculate the running time if I told you that the algorithm always picked the median?

$$T(n) \leq T\left(\frac{n}{2}\right) + O(n) \rightarrow O(n)$$

Quickselect

Hope: that the choice of pivot is “good enough” “often enough”

Theorem:

for every input array, the **average** running time of Quickselect is O(n)

Quickselect

Where is all the work done?

```
FUNCTION RSelect(array, left_index, right_index, ith)
    IF left_index == right_index
        RETURN array[left_index]

    MovePivotToLeft(left_index, right_index)
    pivot_index = Partition(array, left_index, right_index)

    IF ith == pivot_index + 1
        RETURN array[pivot_index]

    ELSE IF ith < pivot_index + 1
        RETURN RSelect(array, left_index, pivot_index, ith)

    ELSE
        RETURN RSelect(array, pivot_index + 1, right_index, ith)
```

```
FUNCTION Partition(array, left_index, right_index)

    pivot_value = array[left_index]

    i = left_index + 1
    FOR j IN [left_index + 1 ..< right_index]
        IF array[j] < pivot_value
            swap(array, i, j)
            i = i + 1

    swap(array, left_index, i - 1)
    RETURN i - 1
```

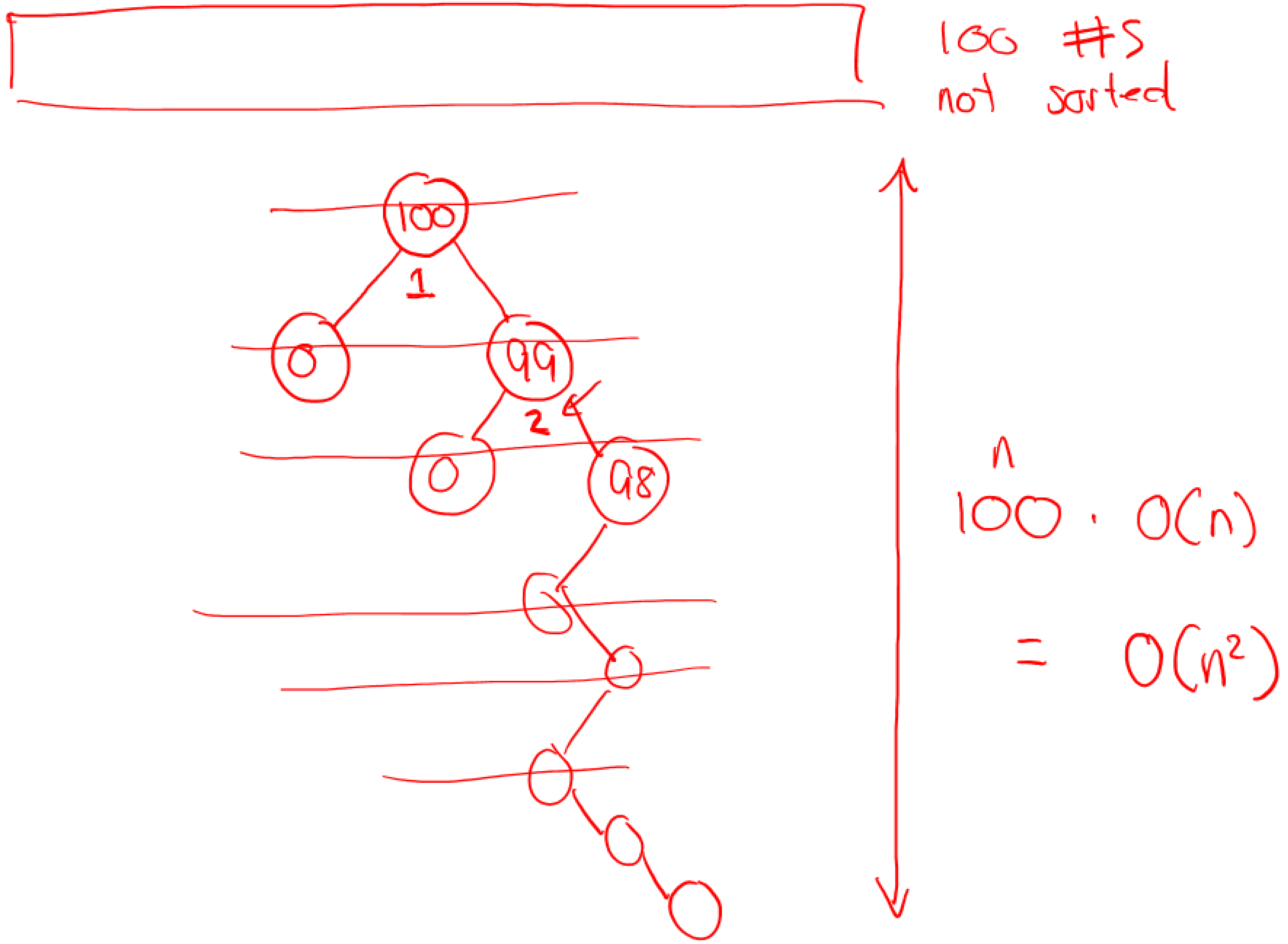
Quickselect

Hope: that the choice of pivot is “good enough” “often enough”

Theorem:

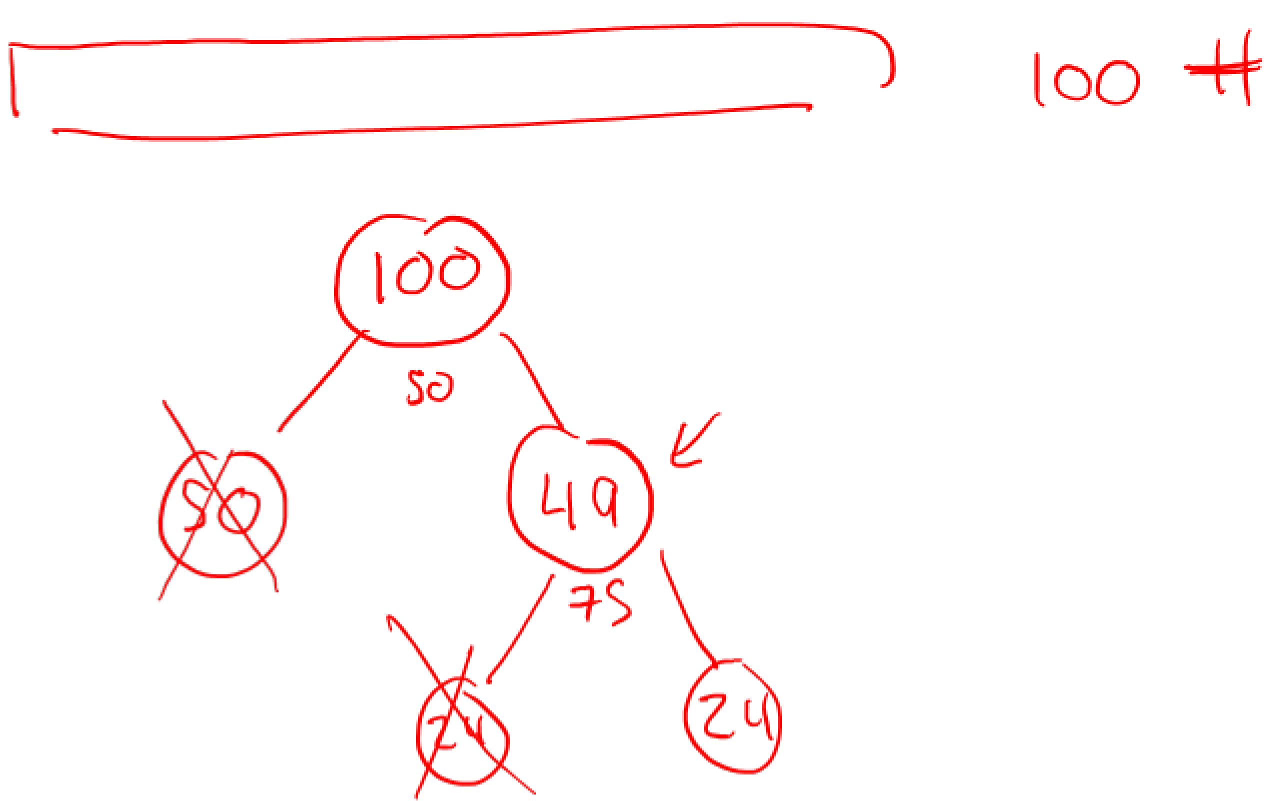
for every input array, the **average** running time of Quickselect is O(n)

All the work is done in partition and the total amount of work done in a single call to partition is $\leq cn$



100 #s
not sorted
100
1
0
99
2
0
98
n
100 · O(n)
= O(n²)

Best Case



$O(n)$

Notation for our proof

Quickselect is in *$phase_j$* if the current subarray size is between

$$n \cdot \left(\frac{3}{4}\right)^{j+1} < (\text{right\_index} - \text{left\_index}) \leq n \cdot \left(\frac{3}{4}\right)^{j}$$

- j = 0 : 0.75n to 1.00n
- j = 1 : 0.56n to 0.75n
- j = 2 : 0.42n to 0.56n
- j = 3 : 0.32n to 0.42n

- Multiple recursive calls can be in the same *$phase_j$*
- Some phases can be skipped

Phase 0

0.75n to 1.00n

Phase 1

0.56n to 0.75n

Phase 2

0.42n to 0.56n

Phase 3

0.32n to 0.42n

Phase 4

0.24n to 0.32n

Proof

- Multiple recursive calls can be in the same $phase_j$
- Some recursive calls can skip over the next $phase_j$
- Let X_j denote the number of recursive calls made during $phase_j$
  - Note: each recursive call in turn calls `Partition`
- What is the total running time of Quickselect?

$$T(n) \leq \sum_{phase_j} X_j c \left(\frac{3}{4}\right)^j n$$

$$T(n) \leq \sum_{phase_j} X_j c \left(\frac{3}{4}\right)^j n$$

- X_j : number of calls in *$phase_j$* (not an indicator variable)
- c : constant for amount of work done by `Partition`
- $(¾)^j n$: upper bound on subarray size during *$phase_j$*
- $c(¾)^j n$: total amount of work during a single call in *$phase_j$*

Probability of leaving a phase?

$n\left(\frac{3}{4}\right)^{j+1}, n\left(\frac{3}{4}\right)^{j}$

- We leave *phase$_j$* when our pivot is within the 25-75% middle part of the subarray.

- So, if both the left and right sides contain less than 75% of elements
- What is the probability that we choose a pivot in (25 .. 75]?

Probability of leaving a phase?

- We have (at worst) a 50% chance to pick a "good" pivot element.
- Which means that we leave the current $phase_j$.

- So, we have reduced our problem to the coin flip problem:

$$E[X_j] \leq E[\#\ of\ coin\ flips\ to\ get\ heads]$$

Heads : good pivot
Tails : bad pivot

Coin Flips

- Let N = the number of flips until you get a heads (a geometric random variable)

$E[N] = 1 + \frac{1}{2} E[N]$

$E[N] = 1 + \frac{1}{2} + \frac{1}{4} E[N]$

$E[N] = 1 + \frac{1}{2} + \frac{1}{4} + 1/8\ E[N]$

$E[N] = 1 + \frac{1}{2} + \frac{1}{4} + 1/8 + 1/16\ E[N]$

…

Need at least one flip. Then we have a 50% chance that it was tails and we need to flip again.

Coin Flips

- Let N = the number of flips until you get a heads (a geometric random variable)

$E[N] = 1 + \frac{1}{2} E[N]$

$E[N] = 1 + \frac{1}{2} + \frac{1}{4} E[N]$

$E[N] = 1 + \frac{1}{2} + \frac{1}{4} + 1/8\ E[N]$

$E[N] = 1 + \frac{1}{2} + \frac{1}{4} + 1/8 + 1/16\ E[N]$

E[N] = 2

Need at least one flip. Then we have a 50% chance that it was tails and we need to flip again.

Coin Flips

- Let N = the number of flips until you get a heads
  (a geometric random variable)

- Alternatively, what is the expected number of heads per coin flip?

$$E[\#of\ heads\, per\ flip] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 = \frac{1}{2}$$

$$Total_{heads} = E[\#of\ heads\, per\ flip] \cdot N$$

$1 = \frac{N}{2}$ → N=2

Back to the proof

$$T(n) \leq \sum_{phase_j} X_j c \left(\frac{3}{4}\right)^j n$$

$$E[T(n)] \leq cn \sum_{phase_j} E[X_j] \left(\frac{3}{4}\right)^j$$

Expected Value of T

$$E[T(n)] \leq E\left[\sum_{phase\,j} X_j\, c\left(\frac{3}{4}\right)^j n\right]$$

$$E[T(n)] \leq cn \sum_{phase\,j} E[X_j]\left(\frac{3}{4}\right)^j$$

?

Expected Value of T

$$E[T(n)] \leq E\left[\sum_{phase\, j} X_j \, c \left(\frac{3}{4}\right)^j n\right]$$

$$E[T(n)] \leq cn \sum_{phase\, j} E[X_j] \left(\frac{3}{4}\right)^j$$

2

Expected Value of T

$$E[T(n)] \leq E\left[\sum_{phase\,j} X_j\, c\left(\frac{3}{4}\right)^j n\right]$$

$$E[T(n)] \leq 2cn \sum_{phase\,j}\left(\frac{3}{4}\right)^j$$

Converges to **4**

We saw the same summation in our proof of the master theorem

Geometric sequence

$$\leq \frac{1}{1-r} = \frac{1}{1-\left(\frac{3}{4}\right)} = 4$$

$$E[T(n)] \leq 2cn4 = c_{combined}n = O(n)$$

Running time of Quickselect

$$E[T(n)] \leq c_{combined}n = O(n)$$

Thus, the ***average*** running time of Quickselect T(n) <= O(n)