# Selection

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

# Order Statistics and Selection

- Given a set of $n$ items, the $i^{\text{th}}$ **order statistic** of it the $i^{\text{th}}$ smallest element of it.
- Minimum, maximum, median, …

- **The Selection Problem:** given a set $A$ of $n$ *distinct* numbers and an integer $i$, find the $i^{\text{th}}$ order statistic of $A$.

# Find Min/Max

- So easy, sequential scan and keep min/max till now…

```
FindMin(A):
min = A[1]
for (i=2 to A.length)
  if (A[i]<min)
    min = A[i]
return min
```

- Make $n-1$ comparisons, but is this the best we can do?
- Yes! Otherwise at least two elements could be the minimum.
  - Initially each element could be the minimum.
  - An adversary answers queries like “compare $x$ with $y$”.
  - Each comparison eliminates at most one element.

# What if we want min *and* max?

- Go through the list twice, one for min and another for max.
- Can we do better? Surprisingly, yes!
  - Group items into pairs. (The first item becomes a "pair" if $n$ is odd.)
  - For each of $\lceil n/2 \rceil$ pairs, find "local" min and max. **$\lfloor n/2 \rfloor$ comparis**
  - Among $\lceil n/2 \rceil$ "local" min, find global min; similarly find global max. **$\le 2 \cdot \lfloor n/2 \rfloor$ comparis**
  - Total # of comparisons is at most $3 \cdot \lfloor n/2 \rfloor$.
- Is $3 \cdot \lfloor n/2 \rfloor$ the best we can do? Remarkably, yes!
  - An item has $+$ mark if it can be max, and has $-$ mark if it can be min.
  - Initially each item has both $+$ and $-$.
  - An adversary answers queries like "compare $x$ with $y$".
  - The adversary can find input such that:
    at most $\lfloor n/2 \rfloor$ comparisons each removes two marks;
    every other comparison removes at most one mark.
  - In total need to remove $2n - 2$ marks.

# General Selection Problem

- Find $i^{\text{th}}$ smallest element (i.e., $i^{\text{th}}$ order statistic)?
- Err… Sort them then return the $i^{\text{th}}$ entry?
- Sure but this takes $\Omega(n \log n)$ time…
- Can we be faster? YES!

- What if $i = q$?
  - $A[q]$ is what we need.
- What if $i < q$?
  - Find $i^{\text{th}}$ order statistic in $A[1 \ldots (q-1)]$.
- What if $i > q$?
  - Find $(i-q)^{\text{th}}$ order statistic in $A[(q+1) \ldots n]$.

**<u>RndQuickSort(A):</u>**

```
if (A.size>1)
  q = RandomPartition(A)
  RndQuickSort(A[1...(q-1)])
  RndQuickSort(A[(q+1)...n])
```

Notice $A[1 \ldots (q-1)]$ contains the smallest $q-1$ elements in $A$.

**This is Divide-and-Conqu**

# Randomized Selection

**A Divide-and-Conquer Alg.**

```
RndSelect(A, i):
if (A.size=1)
  return A[1]
else
  q = RandomPartition(A)
  if (i==q)
    return A[q]
  elseif (i<q)
    return RndSelect(A[1…(q-1)],i)
  else
    return RndSelect(A[(q+1)…A.size],i-q)
```

**Average Runti**

Best-case runtime? $\Theta(n)$

Choose the answer as the pivot in the first call (unlikely to happen).

Worst-case runtime? $\geq cn + c(n-1) + \cdots + c(2) = \Theta(n^2)$

Partition reduces array size by one each time (unlikely to happen).

## A Divide-and-Conque

Best-case: chooses the answer as the pivot right away.

Worst-case: partition process only reduces problem size by one.

What's likely to happen: partition process reduces problem size by a ***constant*** factor.

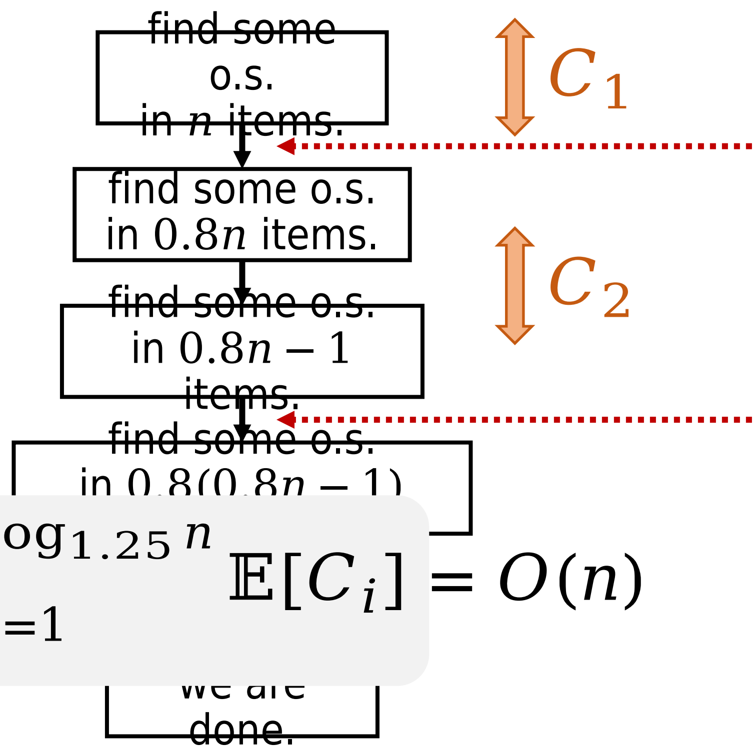
**RndSelect(A, i):**

```
if (A.size=1)
  return A[1]
else
  q = RandomPartition(A)
  if (i==q)
    return A[q]
  elseif (i<q)
    return RndSelect(A[1…(q-1)],i)
  else
    return RndSelect(A[(q+1)…A.size],i-q)
```

Call a partition ***good*** if it reduces problem size to at most 0.8*input_size.

Let r.v. $C_i$ be the cost since the last good partition to the $i^{\text{th}}$ good partition.

At most $\log_{1.25} n$ good partitions can occur.




$$\mathbb{E}[C_i] \le \Theta(1) \cdot 0.8^{i-1} n$$

$$\mathbb{E}[T(n)] \le \mathbb{E}\left[\sum_{i=1}^{\log_{1.25} n} C_i\right] = \sum_{i=1}^{\log_{1.25} n} \mathbb{E}[C_i] = O(n)$$

## A Divide-and-Conque

```
RndSelect(A, i):
if (A.size=1)
  return A[1]
else
  q = RandomPartition(A)
  if (i==q)
    return A[q]
  elseif (i<q)
    return RndSelect(A[1...(q-1)],i)
  else
    return RndSelect(A[(q+1)...A.size],i-q)
```

Best-case: chooses the answer as the pivot right away.

Worst-case: partition process only reduces problem size by one.

What's likely to happen: partition process reduces problem size by a *constant* factor.

Cost on an input of size $n$
$\leq$ (cost on an input of size $0.8n$) + (cost to reduce input to size $\leq 0.8n$)

$$\mathbb{E}[T(n)] \leq \mathbb{E}[T(0.8n)] + \mathbb{E}[cost\ to\ reduce\ size\ n\ input\ t$$

$$\mathbb{E}[T(n)] \leq \mathbb{E}[T(0.8n)] + O(n)$$

$$\mathbb{E}[T(n)] = O(n)$$

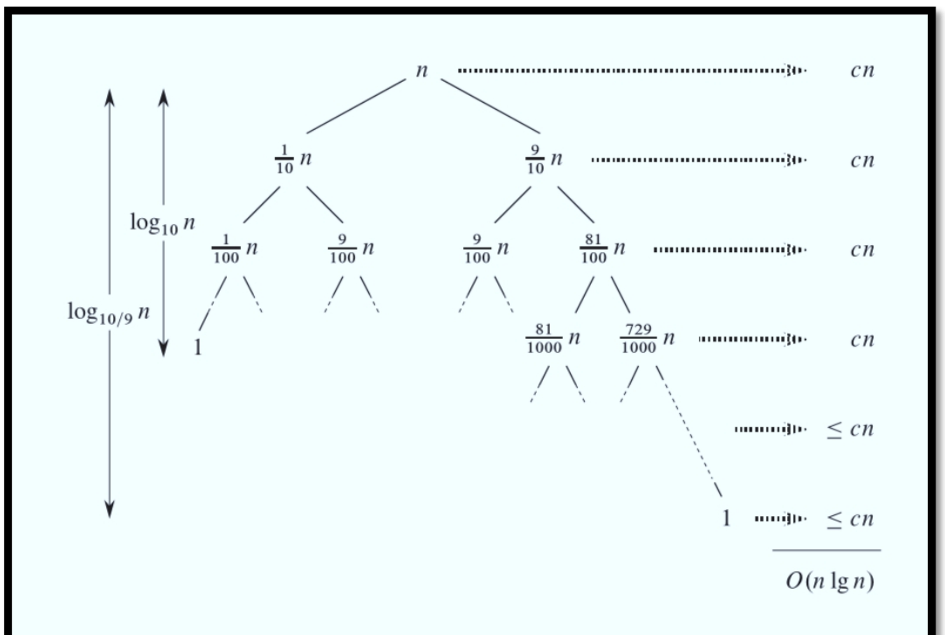
# RndQuickSort vs RndSelect

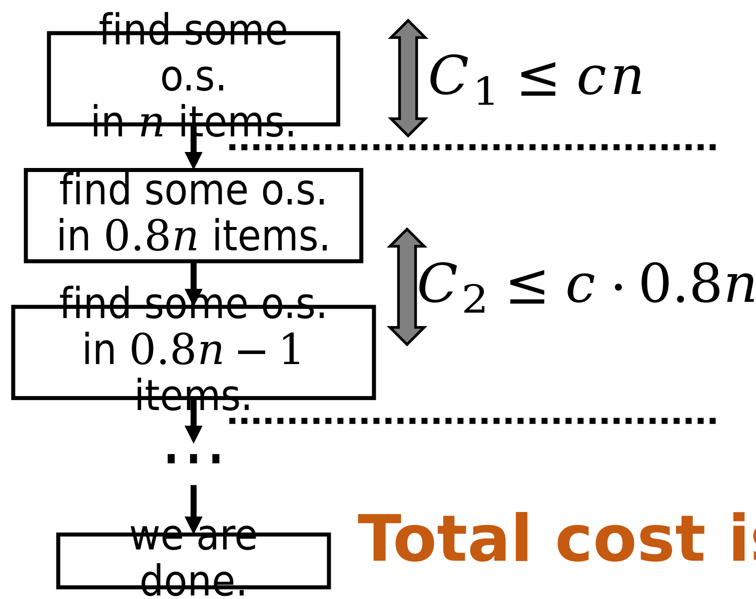
**RndSelect(A, i):**

```
if (A.size=1)
  return A[1]
else
  q = RandomPartition(A)
  if (i==q)
    return A[q]
  elseif (i<q)
    return RndSelect(A[1...(q-1)],i)
  else
    return RndSelect(A[(q+1)...A.size],i-q)
```

**RndQuickSort(A):**

```
if (A.size>1)
  q = RandomPartition(A)
  RndQuickSort(A[1...(q-1)])
  RndQuickSort(A[(q+1)...n])
```




**Total cost is *O*(**

# We are not done with selection…

- Can we guarantee **worst-case** runtime of $O(n)$?
- The reason that RndSelect could be slow is that RandomPartition might return an *unbalanced* partition.
- Needs a partition procedure that guarantees to be *balanced*. (without using too much time; $O(n)$ time to be specific)

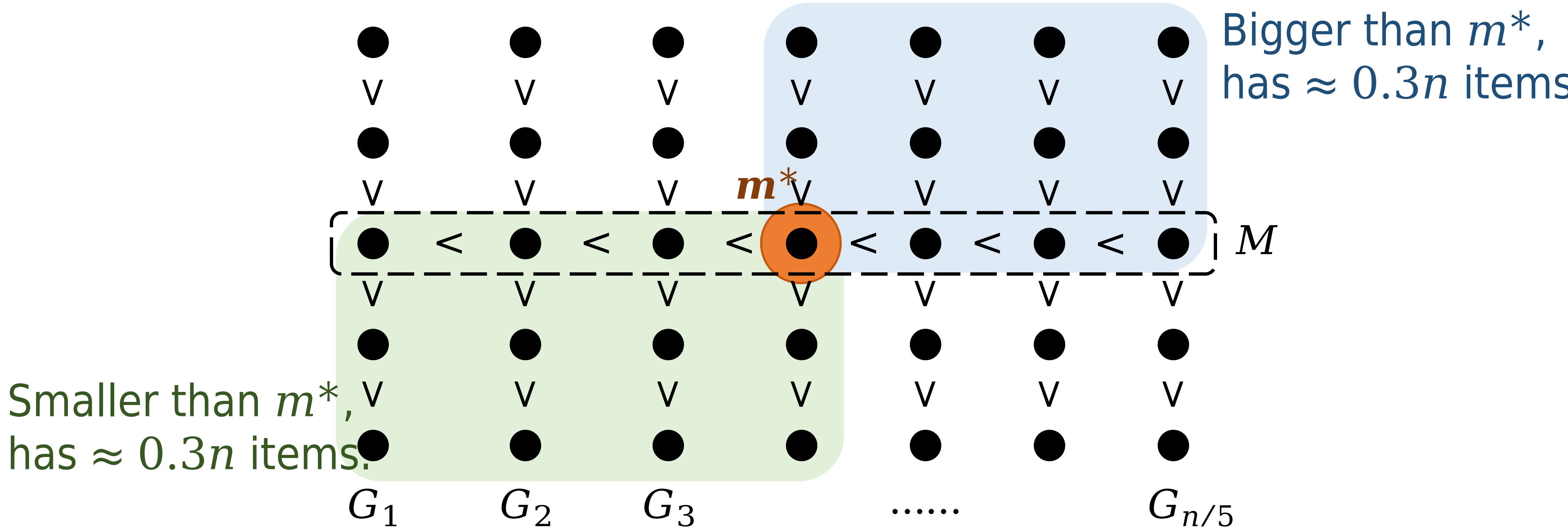
**RndSelect(A, i):**

```
if (A.size=1)
  return A[1]
else
  q = RandomPartition(A)
  if (i==q)
    return A[q]
  elseif (i<q)
    return RndSelect(A[1…(q-1)],i)
  else
    return RndSelect(A[(q+1)…A.size],i-q)
```

# Median of medians

- Divide elements into $n/5$ groups, each containing 5 elements, call these groups $G_1, G_2, \cdots, G_{n/5}$.
- Find the medians of these $n/5$ groups, let $M$ be this set of medians.
- Find the median of $M$, call it $m^*$.

Partition using $m^*$ as pivot is good: the smaller split has $\geq 0.3n$ items.



# Finding median of medians

- Divide elements into $n/5$ groups, each containing $5$ elements, call these groups $G_1, G_2, \ldots, G_{n/5}$.
  Trivial, $\Theta(n)$ time.
- Find the medians of these $n/5$ groups, let $M$ be this set of medians.
  Sort each group, then find the medians.
  Cost is $(n/5) \cdot \Theta(1) = \Theta(n)$.
- Find the median of $M$, call it $m^*$.
  Use QuickSelect, recursively!

**QuickSelect(A, i):**

```
if (A.size=1)
  return A[1]
else
  m = MedianOfMedians(A)
  q = PartitionWithPivot(A,m)
  if (i==q)
    return A[q]
  elseif (i<q)
    return QuickSelect(A[1..
  else
    return QuickSelect(A[(q+
```

How much time?
(Can only afford $O(n)$.)

**MedianOfMedians(A):**

```
<G_1,G_2,...,G_n/5> = CreateGroups(A)
for (i=1 to n/5)
  Sort(G_i)
M = GetMediansFromSortedGroups(G_1,G_2,...,G_n/5)
return QuickSelect(M,(n/5)/2)
```

# Time complexity?

**MedianOfMedians(A):**

```
<G_1,G_2,...,G_n/5> = CreateGroups(A)
for (i=1 to n/5)
  Sort(G_i)
M = GetMediansFromSortedGroups(G_1,G_2,...,G_n/5)
return QuickSelect(M,(n/5)/2)
```

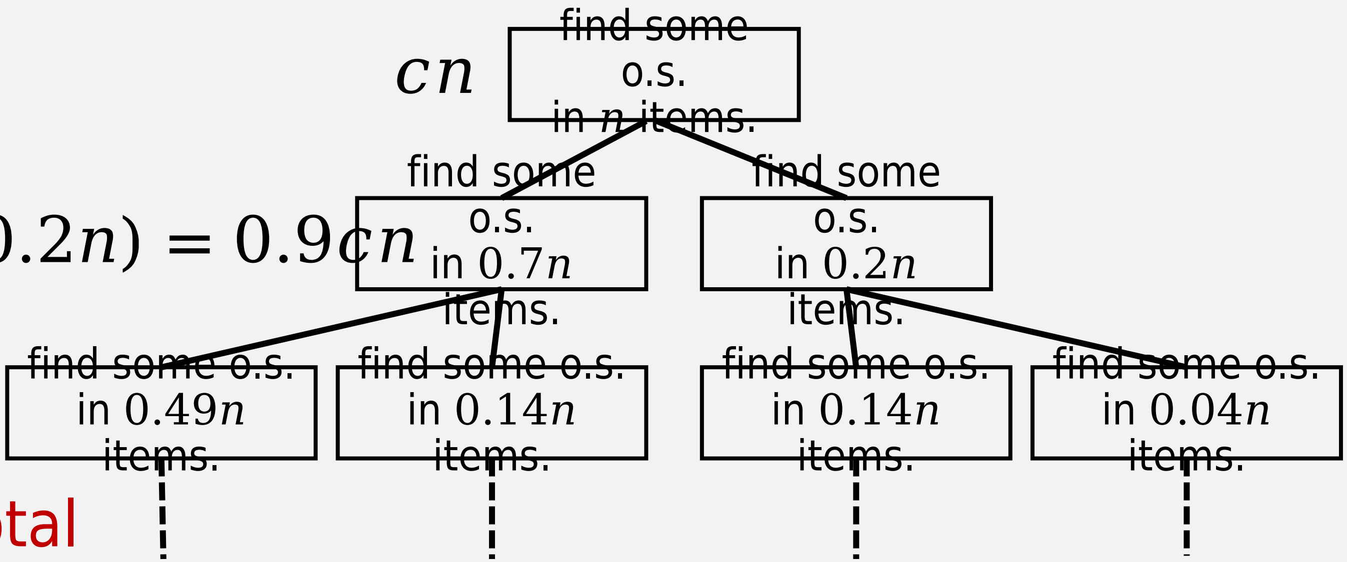
**QuickSelect(A, i):**

```
if (A.size=1)
  return A[1]
else
  m = MedianOfMedians(A)
  q = PartitionWithPivot(A,m)
  if (i==q)
    return A[q]
  elseif (i<q)
    return QuickSelect(A[1...(q-1)],i)
  else
    return QuickSelect(A[(q+1)...A.size],i-q)
```

$cn$

$c(0.7n + 0.2n) = 0.9cn$

$0.81cn$




$O(n)$ in total

## Time complexity?

**MedianOfMedians(A):**

```
<G_1,G_2,...,G_{n/5}> = CreateGroups(A)
for (i=1 to n/5)
  Sort(G_i)
M = GetMediansFromSortedGroups(G_1,G_2,...,G_{n/5})
return QuickSelect(M,(n/5)/2)
```

**QuickSelect(A, i):**

```
if (A.size=1)
  return A[1]
else
  m = MedianOfMedians(A)
  q = PartitionWithPivot(A,m)
  if (i==q)
    return A[q]
  elseif (i<q)
    return QuickSelect(A[1...(q-1)],i)
  else
    return QuickSelect(A[(q+1)...A.size],i-q)
```

$$T(n) \leq T(0.7n) + T(0.2n) + O(n)$$

$$T(n) = O(n)$$

You can verify this by the **substitution method**.
(I.e., assume $T(n) \leq cn$ and then verify.)

# Complexity of general selection

- QuickSelect uses $O(n)$ time/comparisons.
- Solving general selection needs at least $n-1$ comparisons.
  - Since finding min/max needs at least $n-1$ comparisons.

- So the lower and upper bounds match *asymptotically*.
- But if we care about constants, needs (much) more efforts.

# Reading

- [CLRS] Ch.9