Indian Institute of Information Technology Allahabad

Data Structures

Quick Sort (Randomized Algorithm)

Dr. Shiv Ram Dubey
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

What is a Randomized Algorithm?

- An algorithm that incorporates randomness as part of its operation.
- Basically, we'll make random choices during the algorithm:
  - Sometimes, we'll just hope that our algorithm is fast!
  - Other times, we'll just hope that it works!
- Let's formalize this...

Las Vegas vs. Monte Carlo

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

Las Vegas vs. Monte Carlo

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

MONTE CARLO ALGORITHMS

Correctness is a random variable.
(i.e. there's a chance the output is wrong)

But the runtime is guaranteed!

Las Vegas vs. Monte Carlo

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

↑

We'll focus on these algorithms today
(BogoSort, QuickSort)

MONTE CARLO ALGORITHMS

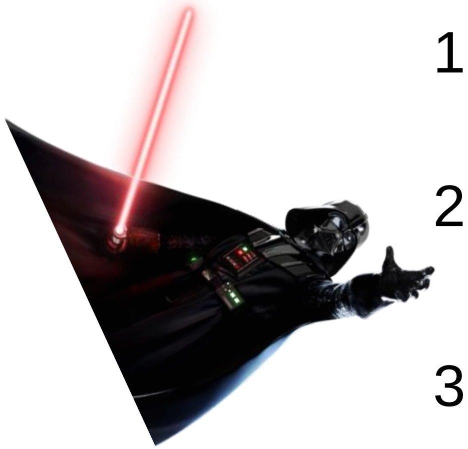
Correctness is a random variable.
(i.e. there's a chance the output is wrong)

But the runtime is guaranteed!

↑

You'll see some examples of these later in the DAA course!

How do we measure the runtime of a randomized algorithm?



Scenario 1

1. You publish your algorithm.
2. Bad guy picks the input.
3. You run your randomized algorithm.

Scenario 2

1. You publish your algorithm.
2. Bad guy picks the input.
3. Bad guy chooses the randomness (fixes the dice) and runs your algorithm.

- In **Scenario 1**, the running time is a **random variable.**
  - It makes sense to talk about **expected running time**.
- In **Scenario 2**, the running time is **not random**.
  - We call this the **worst-case running time** of the randomized algorithm.

How do we measure the runtime of a randomized algorithm?

in both cases, we are still thinking about the *WORST-CASE INPUT*

Scenario 1 **Scenario 2**

Don't get confused!!!

Even with randomized algorithms, we are still considering the *WORST CASE INPUT*, regardless of whether we're computing expected or worst-case runtime.

Expected runtime ***IS NOT*** runtime when given an expected input! We are taking the expectation over the random choices that our algorithm would make, ***NOT*** an expectation over the distribution of possible inputs.

- In **Scenario 2**, the running time is **not random**.
  - We call this the **worst-case running time** of the randomized algorithm.

Quick Probability Exercise

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with probability 99/100.

a. What is the expected value $\mathbb{E}[X]$?

Quick Probability Exercise

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with probability 99/100.

a. What is the expected value $\mathbb{E}[X]$? $\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$

Quick Probability Exercise

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with probability 99/100.

a. What is the expected value $\mathbb{E}[X]$? $\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

Quick Probability Exercise

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with probability 99/100.

a. What is the expected value $\mathbb{E}[X]$? $\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

By linearity of expectation: $\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{n}{100}$

Quick Probability Exercise

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with probability 99/100.

a. What is the expected value $\mathbb{E}[X]$? $\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

By linearity of expectation: $\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{n}{100}$

c. Suppose you draw independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, and you stop when you see the first "**1**". Let N be the last index that you draw. What is the expected value of N?

Quick Probability Exercise

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with probability 99/100.

a. What is the expected value $\mathbb{E}[X]$? $\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}\left[\sum_{i=1}^{n} X_i\right]$?

By linearity of expectation: $\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{n}{100}$

c. Suppose you draw independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, and you stop when you see the first "**1**". Let N be the last index that you draw. What is the expected value of N?

N is a *geometric random variable*. We can use the formula: $\mathbb{E}[N] = \frac{1}{p} = \frac{1}{1/100} = 100$

Geometric Random Variable

- If **N** represents "number of trials/attempts", and **p** is the probability of "success" on each trial, then:

$$\mathbb{E}[N] = \frac{1}{p}$$

$$\begin{aligned}\mathbb{E}[N] &= 1(p) + (1 + \mathbb{E}[N])(1 - p) \\ &= p + (1 - p) + (1 - p)\mathbb{E}[N] \\ &= 1 + (1 - p)\mathbb{E}[N]\end{aligned}$$

$$\begin{aligned}\mathbb{E}[N](1 - (1 - p)) &= 1 \\ \mathbb{E}[N](p) &= 1 \\ \mathbb{E}[N] &= \frac{1}{p}\end{aligned}$$

Bogo Sort

A bit silly, but a great pedagogical tool!

Bogo Sort

```
BOGOSORT(A):
 while True:
    A.shuffle()
    sorted = True
    for i in [0,...,n-2]:
       if A[i] > A[i+1]:
             sorted = False
    if sorted:
       return A
```

This randomly permutes A (assume it takes O(n) time)

Bogo Sort: Expected Runtime

```
BOGOSORT(A):
 while True:
   A.shuffle()
   sorted = True
   for i in [0,...,n-2]:
     if A[i] > A[i+1]:
          sorted = False
   if sorted:
     return A
```

What is the expected number of iterations?

Bogo Sort: Expected Runtime

```
BOGOSORT(A):
 while True:
   A.shuffle()
   sorted = True
   for i in [0,...,n-2]:
     if A[i] > A[i+1]:
          sorted = False
   if sorted:
     return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

Bogo Sort: Expected Runtime

```
BOGOSORT(A):
 while True:
   A.shuffle()
   sorted = True
   for i in [0,...,n-2]:
      if A[i] > A[i+1]:
            sorted = False
   if sorted:
      return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

Probability that $\mathbf{X_i = 1}$ (A is sorted) = **1/n!**

since there are n! possible orderings of A and only one is sorted (assume A has distinct elements) ⇒ $E[X_i] = 1/n!$

Bogo Sort: Expected Runtime

```
BOGOSORT(A):
 while True:
   A.shuffle()
   sorted = True
   for i in [0,...,n-2]:
     if A[i] > A[i+1]:
          sorted = False
   if sorted:
     return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

Probability that $\mathbf{X_i = 1}$ (A is sorted) = **1/n!**

since there are n! possible orderings of A and only one is sorted (assume A has distinct elements) ⇒ $E[X_i] = 1/n!$

E[# of iterations/trials] = 1/(prob. of success on each trial)

= 1/(1/n!) = **n!**

Bogo Sort: Expected Runtime

```
BOGOSORT(A):
 while True:
    A.shuffle()
    sorted = True
    for i in [0,...,n-2]:
       if A[i] > A[i+1]:
             sorted = False
    if sorted:
       return A
```

E[runtime on a list of length n]

= **E**[(# of iterations) * (time per iteration)]

= (time per iteration) * **E**[# of iterations]

= O(n) * **E**[# of iterations]

= O(n) * (n!)

= O(n * n!)

= ***REALLY REALLY BIG***

Bogo Sort: Worst-Case Runtime

```
BOGOSORT(A):
 while True:
    A.shuffle()
    sorted = True
    for i in [0,...,n-2]:
       if A[i] > A[i+1]:
             sorted = False
    if sorted:
       return A
```

Bogo Sort: Worst-Case Runtime

```
BOGOSORT(A):
 while True:
   A.shuffle()
   sorted = True
   for i in [0,...,n-2]:
      if A[i] > A[i+1]:
            sorted = False
   if sorted:
      return A
```

Worst-case runtime =

∞

This is as if the "bad guy" chooses all the randomness in the algorithm, so each shuffle could be unlucky… forever...

What have we learned?

EXPECTED RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. You get to roll the dice (leave it up to randomness)

WORST-CASE RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. Bad guy “rolls” the dice (will choose the randomness in the worst way possible)

What have we learned?

EXPECTED RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. You get to roll the dice (leave it up to randomness)

WORST-CASE RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. Bad guy “rolls” the dice (will choose the randomness in the worst way possible)

Don’t use BogoSort.

Quick Sort

A much better randomized algorithm

Quick Sort Overview

EXPECTED RUNNING TIME	WORST-CASE RUNNING TIME
$O(n \log n)$	$O(n^2)$

Quick Sort Overview

EXPECTED RUNNING TIME

$O(n \log n)$

WORST-CASE RUNNING TIME

$O(n^2)$

In practice, it works great! It's competitive with MergeSort (& often better in some contexts!), and it runs *in place* (no need for lots of additional memory)

Quick Sort: The Idea

Let's use DIVIDE-and-CONQUER again!

Select a pivot *at random*

Partition around it

Recursively sort L and R!

Quick Sort: The Idea

Select a pivot

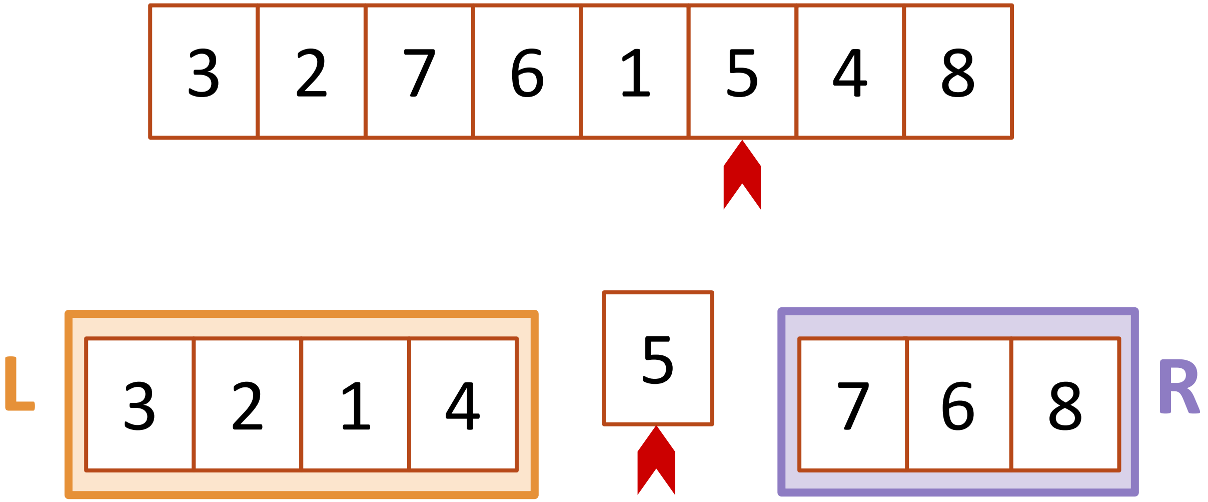
3	2	7	6	1	5	4	8

Pick this pivot
uniformly at random!

Quick Sort: The Idea

Select a pivot

Partition around it



Pick this pivot uniformly at random!

Partition around pivot: **L** has elements less than pivot, and **R** has elements greater than pivot.

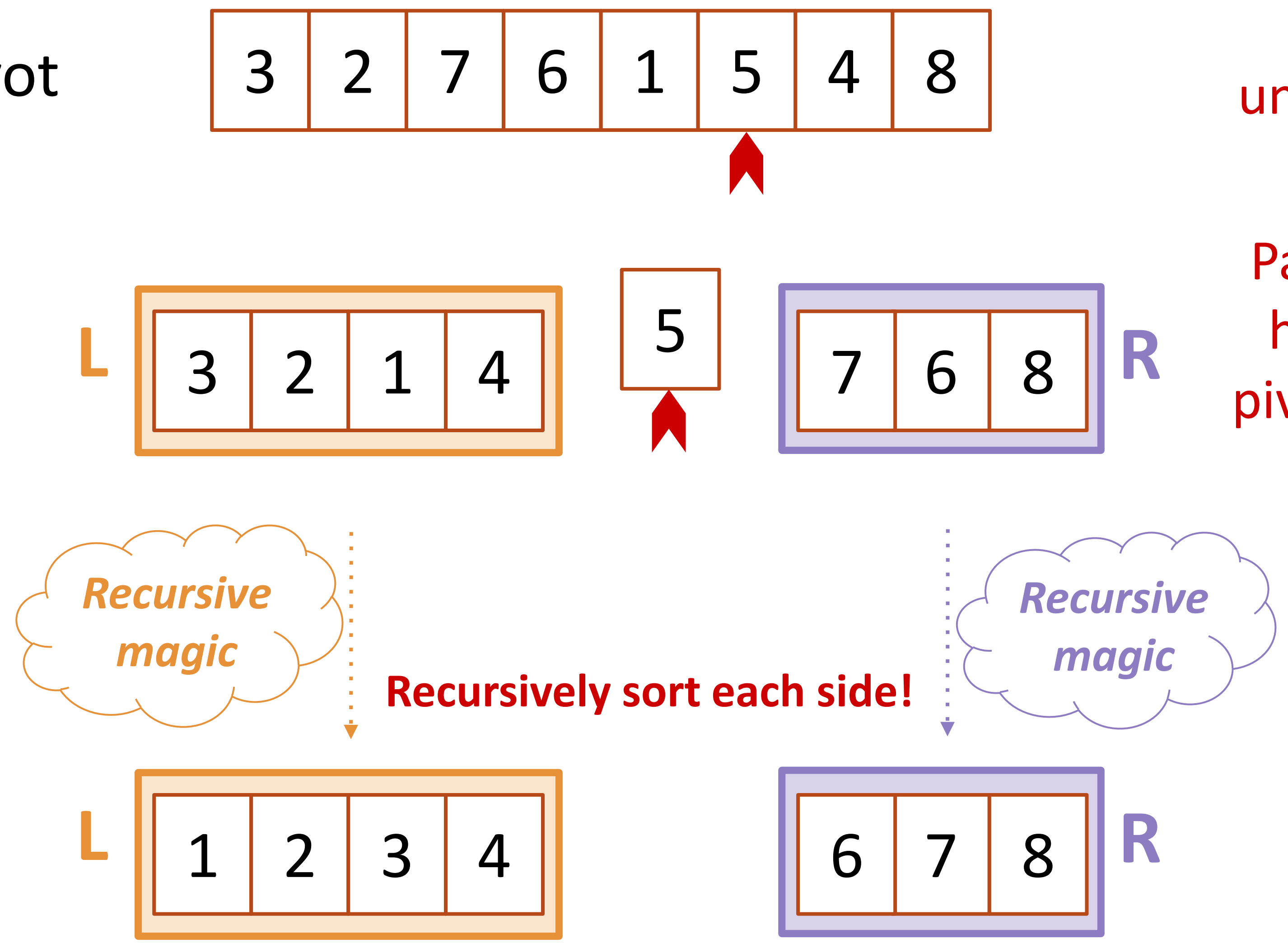
Quick Sort: The Idea

Select a pivot

Pick this pivot uniformly at random!

Partition around it

Partition around pivot: **L** has elements less than pivot, and **R** has elements greater than pivot.



Recurse!

Quick Sort: Pseudo-Code

```
QUICKSORT(A):
  if len(A) <= 1:
      return
  pivot = random.choice(A)
  PARTITION A into:
      L (less than pivot) and
      R (greater than pivot)
  Replace A with [L, pivot, R]
  QUICKSORT(L)
  QUICKSORT(R)
```

Quick Sort: Recurrence Relation

```
QUICKSORT(A):
  if len(A) <= 1:
      return
  pivot = random.choice(A)
  PARTITION A into:
      L (less than pivot) and
      R (greater than pivot)
  Replace A with [L, pivot, R]
  QUICKSORT(L)
  QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

Quick Sort: Ideal Runtime?

```
QUICKSORT(A):
  if len(A) <= 1:
      return
  pivot = random.choice(A)
  PARTITION A into:
      L (less than pivot) and
      R (greater than pivot)
  Replace A with [L, pivot, R]
  QUICKSORT(L)
  QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

Ideal Runtime?

Quick Sort: Ideal Runtime?

```
QUICKSORT(A):
  if len(A) <= 1:
      return
  pivot = random.choice(A)
  PARTITION A into:
      L (less than pivot) and
      R (greater than pivot)
  Replace A with [L, pivot, R]
  QUICKSORT(L)
  QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

In an ideal world, the pivot would split the array exactly in half, and we'd get:

$$T(n) = T(n/2) + T(n/2) + O(n)$$

Quick Sort: Ideal Runtime?

QUICKSORT(A):
  if len(A) <= 1:
    return
  pivot = rando
  PARTITION
    L (less tha
    R (greate
  Replace A w
  QUICKSORT(L)
  QUICKSORT(R)

In an ideal world:

$T(n) = 2 \cdot T(n/2) + O(n)$

$T(n) = $ **$O(n \log n)$**

Recurrence Relation for QUICKSORT

+ T(|R|) + O(n)

(1) = O(1)

, the pivot would
xactly in half, and
we'd get:

$T(n) = T(n/2) + T(n/2) + O(n)$

Quick Sort: Worst-Case Runtime?

```
QUICKSORT(A):
  if len(A) <= 1:
      return
  pivot = random.choice(A)
  PARTITION A into:
      L (less than pivot) and
      R (greater than pivot)
  Replace A with [L, pivot, R]
  QUICKSORT(L)
  QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

Worst-Case Runtime?

Quick Sort: Worst-Case Runtime?

```
QUICKSORT(A):
  if len(A) <= 1:
      return
  pivot = random.choice(A)
  PARTITION A into:
      L (less than pivot) and
      R (greater than pivot)
  Replace A with [L, pivot, R]
  QUICKSORT(L)
  QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

With the unluckiest randomness, the pivot would be either min(A) or max(A):

$$T(n) = T(0) + T(n-1) + O(n)$$

Quick Sort: Worst-Case Runtime?

QUICKSORT(A):
if len(A) <= 1:
return
pivot = ra
PARTITIO
L (less
R (gre
Replace A
QUICKSORT(L)
QUICKSORT(R)

Recurrence Relation for QUICKSORT

T(|R|) + O(n)
) = O(1)

With the worst "randomness"

$T(n) = T(n-1) + O(n)$

$T(n) = O(n^2)$

t randomness,
either min(A)
or max(A):

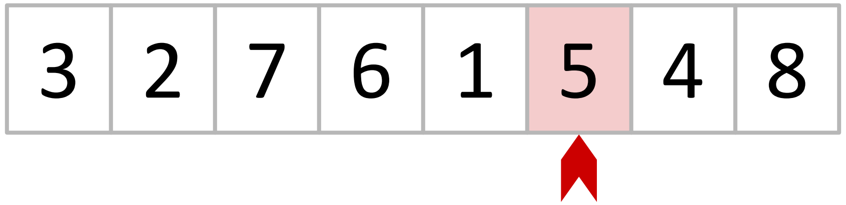
$T(n) = T(0) + T(n-1) + O(n)$

Quick Sort: Expected Runtime

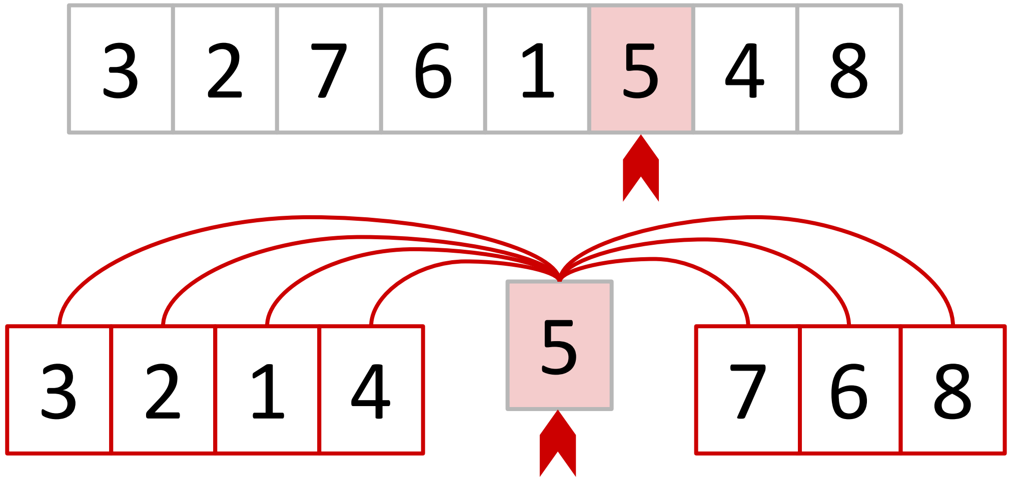
$$O(n \log n)$$

- In order to prove this expected runtime:
  - Lets compute
    - How many times are any two items compared, in expectation?

How Many Comparisons?

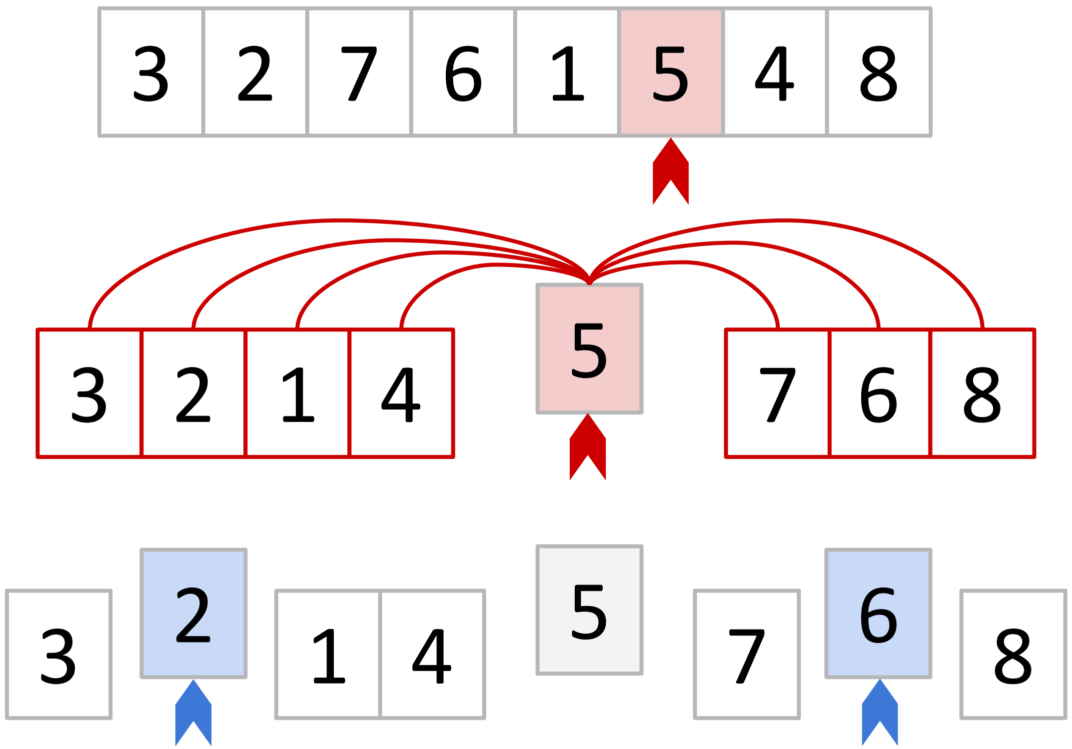


How Many Comparisons?



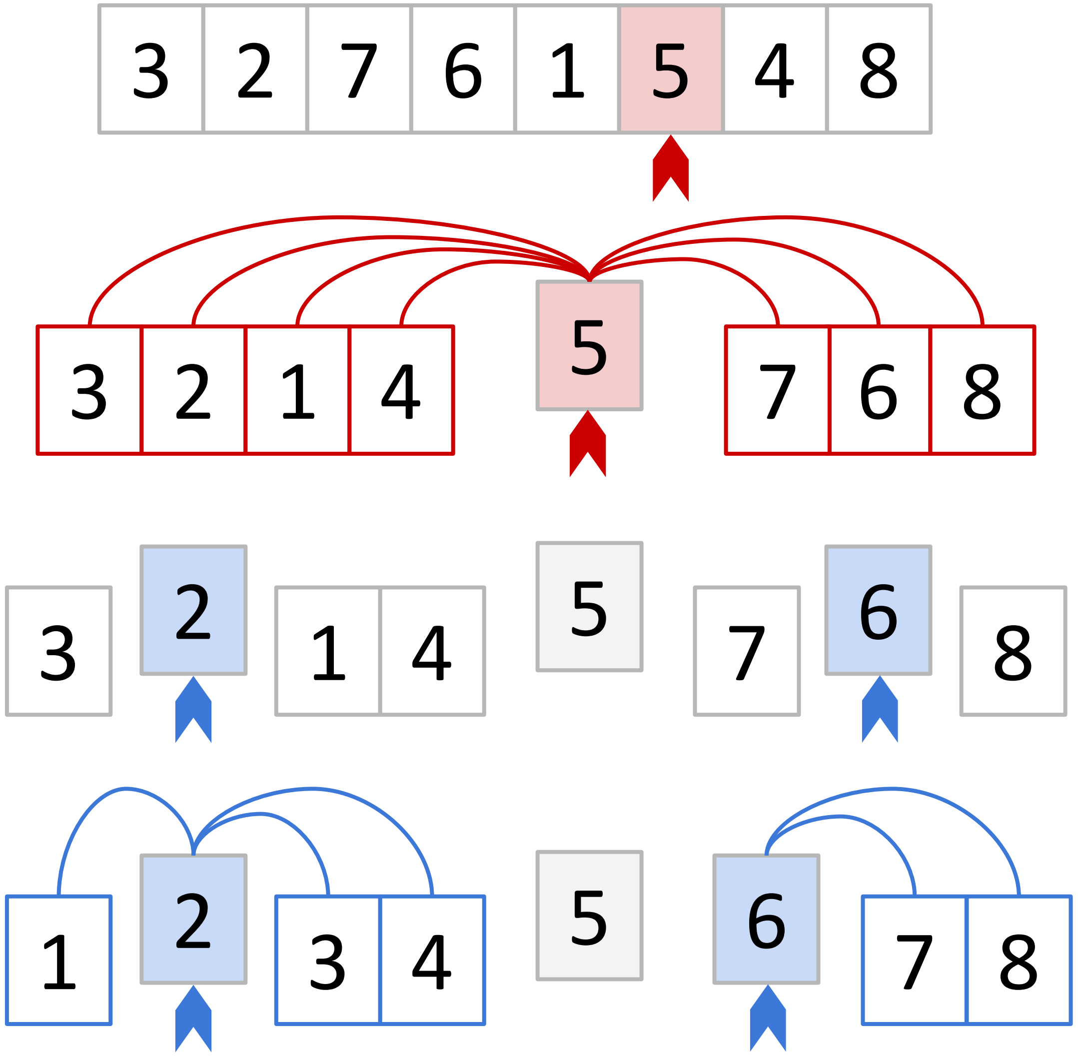
Everything is compared to 5 once in this first step… and then never again with **5**.

How Many Comparisons?



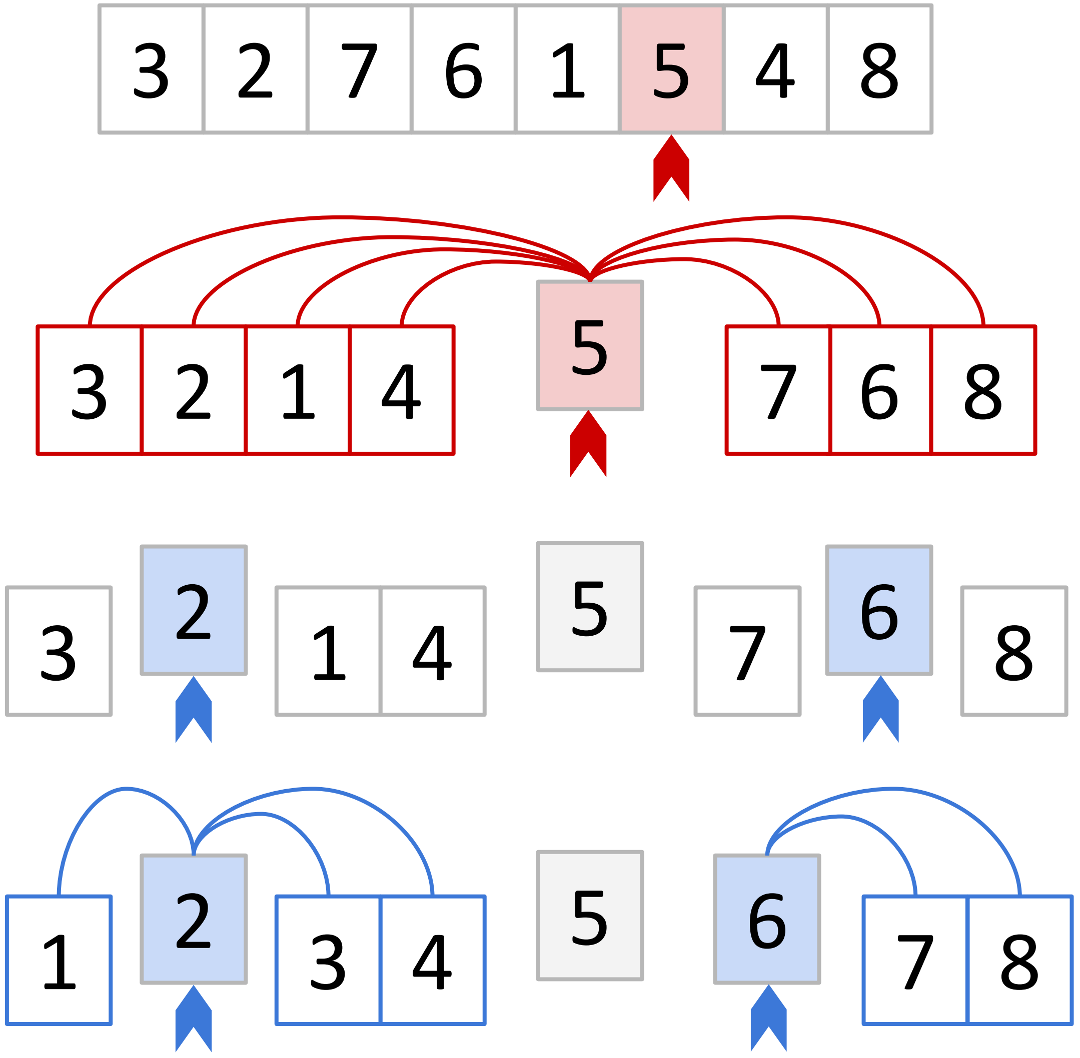
Everything is compared to 5 once in this first step… and then never again with **5**.

How Many Comparisons?



Everything is compared to 5 once in this first step… and then never again with **5**.

How Many Comparisons?



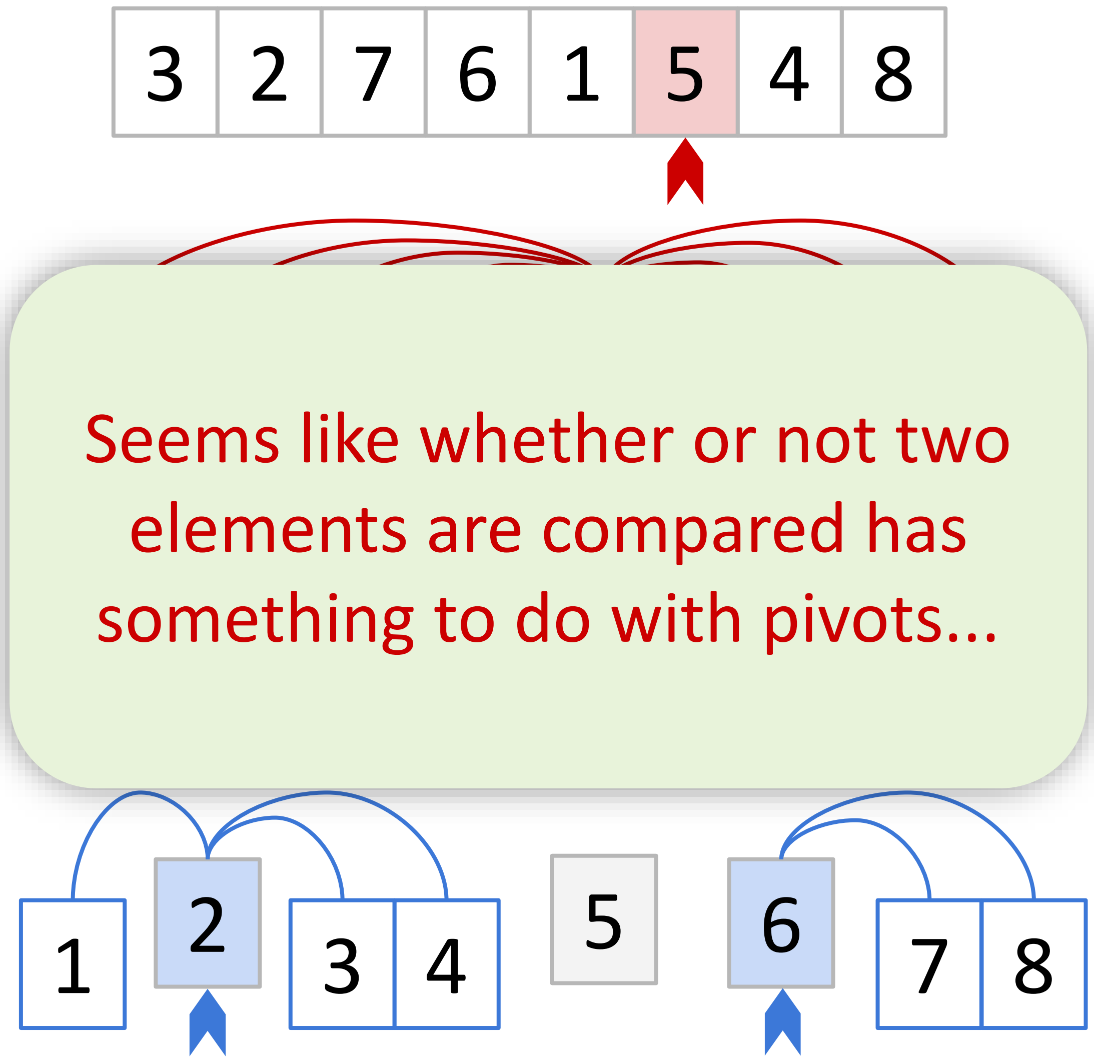
Everything is compared to 5 once in this first step… and then never again with **5**.

Only 1, 3, & 4 are compared to **2**.

And only 7 & 8 are compared with **6**.

No comparisons ever happen between two numbers on opposite sides of 5.

How Many Comparisons?



Seems like whether or not two elements are compared has something to do with pivots...

Everything is compared to 5 once in this first step... and then never again with **5**.

Only 1, 3, & 4 are compared to **2**.

And only 7 & 8 are compared with **6**.

No comparisons ever happen between two numbers on opposite sides of 5.

How Many Comparisons?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b} = 1}$ if **a** and **b** are compared

$\mathbf{X_{a,b} = 0}$ otherwise

How Many Comparisons?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b} = 1}$ if **a** and **b** are compared

$\mathbf{X_{a,b} = 0}$ otherwise

In our example, $\mathbf{X_{2,5}}$ took on the value **1** since **2** and **5** were compared.
On the other hand, $\mathbf{X_{3,7}}$ took on the value **0** since **3** and **7** are *not* compared.

How Many Comparisons?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b} = 1}$ if **a** and **b** are compared

$\mathbf{X_{a,b} = 0}$ otherwise

In our example, $\mathbf{X_{2,5}}$ took on the value **1** since **2** and **5** were compared.
On the other hand, $\mathbf{X_{3,7}}$ took on the value **0** since **3** and **7** are *not* compared.

Total number of comparisons =

$$\mathbb{E}\left[\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} X_{a,b}\right]$$

How Many Comparisons?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b}}$ = **1** if **a** and **b** are compared

$\mathbf{X_{a,b}}$ = **0** otherwise

In our example, $\mathbf{X_{2,5}}$ took on the value **1** since **2** and **5** were compared.
On the other hand, $\mathbf{X_{3,7}}$ took on the value **0** since **3** and **7** are *not* compared.

Total number of comparisons =

$$\mathbb{E}\left[\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} X_{a,b}\right] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} \mathbb{E}\left[X_{a,b}\right]$$

by linearity of expectation!

How Many Comparisons?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b} = 1}$ if **a** and **b** are compared

$\mathbf{X_{a,b} = 0}$ otherwise

In our example, $\mathbf{X_{2,5}}$ took on the value **1** since **2** and **5** were compared.
On the other hand, $\mathbf{X_{3,7}}$ took on the value **0** since **3** and **7** are *not* compared.

Total number of comparisons =

$$\mathbb{E}\left[\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} X_{a,b}\right] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} \mathbb{E}[X_{a,b}]$$

by linearity of expectation!

We need to figure out this value!

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

$$E[X_{a,b}] = 1 \cdot P(X_{a,b} = 1) + 0 \cdot P(X_{a,b} = 0) = P(X_{a,b} = 1)$$

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

$$\mathbf{E[X_{a,b}] = 1 \cdot P(X_{a,b} = 1) + 0 \cdot P(X_{a,b} = 0) = P(X_{a,b} = 1)}$$

So, what's $\mathbf{P(X_{a,b} = 1)}$?

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

$$E[X_{a,b}] = 1 \cdot P(X_{a,b} = 1) + 0 \cdot P(X_{a,b} = 0) = P(X_{a,b} = 1)$$

So, what's $\mathbf{P(X_{a,b} = 1)}$?

It's the probability that **a** and **b** are compared. Consider this example:

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

$\mathbf{P(X_{3,7} = 1)}$ is the probability that **3** and **7** are compared.

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

$$E[X_{a,b}] = 1 \cdot P(X_{a,b} = 1) + 0 \cdot P(X_{a,b} = 0) = P(X_{a,b} = 1)$$

So, what's $\mathbf{P(X_{a,b} = 1)}$?

It's the probability that **a** and **b** are compared. Consider this example:

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

$\mathbf{P(X_{3,7} = 1)}$ is the probability that **3** and **7** are compared.

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

This is exactly the probability that either 3 or 7 is first picked to be a pivot out of the highlighted entries.

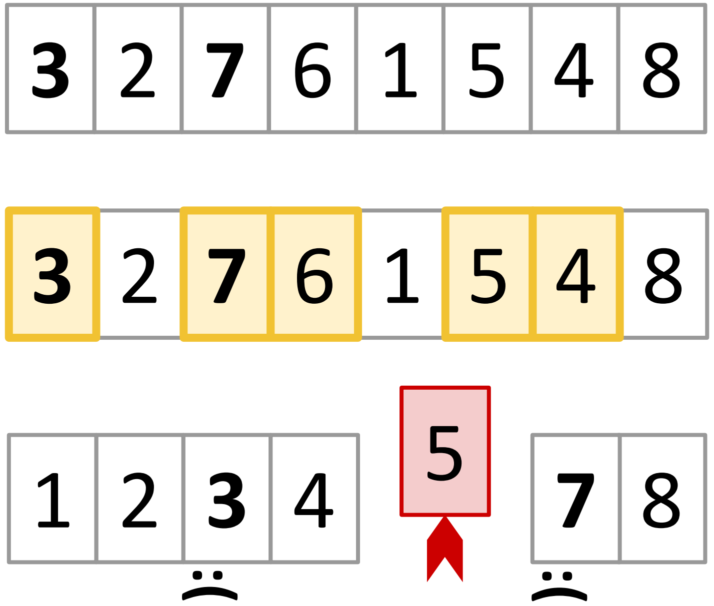
How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

$$\mathbf{E[X_{a,b}] = 1 \cdot P(X_{a,b} = 1) + 0 \cdot P(X_{a,b} = 0) = P(X_{a,b} = 1)}$$

So, what's $\mathbf{P(X_{a,b} = 1)}$?

It's the probability that **a** and **b** are compared. Consider this example:



$\mathbf{P(X_{3,7} = 1)}$ is the probability that **3** and **7** are compared.

This is exactly the probability that either 3 or 7 is first picked to be a pivot out of the highlighted entries.

If 4, 5, or 6 get picked as a pivot first, then 3 and 7 would be separated and never see each other again.

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

E[X$_{a,b}$

So, w

It's t

3	2

3	2

first

.

P(X$_{a,b}$ = 1) *aka probability that **a** & **b** are compared*

=

probability that either **a** or **b** are selected as a pivot before elements between **a** and **b**.

=

$$\frac{2}{\text{(\# elements from } \mathbf{a} \text{ to } \mathbf{b}\text{, inclusive)}}$$

1	2	**3**	4	5	**7**	8	

If 4, 5, or 6 get picked as a pivot first, then 3 and 7 would be separated and never see each other again.

How Many Comparisons?

So, what's $\mathbf{E[X_{a,b}]}$?

P($X_{a,b}$ = 1) *aka probability that **a** & **b** are compared*

=

probability that either **a** or **b** are selected as a pivot before elements between **a** and **b**.

=

$$\frac{2}{b - a + 1}$$

1	2	**3**	4

5

7	8

If 4, 5, or 6 get picked as a pivot first, then 3 and 7 would be separated and never see each other again.

Quick Sort Expected Runtime

Total number of comparisons = $$\sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} \mathbb{E}\left[X_{a,b}\right]$$

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}\left[X_{a,b}\right]=\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b,} = 1)$

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} \mathbb{E}\left[X_{a,b}\right] = \sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} \frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b,} = 1)$

$$= \sum_{a=0}^{n-2} \sum_{c=1}^{n-a-1} \frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b} = 1)$

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b,} = 1)$

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

$$= 2n\sum_{c=1}^{n-1}\frac{1}{c+1}$$

Nothing in the summation depends on a, so pull 2 out

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b} = 1)$

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

$$= 2n\sum_{c=1}^{n-1}\frac{1}{c+1}$$

Nothing in the summation depends on a, so pull 2 out

$$\leq 2n\sum_{c=1}^{n-1}\frac{1}{c}$$

decrease each denominator → we get the harmonic series!

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b,} = 1)$

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

$$= 2n\sum_{c=1}^{n-1}\frac{1}{c+1}$$

Nothing in the summation depends on a, so pull 2 out

$$\leq 2n\sum_{c=1}^{n-1}\frac{1}{c}$$

decrease each denominator → we get the harmonic series!

$$= O(n\log n)$$

Quick Sort Expected Runtime

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b} = 1)$

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

$$= 2n\sum_{c=1}^{n-1}\frac{1}{c+1}$$

Nothing in the summation depends on a, so pull 2 out

$$\leq 2n\sum_{c=1}^{n-1}\frac{1}{c}$$

decrease each denominator → we get the harmonic series!

$$= O(n\log n)$$

If E[# comparisons] = O(n log n), does this mean E[running time] is also O(n log n)?

YES! Intuitively, the runtime is dominated by comparisons.

Quick Sort

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Worst case runtime:
$O(n^2)$

Expected runtime:
$O(n \log n)$

Quick Sort in Practice

How is it implemented? Do people use it?

Implementing Quick Sort

In practice, a more clever approach is used to implement PARTITION, so that the entire QuickSort algorithm can be implemented “in-place”

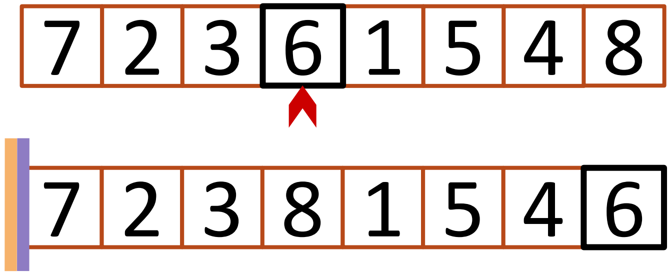
(i.e. via swaps, rather than constructing separate L or R subarrays)

An Example In-Place Partition

7	2	3	6	1	5	4	8

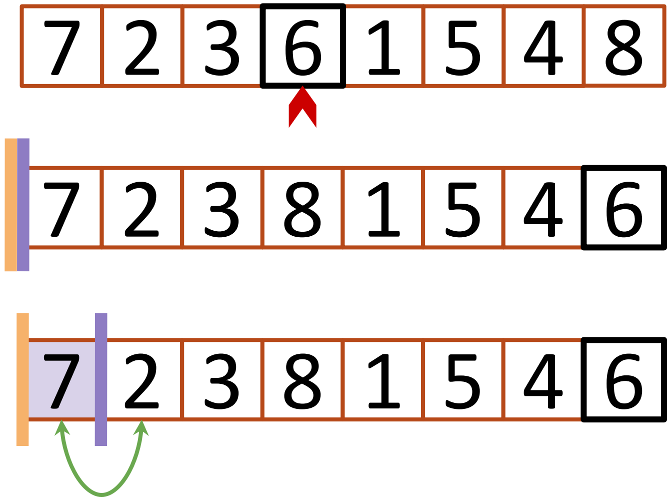
Choose pivot & swap with last element so pivot is at the end.

An Example In-Place Partition



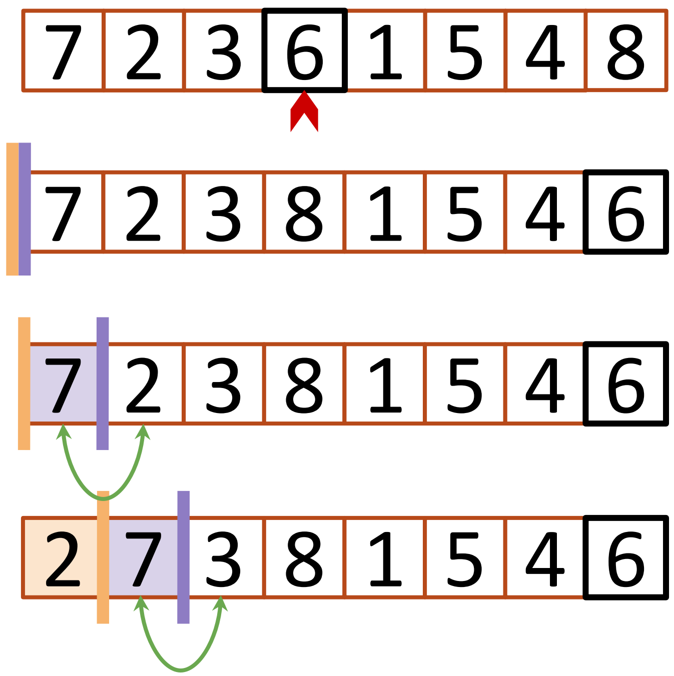
Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize and

An Example In-Place Partition



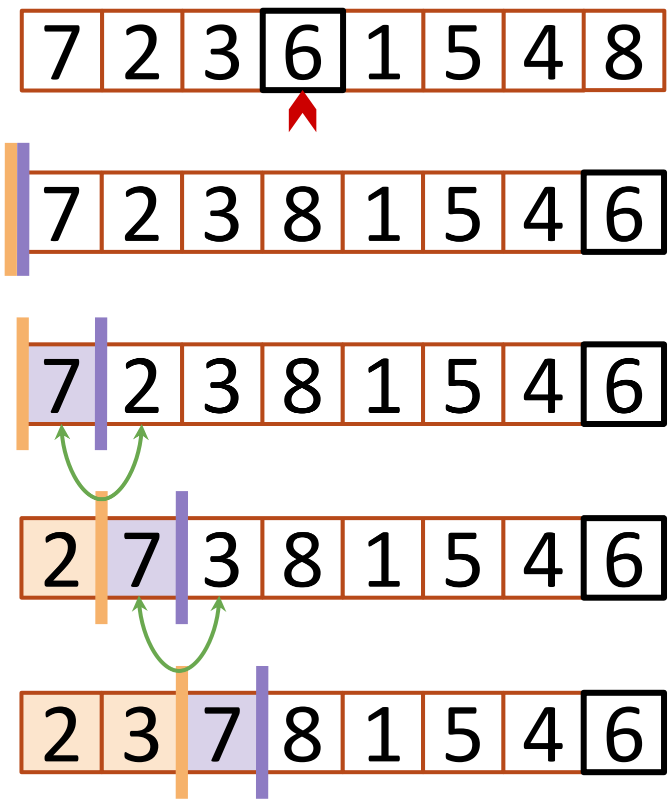
Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars

An Example In-Place Partition



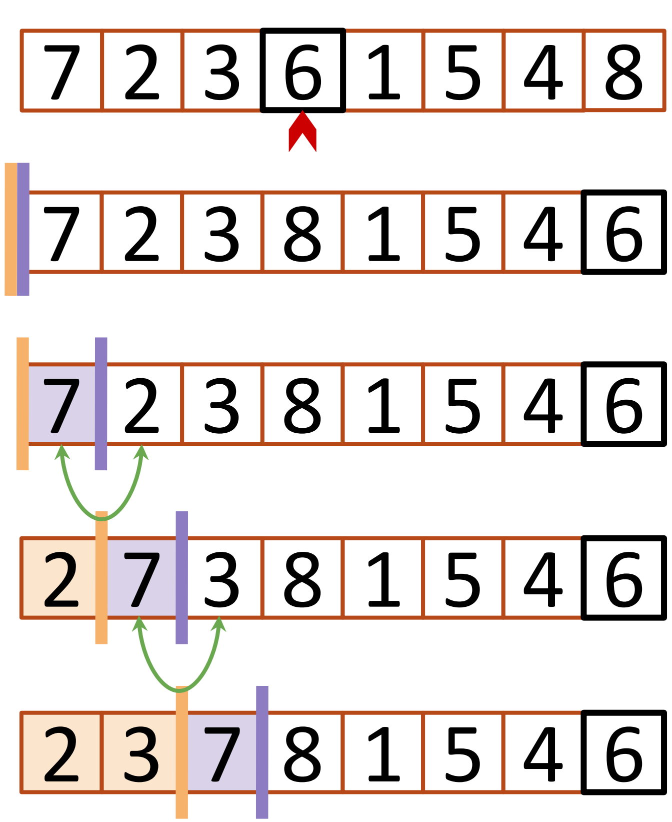
Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the | bar reaches the end, then swap the pivot into the right place.

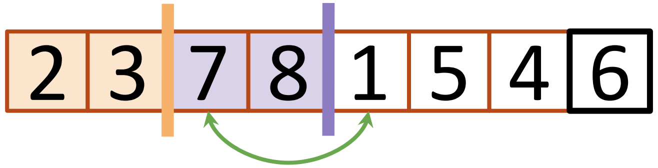
An Example In-Place Partition



Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the | bar reaches the end, then swap the pivot into the right place.

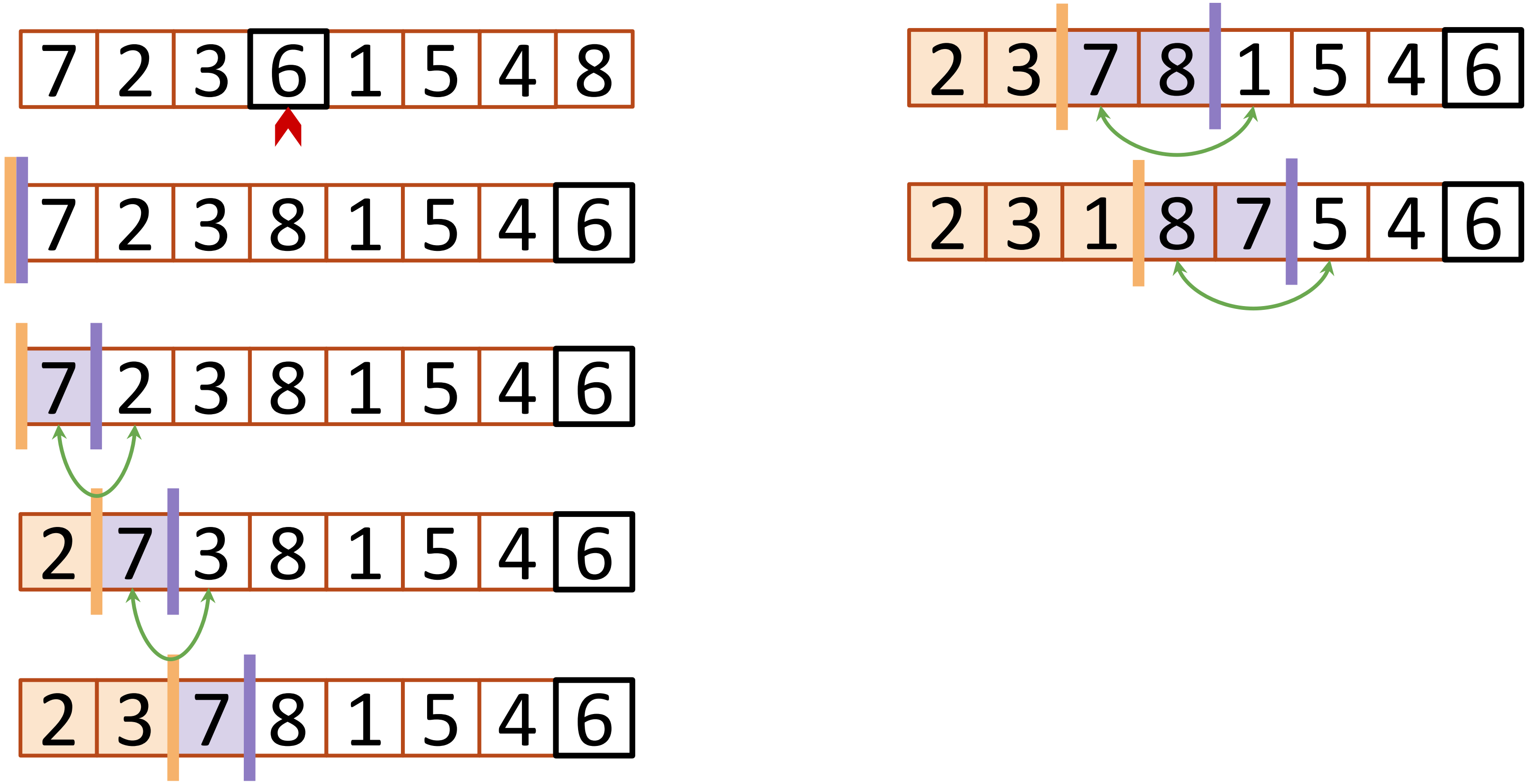
An Example In-Place Partition





Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the | bar reaches the end, then swap the pivot into the right place.

An Example In-Place Partition



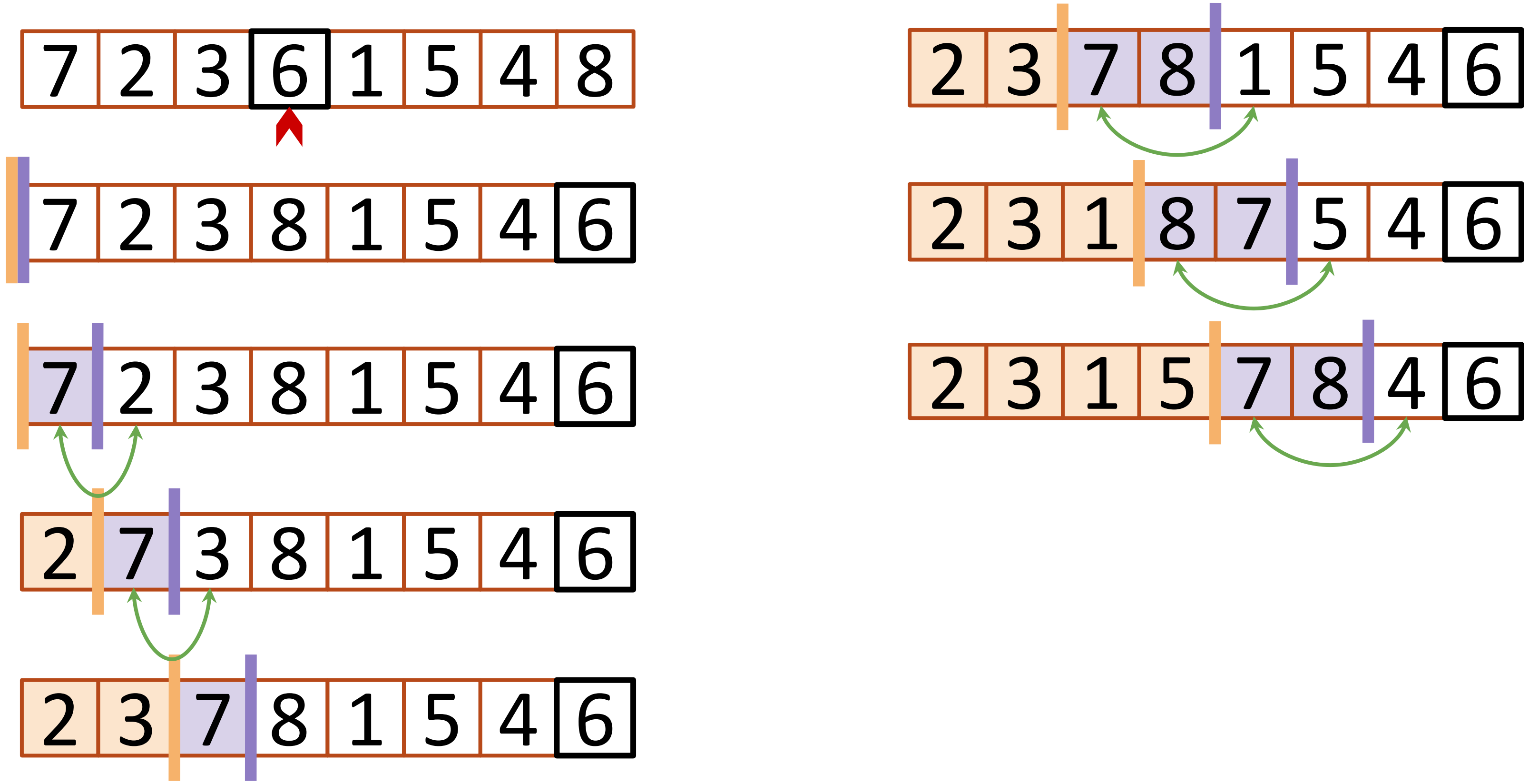
Choose pivot & swap with last element so pivot is at the end.

Initialize and

Increment until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars

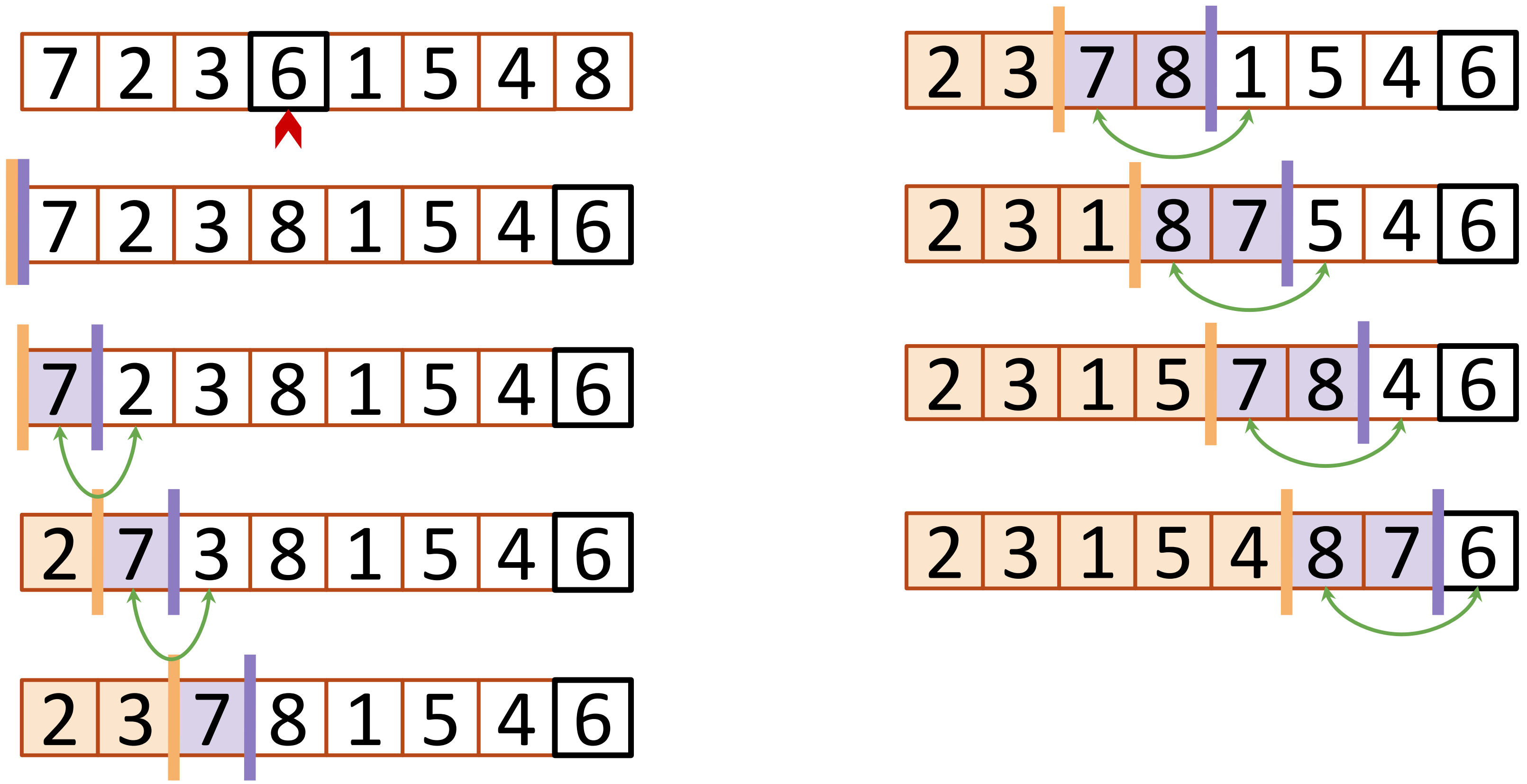
Repeat until the bar reaches the end, then swap the pivot into the right place.

An Example In-Place Partition



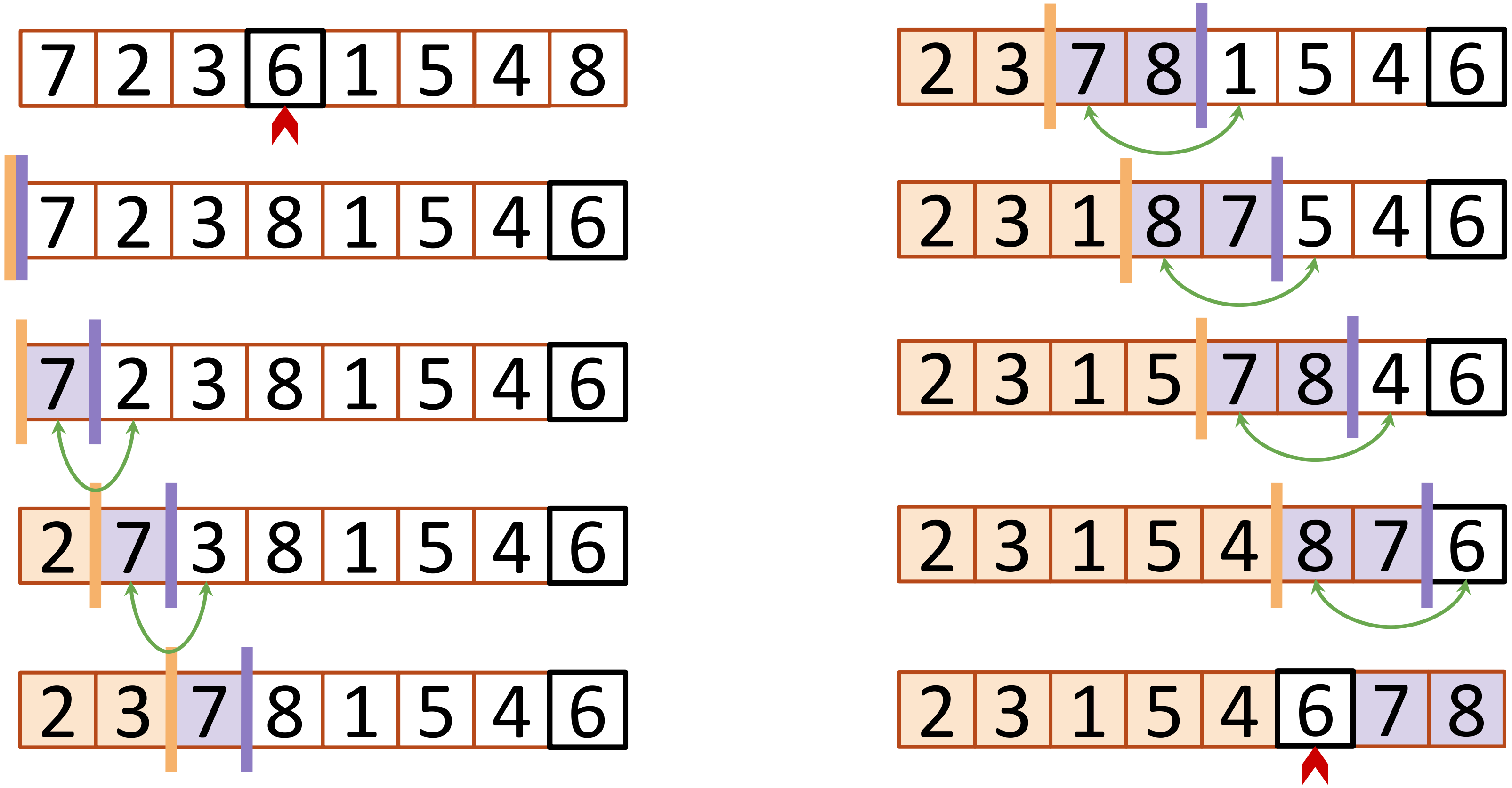
Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the | bar reaches the end, then swap the pivot into the right place.

An Example In-Place Partition



Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the | bar reaches the end, then swap the pivot into the right place.

An Example In-Place Partition



Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize | and | ⇨ Increment | until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the | bar reaches the end, then swap the pivot into the right place.

Quick Sort vs. Merge Sort

You do not need to understand any of this stuff

	QuickSort (random pivot)	**MergeSort** (deterministic)
Runtime	**Worst-case: $O(n^2)$** **Expected: O(n log n)**	**Worst-case: O(n log n)**
Used by	Java (primitive types), C (qsort), Unix, gcc…	Java for objects, perl
In-place? (i.e. with O(log n) extra memory)	Yes, pretty easily!	Easy if you sacrifice runtime (O(nlogn) MERGE runtime). Not so easy if you want to keep runtime & stability.
Stable?	No	Yes
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists

Recap

- Runtimes of **randomized algorithms** can be measured in two main ways:
  - Expected runtime (you roll the dice)
  - Worst-case runtime (the bad guy gets to fix the dice)
- **QUICKSORT!**
  - Another *DIVIDE and CONQUER* sorting algorithm that employs randomness
  - Elegant, structurally simple, and actually used in practice!

Acknowledgement

- Stanford University

Thank You