DSC 40B

Theoretical Foundations II

Lecture 7 | Part 1

The Median and Order Statistics

The Median

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

Algorithms

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

Exercise

Using what we know so far, what approach for finding the median has the best **worst-case time complexity**?

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

- The median is an example of an **order statistic**.

Definition

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:

Exercise

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

Special Cases

- **Minimum**: 1st order statistic.
- **Maximum**: nth order statistic.
- **Median**: $\lceil n/2 \rceil$th order statistic[1].
- p**th Percentile**: $\lceil \frac{p}{100} \cdot n \rceil$th order statistic.

[1] What if n is even?

Goal

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

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]
```

Exercise

What are the best case and worst case time complexities of `select_k`?

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...

DSC 40B

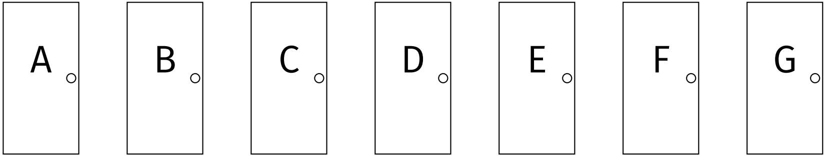
Theoretical Foundations II

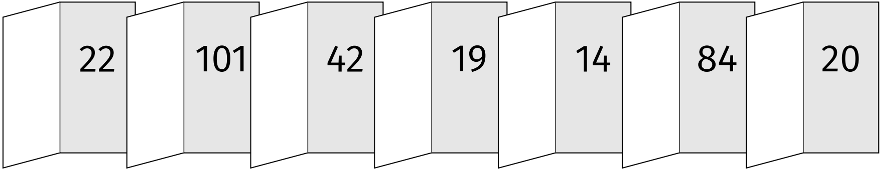
Lecture 7 | Part 2

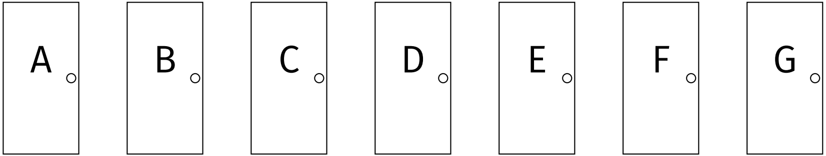
Quickselect

The Goal

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

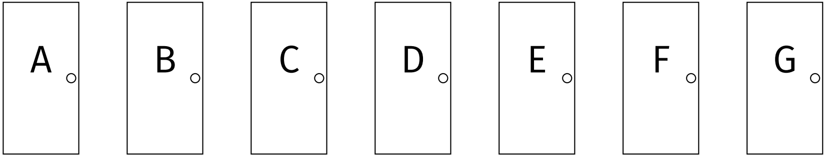

A
B
C
D
E
F
G

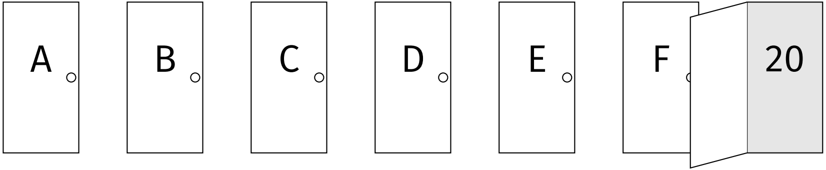

22
101
42
19
14
84
20

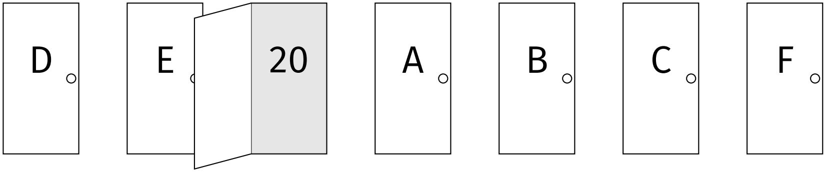

A
B
C
D
E
F
G

Game Show

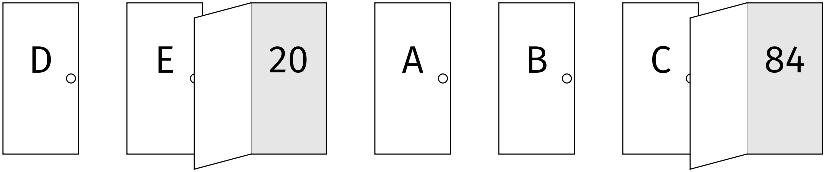
- **Goal**: tell the host the **largest** number.
- **Caution**: with every door opened, your money is reduced.
- **Twist**: After opening a door, the host tells you:
  - which doors are smaller.
  - which doors are larger.
  - they **partition** the doors into higher and lower by moving them.

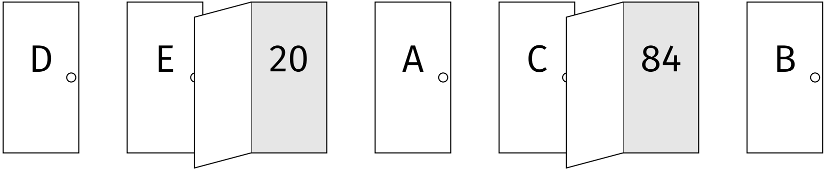

A
B
C
D
E
F
G


A
B
C
D
E
F
20

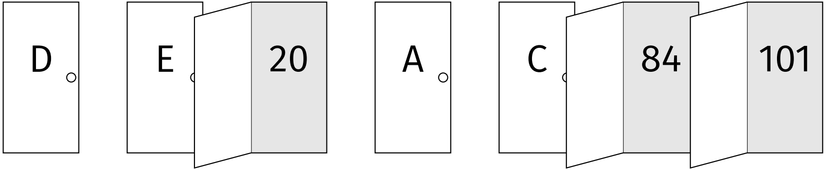


after partitioning


D
E
20
A
B
C
84



after partitioning


D
E
20
A
C
84
101

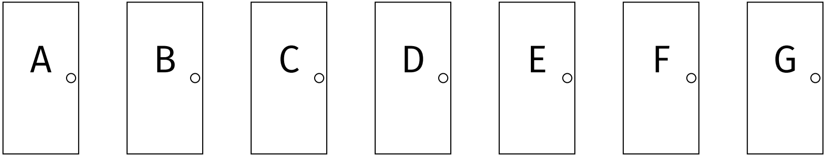
Main Idea

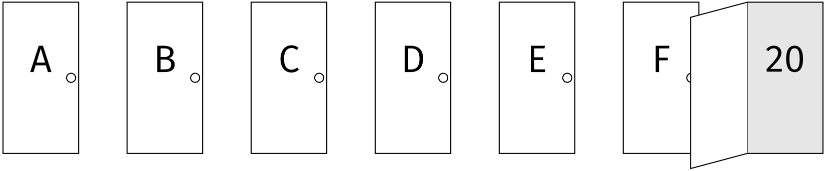
After partitioning, the just-opened door is in the **correct place** in the sorted order (but the other doors may not be).

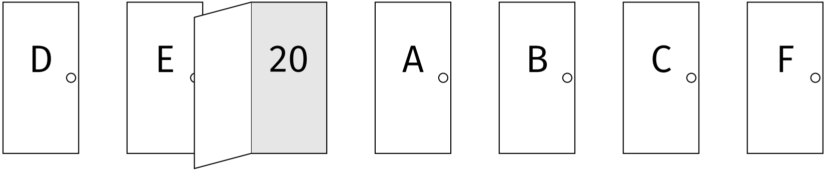
But, every door to the left is smaller (≤), every door to the right is larger (≥).

In general...

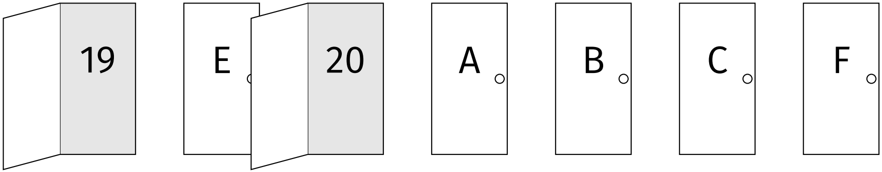
- Let's generalize strategy for kth order statistic.
- Example: $k = 2$.

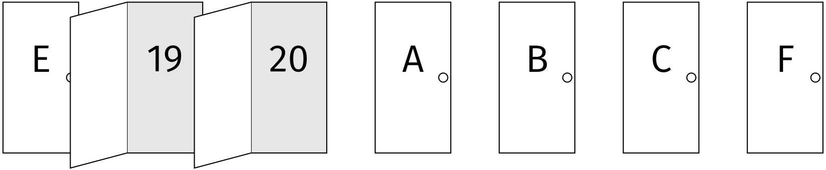

A
B
C
D
E
F
G


A
B
C
D
E
F
20



after partitioning


19
E
20
A
B
C
F



after partitioning

Strategy

- Open 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.

In Code

```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

```
arr = [77, 42, 11, 99, 0, 101]    k = 3
```

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 3

Partition

Paritioning

- 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 $\geq p$.

```python
def partition(arr, start, stop, pivot_ix):
    """Partition arr[start:stop] around pivot."""
    left = []
    pivot_count = 0
    right = []
    pivot = arr[pivot_ix]
    for ix in range(start, stop):
        if arr[ix] < pivot:
            left.append(arr[ix])
        elif arr[ix] == pivot:
            pivot_count += 1
        else:
            right.append(arr[ix])
    ix = start
    for x in left:
        arr[ix] = x
        ix += 1
    for i in range(pivot_count):
        arr[ix] = pivot
        ix += 1
    for x in right:
        arr[ix] = x
        ix += 1
    return start + len(left)
```

Partition

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

Motivation

- Similar to selection sort, we'll use **two** barriers:

- "Middle" barrier:
  - Separates things < pivot from things ≥
  - Points to first thing in "right"

- "End" barrier:
  - Separates processed from processed.
  - Points to first unprocessed thing.

Example

Simplification: start by moving pivot to end.

```
arr = [77, 42, 11, 99, 0, 101]    pivot = 1
```

```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.

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 4

Time Complexity Analysis

Time Complexity

- What is time complexity of `quickselect`?

```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:
```

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?

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’re 1 mile away from home?

Analogy

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

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 must you press it before you’re 1 mile away from home?

Analysis

- We'll call a pivot **good** if it falls in $[\frac{n}{4}, \frac{3n}{4}]$.
  - Probability: 1/2
  - Max problem size: $3n/4$.
- We'll call a pivot **bad** if it falls outside $[\frac{n}{4}, \frac{3n}{4}]$.
  - Probability: 1/2
  - Max problem size: $n - 1$.

Observation

$T(n)$ = time to get from n to base case

Observation

$$
\begin{aligned}
T(n) = {} & \text{time to get from } n \text{ to } \frac{3}{4}n \\
& + \text{time to get from } \frac{3}{4}n \text{ to } \left(\frac{3}{4}\right)^2 n \\
& + \text{time to get from } \left(\frac{3}{4}\right)^2 n \text{ to } \left(\frac{3}{4}\right)^3 n \\
& + \ldots
\end{aligned}
$$

Observation

Expected $T(n)$ = expected time to get from n to $\frac{3}{4}n$

+ expected time to get from $\frac{3}{4}n$ to $\left(\frac{3}{4}\right)^2 n$

+ expected time to get from $\left(\frac{3}{4}\right)^2 n$ to $\left(\frac{3}{4}\right)^3 n$

+ ...

Related

- What is the expected number of coin flips necessary in order to see “heads”?

Related

- What is the expected number of coin flips necessary in order to see “heads”?
- Answer: 2

Implication

- Expected number of calls necessary to go from n to $3n/4$ is two.

- First call does cn work, second does $c \times (3/4)n$, third does $c \times (3/4)^2 n$, ...

Observation

$$
\begin{aligned}
\text{Expected } T(n) &= \text{expected time to get from } n \text{ to } \frac{3}{4}n \\
&+ \text{expected time to get from } \frac{3}{4}n \text{ to } \left(\frac{3}{4}\right)^2 n \\
&+ \text{expected time to get from } \left(\frac{3}{4}\right)^2 n \text{ to } \left(\frac{3}{4}\right)^3 n \\
&+ \ldots
\end{aligned}
$$

Total Expected Time

$$2cn + 2\left(\frac{3}{4}\right)cn + 2\left(\frac{3}{4}\right)^2 cn + \ldots = 2cn \cdot \sum_{p=0}^{\infty} \left(\frac{3}{4}\right)^p$$

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**.

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 5

Quicksort

Last Time

- We saw `mergesort`.
- **Divide**: split list directly down the middle
- **Conquer**: sort each half
- **Combine**: merge sorted halves together

`merge` does all the work

- In `mergesort`, we are lazy when we divide.
- So we have to work to combine.

`[4,1,3,2]` → `[4,1]`,`[3,2]` → `[4,3]`,`[2,3]` → `[1,2,3,4]`

What if?

- Suppose we divide so that everything in `left` is smaller than everything in `right`:
- After sorting, no need for `merge`.
- `[5,1,3,8,6,2]` → `[1,3,2],[5,8,6]`

What if?

- Suppose we divide so that everything in `left` is smaller than everything in `right`:
- After sorting, no need for `merge`.
- `[5,1,3,8,6,2]` → `[1,3,2]`,`[5,8,6]`
- This is what `partition` does!

Quicksort

```python
def quicksort(arr, start, stop):
    """Sort arr[start:stop] in-place."""
    if stop - start > 1:
        pivot_ix = random.randrange(start, stop)
        pivot_ix = partition(arr, start, stop, pivot_ix)
        quicksort(arr, start, pivot_ix)
        quicksort(arr, pivot_ix+1, stop)
```

Time Complexity

- Average case: $\Theta(n \log n)$
- Worst case: $\Theta(n^2)$.
- Like with `quickselect`, worst case is **very rare**.

Mergesort vs Quicksort

- Mergesort has better worst case complexity.
- But in practice, Quicksort is often faster.
- Takes less memory, too.

Memory Requirements

- `merges` requires output array, $\Theta(n)$ additional space.
- `partition` works in-place, requires no additional space[2]
- Example: sorting 3 GB of data with 4 GB of RAM.

[2] Call stack for `quicksort` requires $\Theta(\log n)$ additional space.