DSC 40B

Lecture 11 : The Median and Order Statistics

The Median   mic

- How fast can we find a **median** of n numbers?

Modify selection_sort so that it computes a **median** of the input array. What is the time complexity?

```python
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(n-1):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
                arr[min_ix], arr[barrier_ix]
        )
```

Algorithms

- We have seen several ways of computing a median:
  - Alg. 1: Minimize absolute error, **brute force**.
  - Alg. 2: Use definition (half ≤, half ≥).
  - ...

Best so far...

- Sort the list with `mergesort`, return middle element.
- Time complexity: $\Theta(n \log n)$.

Is sorting necessary?

- Need to sort the whole list just to find middle?
- Seems like **more work** than necessary.

Today

- We'll design an algorithm which runs in $\Theta(n)$ expected time.
- Much more useful than *just* finding median...

Order Statistics, definition

- The **median** is an *example* of an order statistic.
- Given n numbers, the **kth order statistic** is the kth *smallest* number in the collection.

Given n numbers, the **kth order statistic** is the kth *smallest* number in the collection.

Example

[99, 42, -77, -12, 101]

- 1st order statistic: ?
- 2nd order statistic: ?
- 4th order statistic: ?

Given n numbers, the **kth order statistic** is the kth *smallest* number in the collection.

Example

[99, 42, -77, -12, 101]

- 1st order statistic: -77
- 2nd order statistic: ?
- 4th order statistic: ?

Given n numbers, the **kth order statistic** is the kth *smallest* number in the collection.

Example

[99, 42, -77, -12, 101]

- 1st order statistic: -77
- 2nd order statistic: -12
- 4th order statistic: ?

Given n numbers, the **kth order statistic** is the kth *smallest* number in the collection.

Example

[99, 42, -77, -12, 101]

- 1st order statistic: -77
- 2nd order statistic: -12
- 4th order statistic: 99

Exercise

Some special cases of order statistics go by different names. Can you think of some?

Special Cases

- 1st order statistic: ?
- nth order statistic: ?
- $\lceil n/2 \rceil$th order statistic: ?
  - What if n is even?
- $\lceil \frac{p}{100} \cdot n \rceil$th order statistic: ?

Special Cases

- 1st order statistic: **minimum**
- nth order statistic: ?
- $\lceil n/2 \rceil$th order statistic: ?
    - What if n is even?
- $\lceil \frac{p}{100} \cdot n \rceil$th order statistic: ?

Special Cases

- 1st order statistic: **minimum**
- nth order statistic: **maximum**
- $\lceil n/2 \rceil$th order statistic: ?
    - What if n is even?
- $\lceil \frac{p}{100} \cdot n \rceil$th order statistic: ?

Special Cases

- 1st order statistic: **minimum**
- nth order statistic: **maximum**
- $\lceil n/2 \rceil$th order statistic: **median**
  - What if n is even?
- $\lceil p \;\; 100 \cdot n \rceil$th order statistic: ?

Special Cases

- 1st order statistic: **minimum**
- nth order statistic: **maximum**
- $\lceil n/2 \rceil$th order statistic: **median**
    - What if n is even?
- $\lceil \frac{p}{100} \cdot n \rceil$th order statistic: ***p*th percentile**

Goal

- **Fast** algorithm for computing *any* order statistic.
- Interestingly, some seem easier than others.
- Our algorithm will find **any** order statistic in $\boldsymbol{\Theta(n)}$.

Approach #1

- We can modify `selection_sort` to find the kth order statistic.
- **Loop invariant**: after kth iteration, first k elements are in final sorted order.

```python
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(n-1):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
                arr[min_ix], arr[barrier_ix]
        )
```

```python
def select_k(arr, k):
    """Find kth order statistic."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(k):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
                arr[min_ix], arr[barrier_ix]
        )
    return arr[k-1]
```

Approach #1

- 1st order statistic: $\Theta(n)$.
- nth order statistic: $\Theta(n^2)$.
- Median: $\Theta(n^2)$.
- kth order statistic: $\Theta(kn)$.

Exercise

- Describe how to find any order statistic in $\Theta(n \log n)$ time.

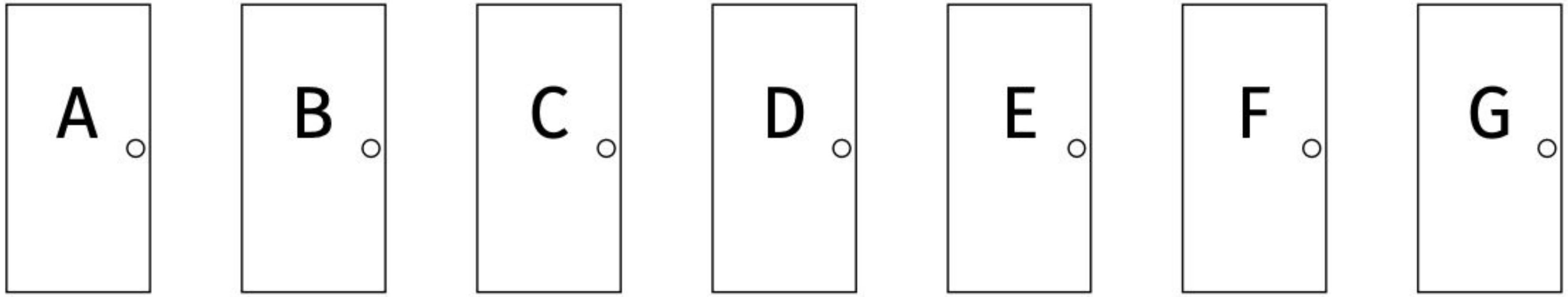
Approach #2

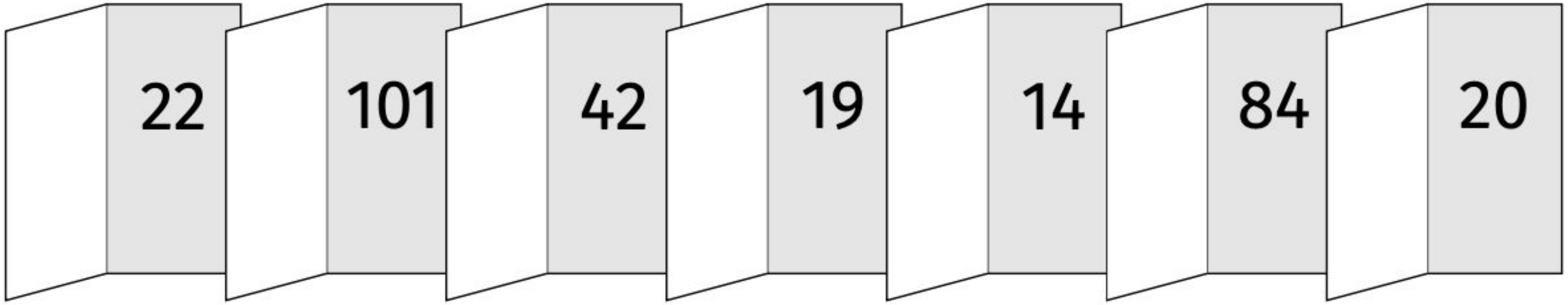
- Sort with `mergesort`, return arr[k-1]
- $\Theta(n \log n)$ time. Could be better...

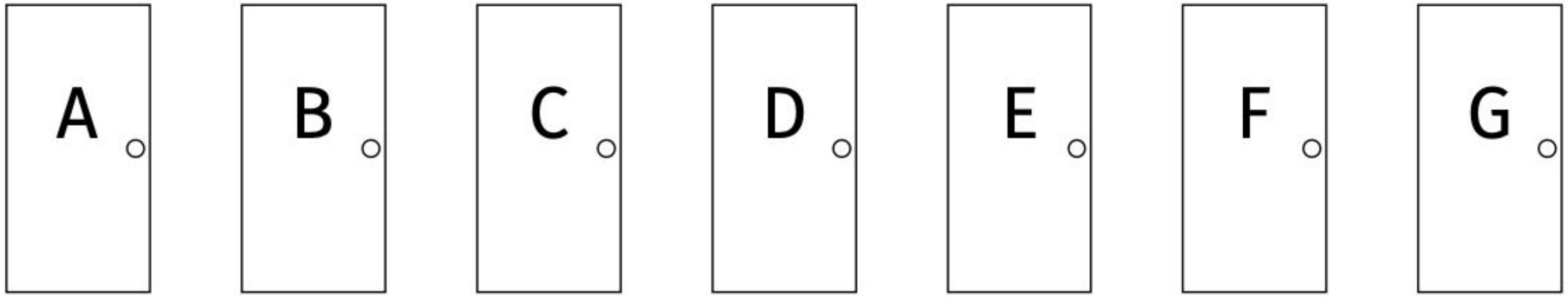
Quickselect

The Goal

- Given a collection of n numbers and an order, k.
- Find the kth smallest number in the collection.

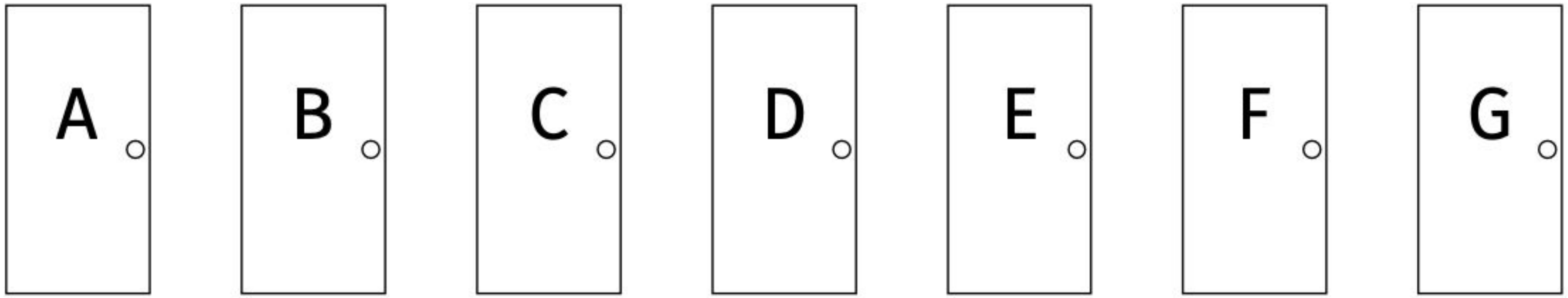

A
B
C
D
E
F
G

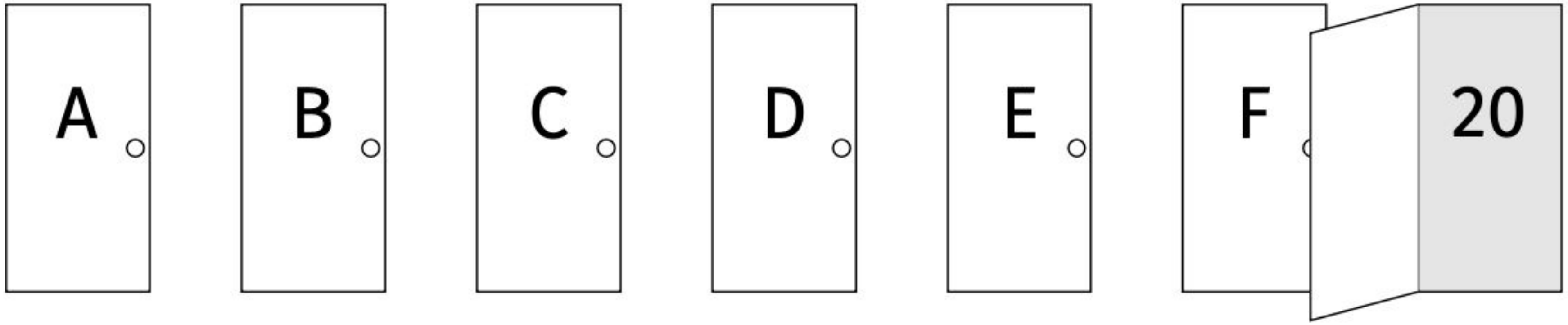

22
101
42
19
14
84
20

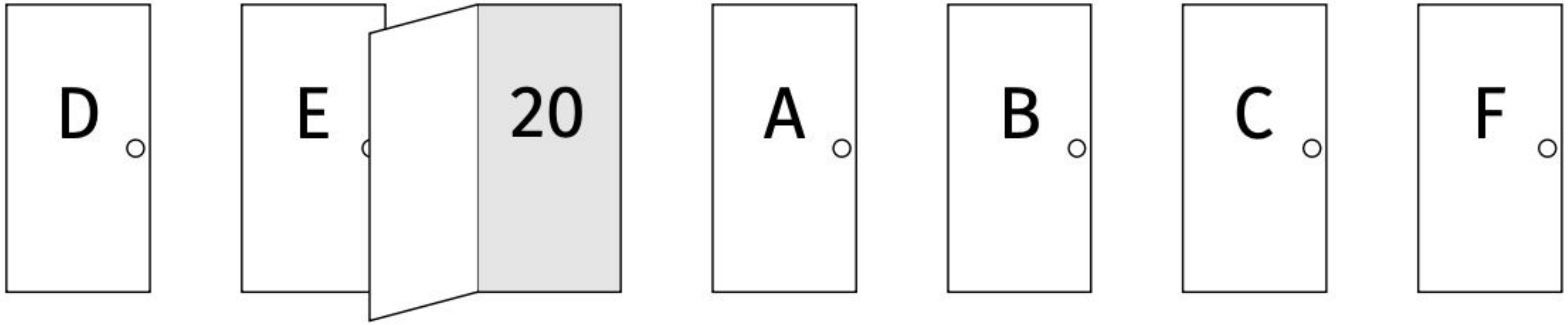

A
B
C
D
E
F
G

Game Show

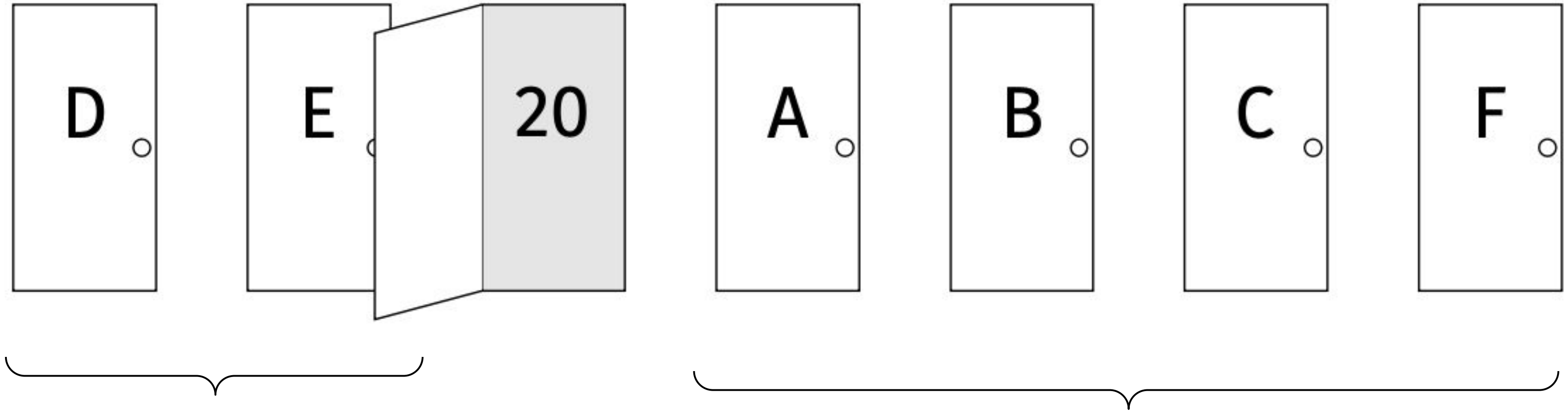
- **Goal**: tell the host the **largest** number.
- **Caution**: with every door opened, your money is reduced.
- **Twist**: After opening a door, the host tells you:
  - which doors are *smaller*.
  - which doors are *larger*.
  - they **partition** the doors into *higher* and *lower* by moving them.


A
B
C
D
E
F
G

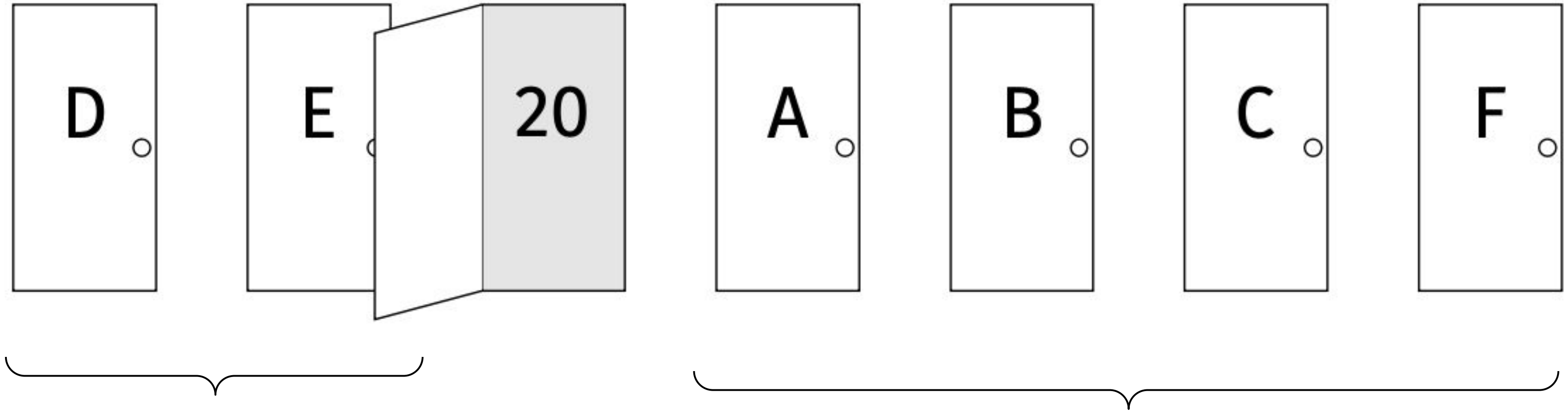

A
B
C
D
E
F
20



after partitioning

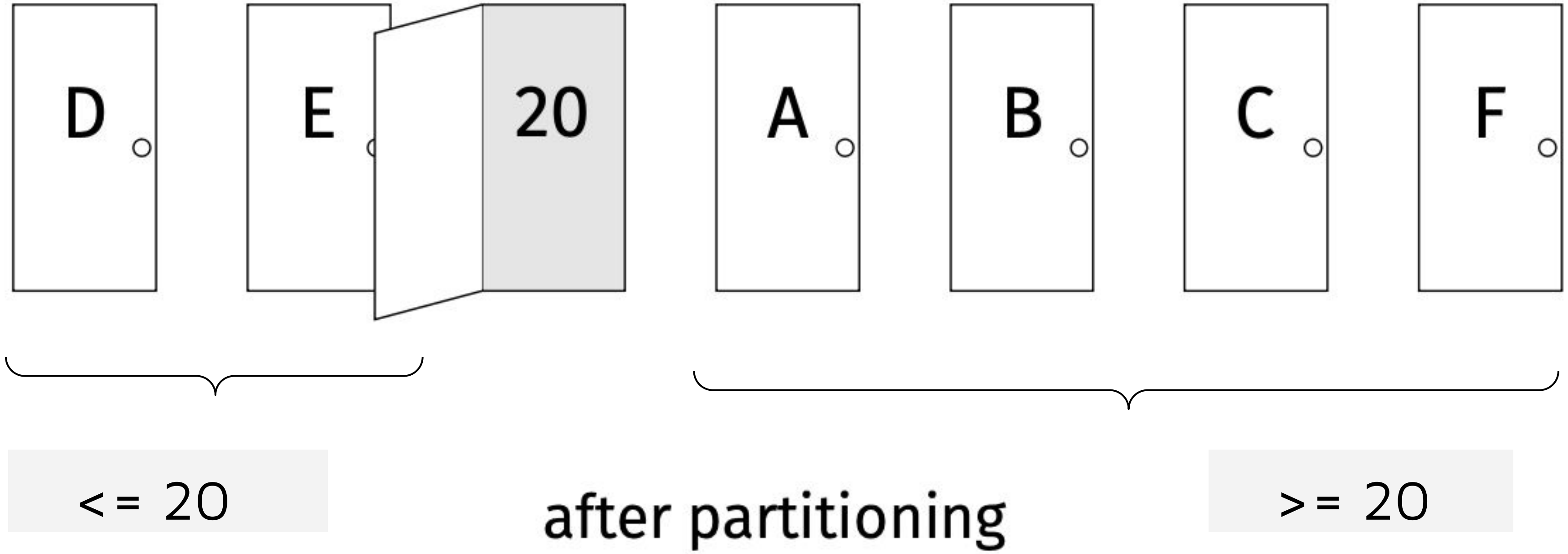


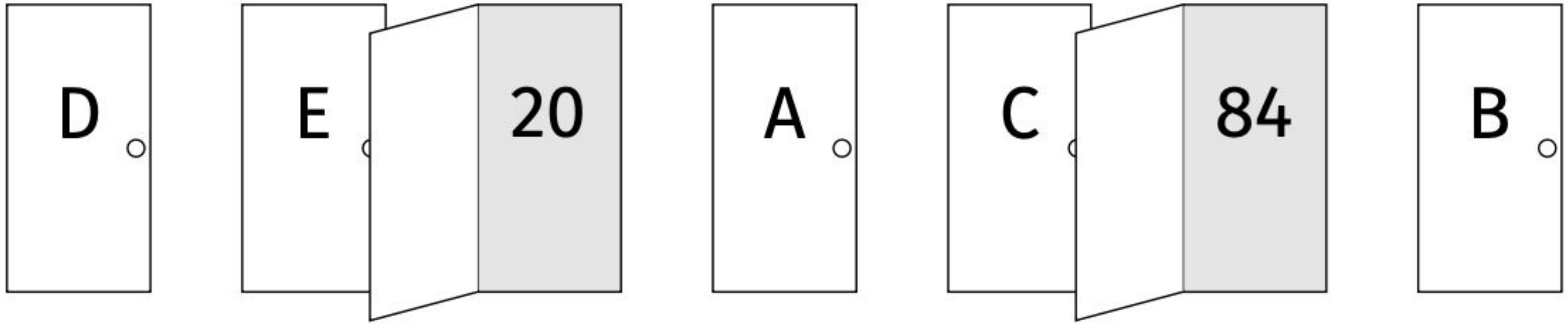
after partitioning



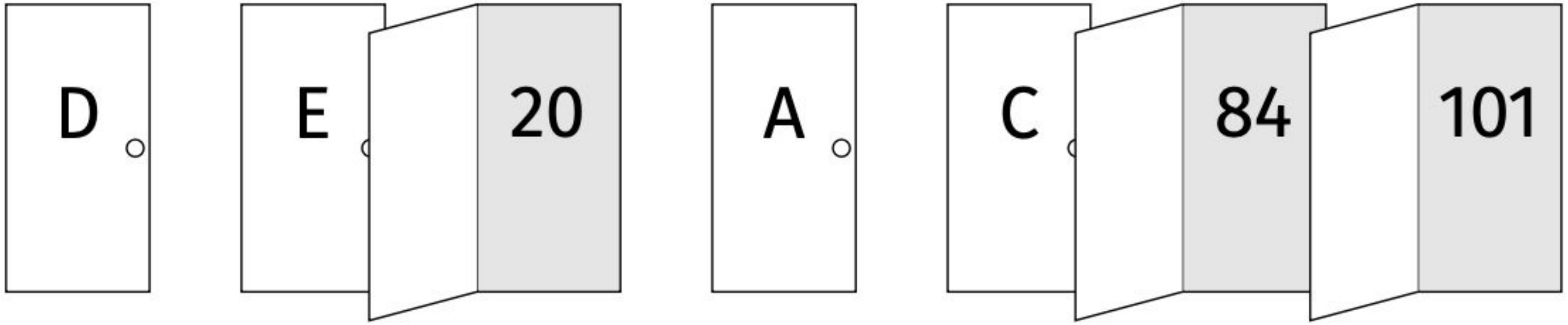
<= 20

after partitioning


D
E
20
A
B
C
F
<= 20
after partitioning
>= 20



after partitioning

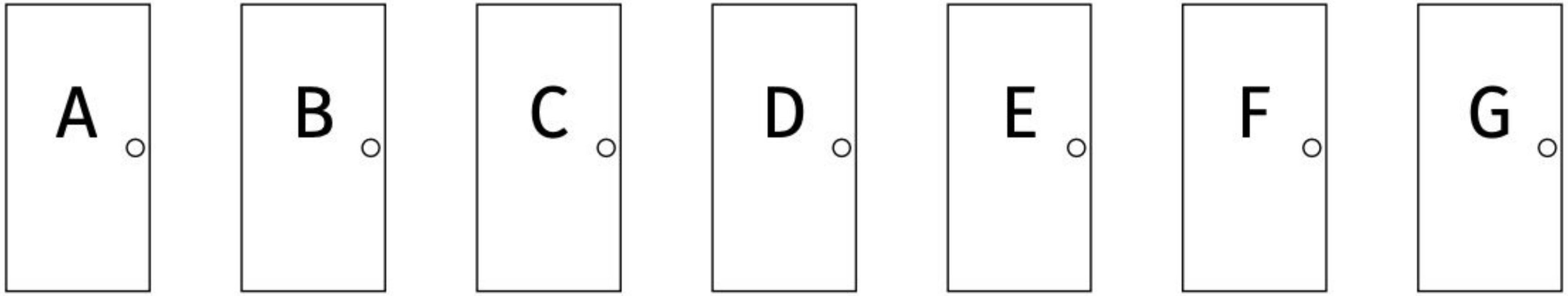

D
E
20
A
C
84
101

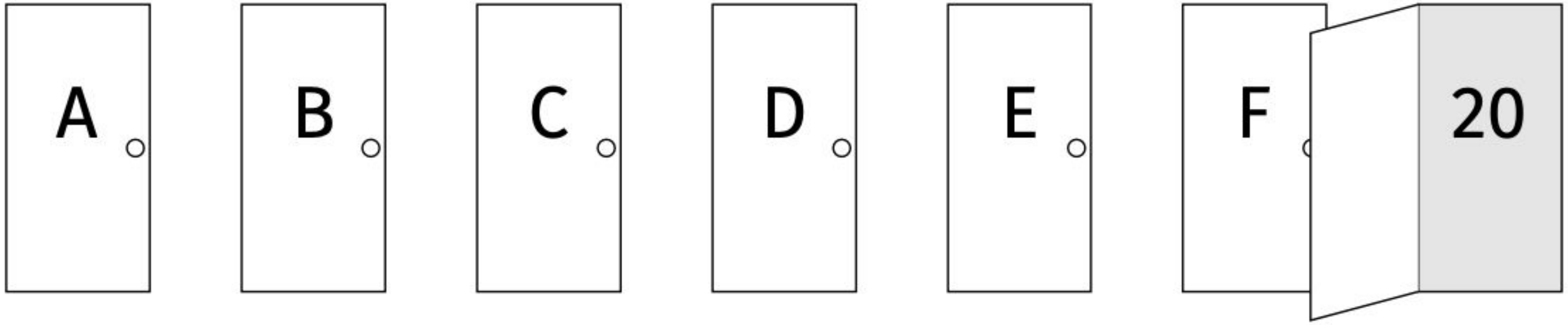
Main Idea

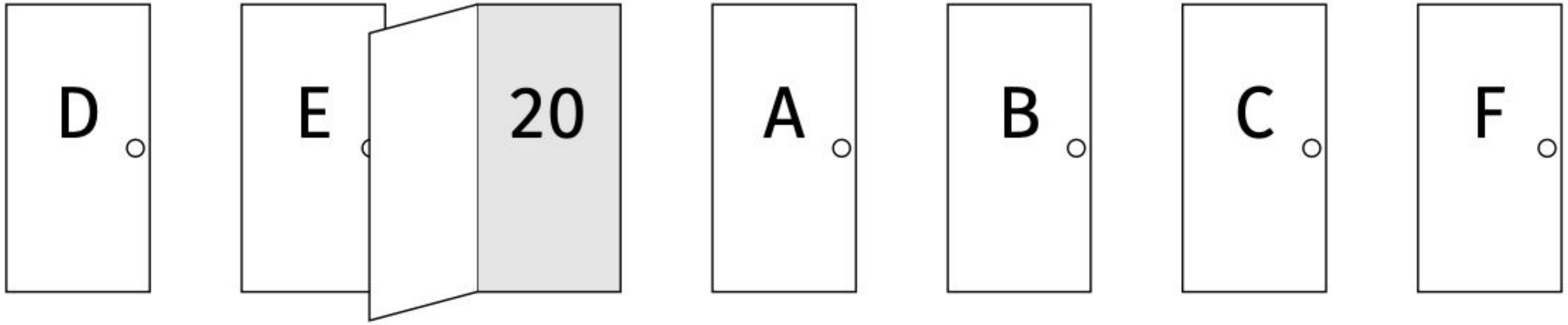
- After partitioning, the just-opened door is in the **correct place** in the sorted order (but the other doors may not be).
- But, every door to the left is *smaller* (≤), every door to the right is *larger* (≥).

In general…

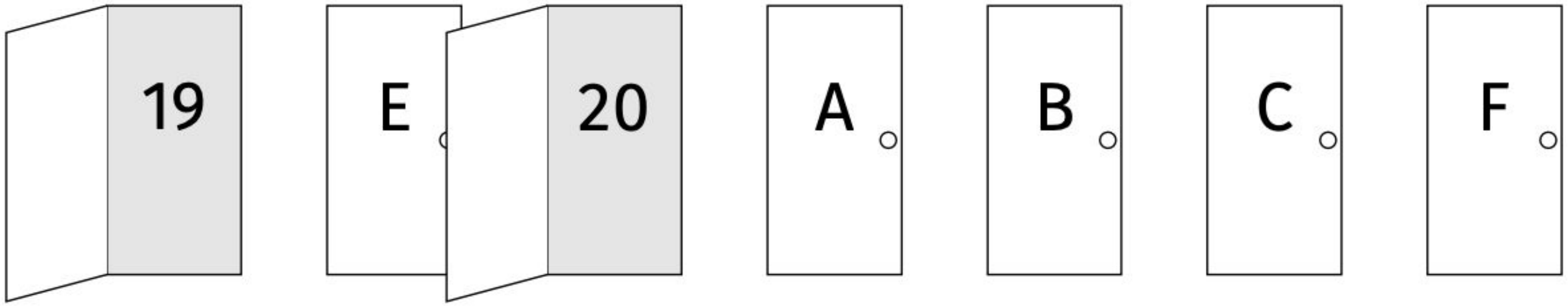
- Let’s generalize strategy for kth order statistic.
- Example: k = 2.

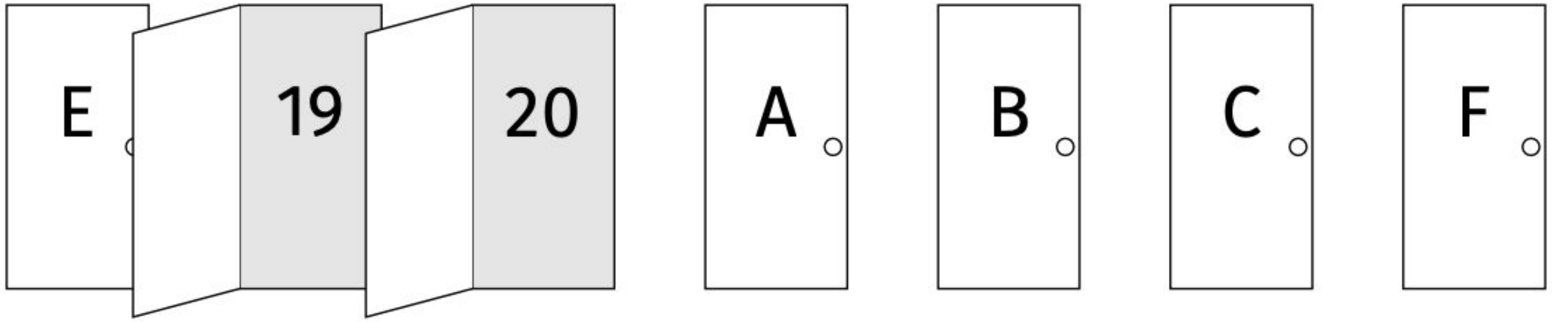

A
B
C
D
E
F
G


A
B
C
D
E
F
20



after partitioning


19
E
20
A
B
C
F



after partitioning

Strategy

- Open an **arbitrary** door (that hasn't been ruled out).
- **Partition** the doors around this number:
  - Move doors smaller than this to the left,
  - Larger than this to the right.
- Let p be our door's new position, k be the order we want.
  - If $p = k$, return this door.
  - If $p < k$, rule out doors to left.
  - If $p > k$, rule out doors to right.
- Repeat *recursively*.

```python
import random
def quickselect(arr, k, start, stop):
    """Finds kth order statistic in numbers[start:stop])"""
    pivot_ix = random.randrange(start, stop)
    pivot_ix = partition(arr, start, stop, pivot_ix)
    pivot_order = pivot_ix + 1
    if pivot_order == k:
        return arr[pivot_ix]
    elif pivot_order < k:
        return quickselect(arr, k, pivot_ix + 1, stop)
    else:
        return quickselect(arr, k, start, pivot_ix)
```

Example, k = 3

arr = [77, 42, 11, 99, 0, 101]

0 1 2 3 4 5

Example, k = 3

arr = [**77, 42, 11, 99, 0, 101**]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

Example, k = 3

arr = [77, 42, 11, 99, 0, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

pivot_index = ?

Example, k = 3

arr = [77, 42, 11, **99**, 0, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

pivot_index = 3, partition

Example, k = 3

arr = [77, 42, 11, **99**, 0, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

pivot_index = 3, partition

arr = [77, 42, 11, 0, **99**, 101]

0 1 2 3 4 5

Example, k = 3

arr = [77, 42, 11, **99**, 0, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

pivot_index = 3, partition

arr = [77, 42, 11, 0, **99**, 101]

0 1 2 3 4 5

pivot_index = 4

Example, k = 3

arr = [77, 42, 11, **99**, 0, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

pivot_index = 3, partition

arr = [**77, 42, 11, 0**, **99**, 101]

0 1 2 3 4 5

pivot_index = 4

Example, k = 3

arr = [77, 42, 11, **99**, 0, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

pivot_index = 3, partition

arr = [**77, 42, 11, 0**, **99**, 101]

0 1 2 3 4 5

pivot_index = 4

Example, k = 3

arr = [**77, 42, 11, 0**, 99, 101]

0 1 2 3 4 5

pivot_index = ?

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

Example, k = 3

arr = [77, 42, 11, 0, 99, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

pivot_index = 3, partition

arr = [0, 42, 11, 77, 99, 101]

0 1 2 3 4 5

pivot_index = 0

Example, k = 3

arr = [77, 42, 11, 0, 99, 101]

0 1 2 3 4 5

pivot_index = 3, partition

arr = [0, 42, 11, 77, 99, 101]

0 1 2 3 4 5

pivot_index = 0

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

qs(arr, 3, ?, ?)

Example, k = 3

arr = [77, 42, 11, 0, 99, 101]

0 1 2 3 4 5

pivot_index = 3, partition

arr = [0, 42, 11, 77, 99, 101]

0 1 2 3 4 5

pivot_index = 0

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

qs(arr, 3, 1, 4)

Example, k = 3. If we want to be done?

arr = [0, **42, 11, 77**, 99, 101]

0 1 2 3 4 5

pivot_index = ?

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

qs(arr, 3, 1, 4)

Example, k = 3. If we want to be done?

arr = [0, **42**, **11**, **77**, 99, 101]

0 1 2 3 4 5

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

pivot_index = 1, partition

qs(arr, 3, 1, 4)

arr = [0, **11**, **42**, **77**, 99, 101]

0 1 2 3 4 5

pivot_index = 2

Example, k = 3. If we want to be done?

arr = [0, **42**, **11**, **77**, 99, 101]

0 1 2 3 4 5

pivot_index = 1, partition

arr = [0, **11**, **42**, **77**, 99, 101]

0 1 2 3 4 5

pivot_index = 02

qs(arr, 3, 0, 6)

qs(arr, 3, 0, 4)

qs(arr, 3, 1, 4)

return 42

Partition

Partitioning

- Given an array of n numbers and the index of a **pivot** p.
- Rearrange elements so that:
  - Everything < p is first.
  - Everything = p is next.
  - Everything > p is last.
- Return index of first element ≥ p.

Approach #1

- [77, 42, 11, 99, 0, 101], **pivot** is 11.
- [, , , , ,]
- [, , , , , **77**]
- [, , , , **42**, 77]
- skip for now
- [, , , **99**, 42, 77]
- [**0**, , , 99, 42, 77]
- [0, , **101**, 99, 42, 77]
- [0, **11**, 101, 99, 42, 77]

Approach #1

- [77, 42, 11, 99, 0, 101], **pivot** is 11.
- [, , , , ,]
- [, , , , , **77**]
- [, , , , **42**, 77]
- skip for now
- [, , , **99**, 42, 77]
- [**0**, , , 99, 42, 77]
- [0, , **101**, 99, 42, 77]
- [0, **11**, 101, 99, 42, 77]

Issue: Not in-place.

Partition

- partition takes $\Theta(n)$ time.
- This is **optimal**.
- But we can use memory **more efficiently.**

Approach #2. Motivation

- Similar to selection sort, we'll use two barriers:
- **"Middle"** barrier:
  - Separates things < pivot from things ≥
  - Index of first thing in "right"
- **"End"** barrier:
  - Separates **processed** from *processed*.
  - Index of first "unprocessed" thing.

[< | middle ≥ | end ?]

[< | middle ≥ | end ?]

Example.

arr = [77, 42, 11, 99, 0, 101]

0 1 2 3 4 5

pivot_ix = 1

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| |77, 42, 11, 99, 0, 101]

0 1 2 3 4 5

pivot_ix = 1

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| |77, **42**, 11, 99, 0, 101]

0 1 2 3 4 5

pivot_ix = 1

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| |77, 101, 11, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| | 77, 101, 11, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 77, | **101**, 11, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 77, 101, | **11**, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 77, 101, | **11**, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Swap!

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [| 11, 101, | 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, | 101, 77, 99, | 0, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 101, 77, 99, **0**, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, | 0, 77, 99, | 101, **42**] pivot_ix = 5

0 1 2 3 4 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, 77, 99, 101, **42**]

0 1 2 3 4 5

pivot_ix = 5

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, | 77, 99, 101, | **42**]

0 1 2 3 4 5

pivot_ix = 5

What to do with the pivot?

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, **42**, 99, 101, 77]

0 1 2 3 4 5

pivot_ix = 2

Swap!

Simplification: start by moving pivot to the end of the list

[< | middle ≥ | end ?]

Example.

arr = [11, 0, 42, 99, 101, 77]

0 1 2 3 4 5

pivot_ix = 2

Simplification: start by moving pivot to the end of the list

Loop Invariants

- After **each** iteration:
  - everything in `arr[start:middle_barrier]` is < pivot.
  - everything in `arr[middle_barrier:end_barrier]` is ≥ pivot.
  - everything in `arr[end_barrier:stop]` is “unprocessed”

```python
def in_place_partition(arr, start, stop, pivot_ix):
    def swap(ix_1, ix_2):
        arr[ix_1], arr[ix_2] = arr[ix_2], arr[ix_1]

    pivot = arr[pivot_ix]
    swap(pivot_ix, stop-1)
    middle_barrier = start
    for end_barrier in range(start, stop - 1):
        if arr[end_barrier] < pivot:
            swap(middle_barrier, end_barrier)
            middle_barrier += 1
        # else:
            # do nothing
    swap(middle_barrier, stop-1)
    return middle_barrier
```

Efficiency

- Also takes $\Theta(n)$ time.
- **No auxiliary memory required.**

Time Complexity Analysis

```python
import random
def quickselect(arr, k, start, stop):
    """Finds kth order statistic in numbers[start:stop])"""
    pivot_ix = random.randrange(start, stop)
    pivot_ix = partition(arr, start, stop, pivot_ix)
    pivot_order = pivot_ix + 1
    if pivot_order == k:
        return arr[pivot_ix]
    elif pivot_order < k:
        return quickselect(arr, k, pivot_ix + 1, stop)
    else:
        return quickselect(arr, k, start, pivot_ix)
```

Problem

- We **don't know** the size of the subproblem.
  - Is **random**, can be anywhere from 1 to $n - 1$.
- Difficult to write recurrence relation.

Good and Bad Pivots

- Some pivots are better than others.
- **Good**: splits array into roughly balanced halves.
- **Bad**: splits array into **wildly** unbalanced pieces.

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

[**77**, **42, 11]** if pivot is 77 -> [**42, 11, 77]**

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

[**77**, **42, 11]** if pivot is 77 ->   **[42, 11, 77]**

[**42**, **11**] if pivot is 42 -> [**11**, **42**]

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

[**77, 42, 11, 99**, **101**]

[**77, 42, 11**, **99**] if pivot is 99

[**77**, **42, 11]** if pivot is 77 -> **[42, 11, 77]**

[**42**, **11**] if pivot is 42 -> [**11, 42**]

[**11**]

Worst Case

- Suppose we're searching for k = 1 (minimum).
- **Worst pivot**: the maximum.
- **Worst case**: use max as pivot every time.
- Subproblem size: n – 1.

Worst Case

- Every recursive call is on problem of size $n - 1$.
- $T(n) = T(n - 1) + \Theta(n)$.
  - Solution: $\Theta(n^2)$.
- Intuitively, randomly choosing largest number as pivot every time is **very** unlikely!

$$\frac{1}{n} \times \frac{1}{n-1} \times \frac{1}{n-2} \times \cdots \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{n!}$$

Equally Unlikely

- Pivot falls exactly in the middle, every time.
- Subproblems are of size $n/2$.
- $T(n) = T(n/2) + \Theta(n)$.
  - Solution: $\Theta(n)$.

Typically

- Pivot falls somewhere in the middle.
- Sometimes **good**, sometimes **bad**.
- But **good** pivots reduce problem size by **so much** that they make up for **bad** pivots.

Analogy

- You’re 100 miles away from home.
- You have a button that, if you press it, teleports you **1 mile** closer to home.
- How many times must you press it before you are 1 mile away from home?
- 99

Analogy

- You're 100 miles away from home.
- You have a button that, if you press it, teleports you **half the distance** closer to home.
- How many times must you press it before you are < 1 mile away from home?
- $Log_2 100 \sim 6.64$

Analogy

- You’re 100 miles away from home.
- You have a button that, if you press it, teleports you **half the distance** to home with probability 1/2, does nothing with probability 1/2.
- How many times do you **expect** to press it before you are < 1 mile away from home?
- $2 * \log_2 100 \sim 13.28$

Quickselect

- The same reasoning applies to quickselect.
- If we always get a **good** pivot, time taken is $\Theta(n)$.
- If half the time we get a **bad** pivot, we expect:
  - To make *twice* as many recursive calls.
  - Take *twice* as much time as before.
- But $2\ \Theta(n) = \Theta(n)$.

Quickselect

- **Expected time** complexity: $\Theta(n)$.
- **Worst case**: $\Theta(n^2)$, but *very unlikely*.

Median

- We can find the median in expected linear time with **quickselect**.

Thank you!

Do you have any questions?

CampusWire!