Randomized Algorithms I

Summer 2017 • Lecture 4

A Few Notes

Homework 1

- Solutions released.
- We will try to grade them by Friday morning.
- You will have a week to submit regrade requests from the release of grades.
- Submissions must be submitted before the hard deadline to receive credit.
- After late days have been exhausted, 25% 1 day, 50% 2 days.

Homework 2

- Due Friday 7/14 at 11:59 p.m. on Gradescope.

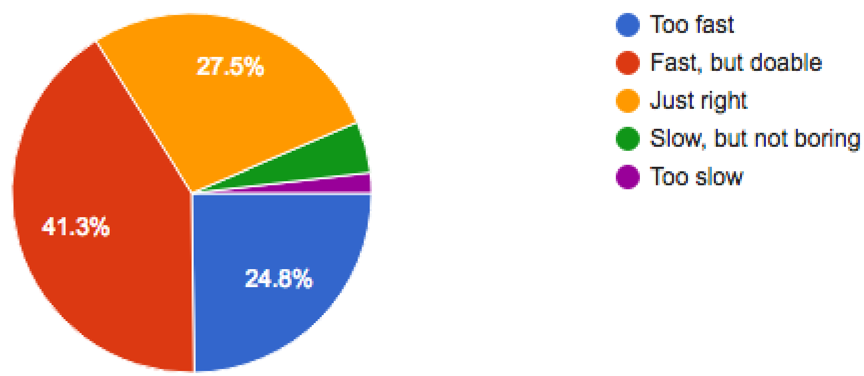
Outline for Today

- Randomized algorithms
  - Quicksort
  - Quickselect
  - Majority element

Week 1-2 Feedback

How do you find the pace of the course?

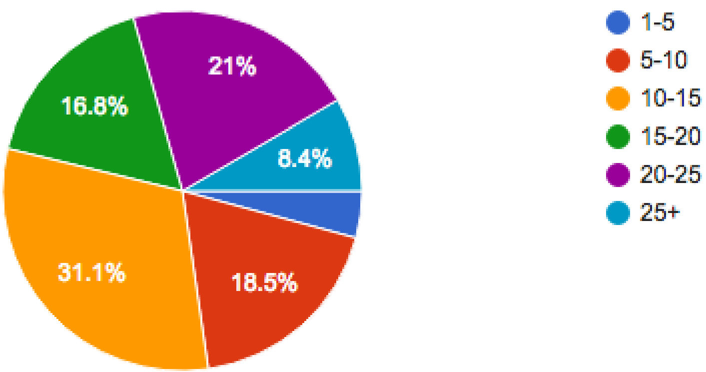
109 responses



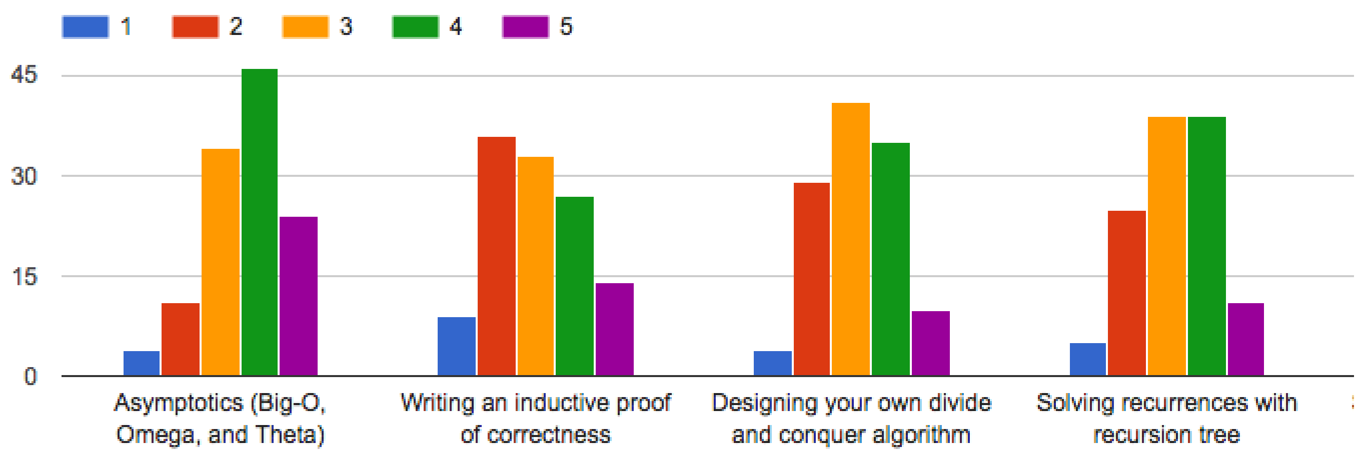
Week 1-2 Feedback

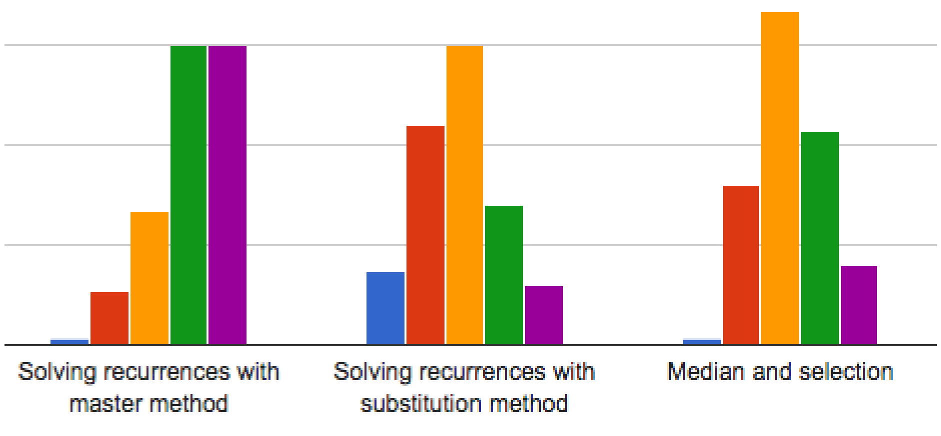
How many hours did you spend on Homework 1?

119 responses



Week 1-2 Feedback





Read our solutions for Homework 1!

Week 1-2 Feedback

Lectures

Focus less on mathematical derivations and notation, and more on visuals and specific examples.
I will highlight key points of the derivations, but leave the details to you and focus more on visuals showing the details of the algorithms.

Slides need more detail. **Okay, all of the detail!**

Better explanations. **Let me know when things aren't making sense.**

Week 1-2 Feedback

Lectures

Focus less on mathematical derivations and notation, and more on visuals and specific examples.
I will highlight key points of the derivations, but leave the details to you and focus more on visuals showing the details of the algorithms.

Slides need more detail. **Okay, all of the detail!**

Better explanations. **Let me know when things aren't making sense.**

Office Hours

I don't like group office hours!

I want more group office hours!

Randomized Algorithms

Randomized Algorithms

A randomized algorithm is an algorithm that incorporates randomness as part of its operation.

Often aim for properties like …

- Good average-case behavior
- Getting exact answers with high probability
- Getting answers that are close to the right answer

Randomized Algorithms

A randomized algorithm is an algorithm that incorporates randomness as part of its operation.

Often aim for properties like …

- Good average-case behavior
- Getting exact answers with high probability
- Getting answers that are close to the right answer

Monte Carlo vs. Las Vegas

- Las Vegas algorithms guarantee correctness, but not runtime.
  We will focus on these algorithms today.
- Monte Carlo algorithms guarantee runtime, but not correctness.
  We will revisit this next week when we see Karger's algorithm.

Properties of Expectation

[Expected prior knowledge]

The expected value of a constant or non-random variable is that constant or random variable itself: **$E[c] = c$**.

Expected value is a linear operator:

$$\mathbf{E[aX + b] = aE[X] + b}$$

$$\mathbf{E[X + Y] = E[X] + E[Y]}$$

Note that the second claim holds even if X and Y are dependent variables.

Bogosort

Our first example of a randomized algorithm is `bogosort`.

It’s not very smart.

Bogosort

```
algorithm bogosort(A):
  while True:
    randomly permute A
    if A is sorted:
      return A
```

Runtime

Bogosort

Unlike most of the deterministic algorithms that we've studied so far, when analyzing a Las Vegas randomized algorithms, we're interested in:

- What's the average-case runtime of the algorithm?
- How does this compare to the worst-case runtime of the algorithm?

Bogosort

```
algorithm bogosort(A):
  while True:
    randomly permute A
    if A is sorted:
      return A
```

Runtime

Expected: 🤔 Worst-case: 🤔

Bogosort

```
algorithm bogosort(A):
  while True:
    randomly permute A
    if A is sorted:
      return A
```

Runtime

Expected: $O(n \cdot n!)$ Worst-case: $O(\infty)$

Pr[randomly permuted array is sorted] = 1/n! By the expectation of geometric distribution (109), we expect to permute **A** n! times before it's sorted. Each permutation requires $O(n)$-time.

Think of this as the adversary chooses the randomness.

Quicksort

Quicksort

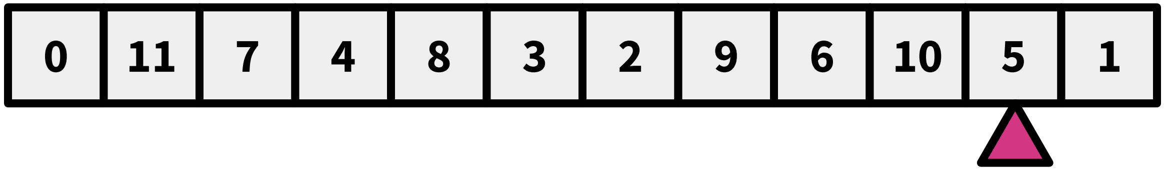
Our next example of a randomized algorithm is `quicksort`.

It's pretty smart.

It behaves as follows:

- If the list has 0 or 1 elements it's sorted.
- Otherwise, choose a pivot and partition around it.
- Recursively apply `quicksort` to the sublists to the left and right of the pivot.

Quicksort



Choose a pivot.

At random, a variant known as **randomized quicksort**.

Quicksort

0	11	7	4	8	3	2	9	6	10	5	1

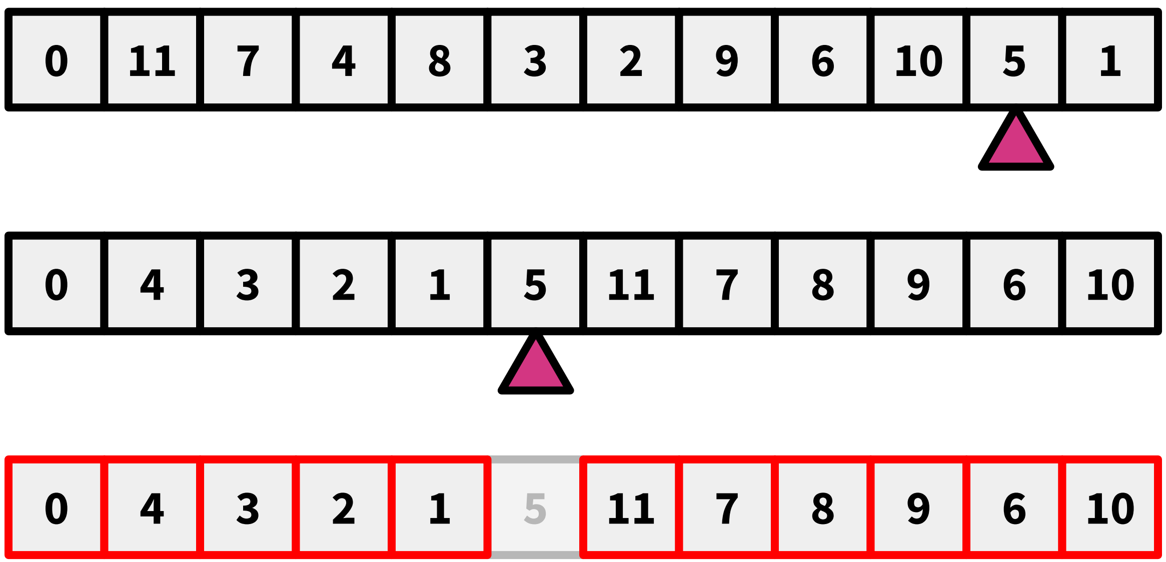
Choose a pivot.

At random, a variant known as **randomized quicksort**.

0	4	3	2	1	5	11	7	8	9	6	10

Partition around it.

Quicksort



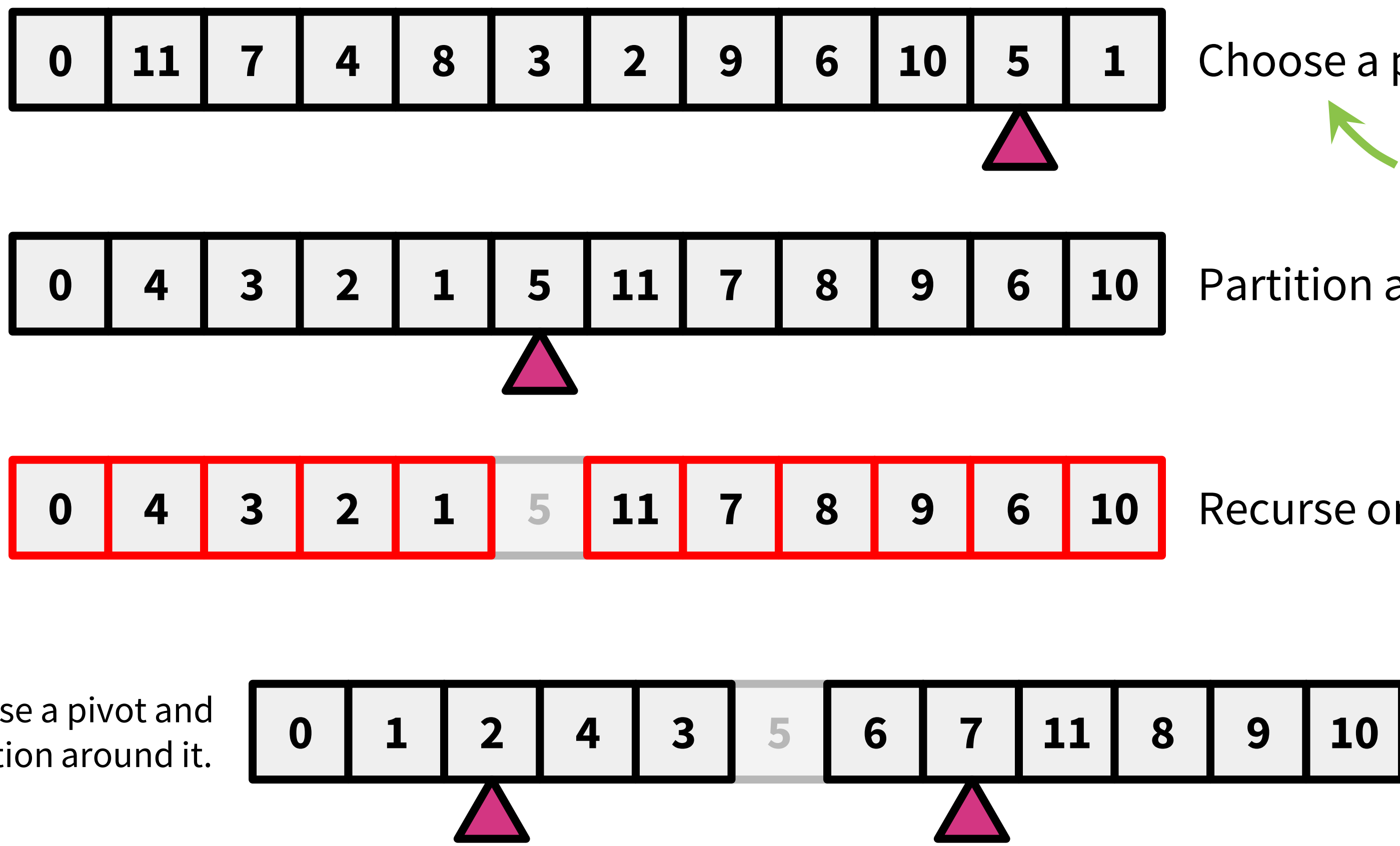
Choose a pivot.

At random, a variant known as **randomized quicksort**.

Partition around it.

Recurse on both subarrays.

Quicksort



Choose a pivot.

At random, a variant known as **randomized quicksort**.

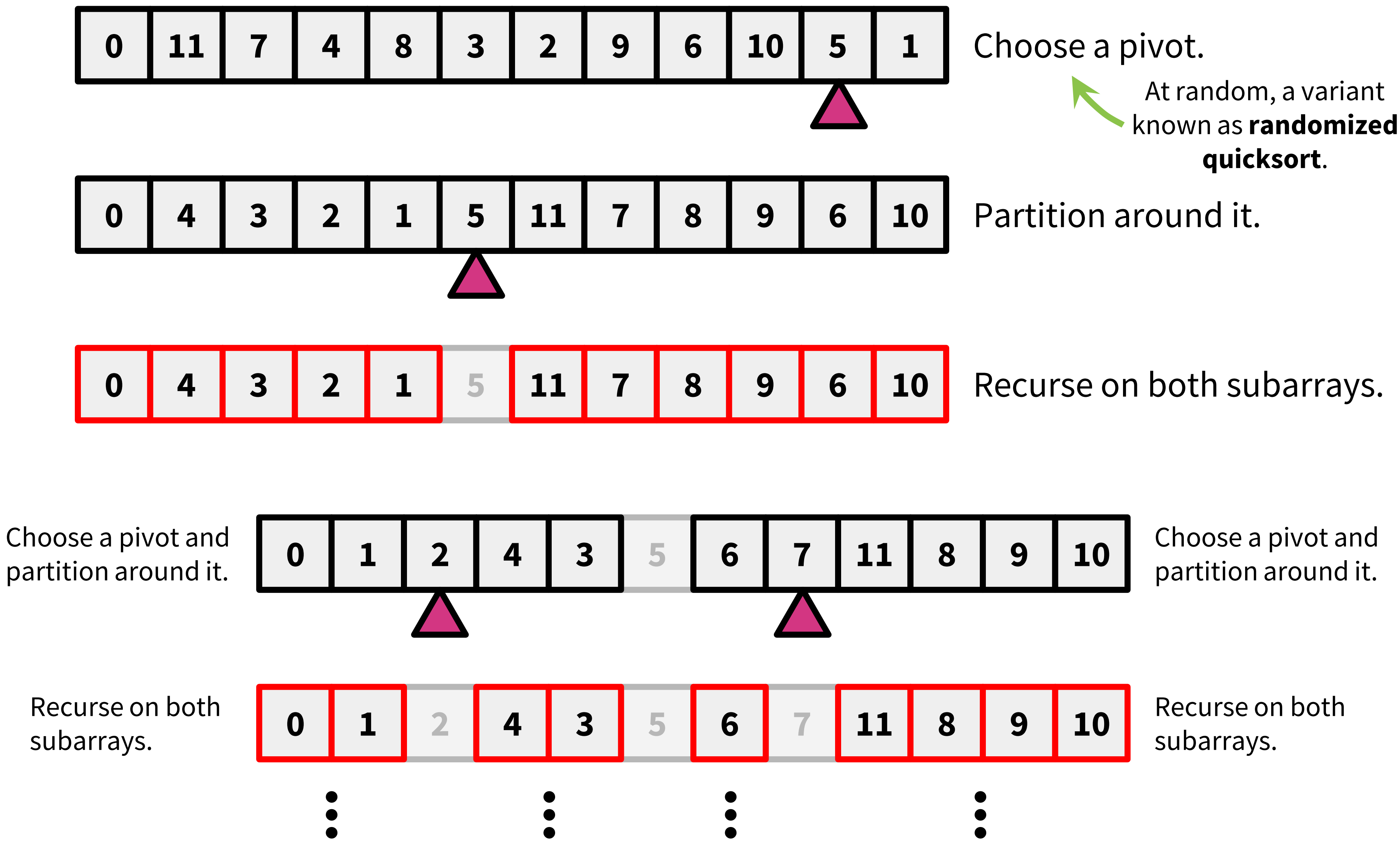
Partition around it.

Recurse on both subarrays.

Choose a pivot and partition around it.

Choose a pivot and partition around it.

Quicksort



0 11 7 4 8 3 2 9 6 10 5 1
Choose a pivot.
At random, a variant known as **randomized quicksort**.
0 4 3 2 1 5 11 7 8 9 6 10
Partition around it.
0 4 3 2 1 5 11 7 8 9 6 10
Recurse on both subarrays.
Choose a pivot and partition around it.
0 1 2 4 3 5 6 7 11 8 9 10
Choose a pivot and partition around it.
Recurse on both subarrays.
0 1 2 4 3 5 6 7 11 8 9 10
Recurse on both subarrays.

Quicksort

```
algorithm quicksort(A):
  if length(A) <= 1:
    return
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  quicksort(L)
  quicksort(R)
```

You can implement this to be in-place; try it out!

Runtime

Expected: 🤔　　Worst-case: 🤔

Quicksort

```
algorithm quicksort(A):
  if length(A) <= 1:
    return
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  quicksort(L)
  quicksort(R)
```

You can implement this to be in-place; try it out!

Runtime

Expected: $O(n\log n)$ Worst-case: $O(n^2)$

Think of this as the adversary chooses the randomness.

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

- What's the recurrence relation? 🤔

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

T(0) = T(1) = Θ(1)

T(n) = 2T(⌊n/2⌋) + Θ(n) Runtime of `partition`.

= `O(nlogn)` Master method a = 1, b = 2, d = 1.

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

T(0) = T(1) = Θ(1)

T(n) = 2T(⌊n/2⌋) + Θ(n) Runtime of `partition`.

= `O(nlogn)` Master method a = 1, b = 2, d = 1.

There's a really bad case, in which partition always picks the smallest or largest element as the pivot.

What's the recurrence relation? 🤔

Initial Observations

There's a really good case, in which partition always picks the median element as the pivot.

What's the recurrence relation? 🤔

$T(0) = T(1) = \Theta(1)$

$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$ Runtime of `partition`.

$= O(n\log n)$ Master method a = 1, b = 2, d = 1.

There's a really bad case, in which partition always picks the smallest or largest element as the pivot.

What's the recurrence relation? 🤔

$T(0) = T(1) = \Theta(1)$

$T(n) = T(n-1) + \Theta(n)$

$= O(n^2)$ Draw the recursion tree.

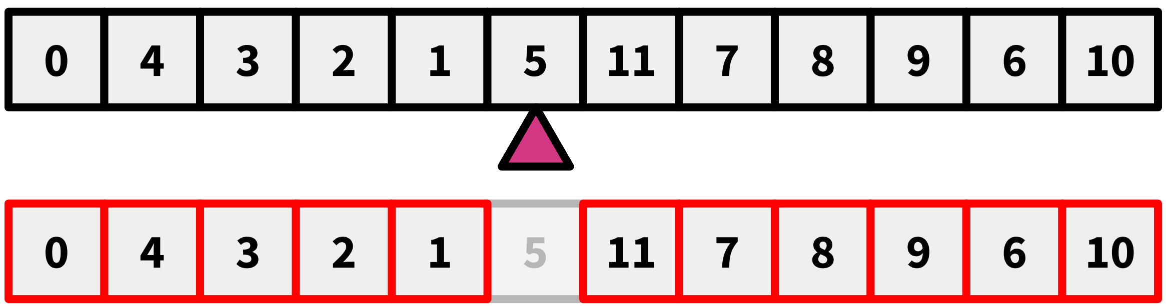
Expected Runtime of Quicksort

How do we know the expected runtime of `quicksort` is **`O(nlogn)`**?

To answer this question, let's count the number of times two elements get compared!

- This might not seem intuitive at first, but it's an approach you can use to analyze runtime of randomized algorithms.

Expected Runtime of Quicksort

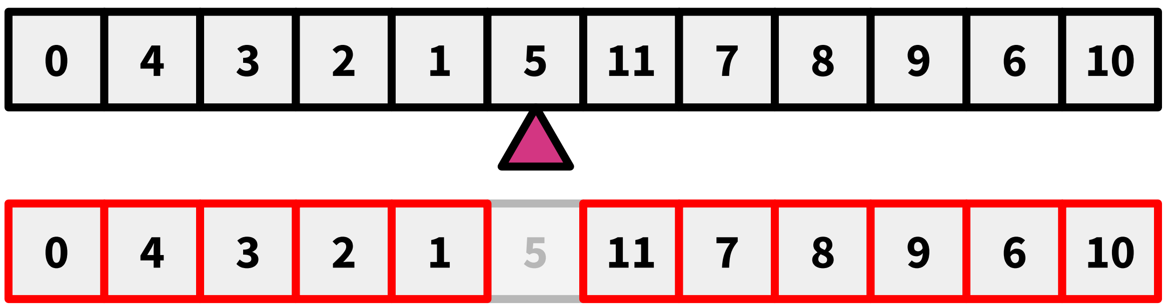


Partition around it.

Recurse on both subarrays.

All elements were compared to **5** in the top recursive call, and then never again.

Expected Runtime of Quicksort

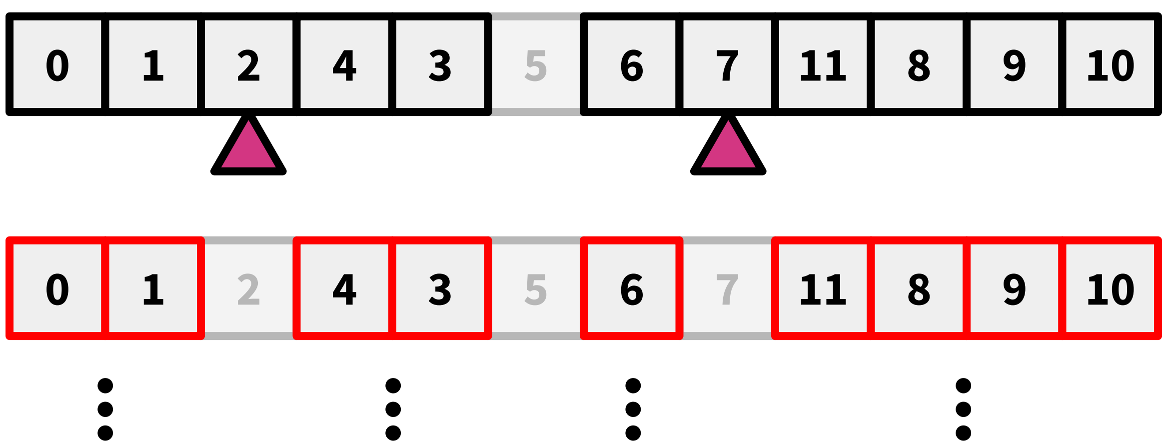


Partition around it.

Recurse on both subarrays.

All elements were compared to **5** in the top recursive call, and then never again.

Choose a pivot and partition around it.



Choose a pivot and partition around it.

Recurse on both subarrays.

Recurse on both subarrays.

Only the elements to the left of **5**, the original pivot, were compared to **2** in the left recursive call; only the elements to the right of the original pivot were compared to **7** in the right recursive call.

Expected Runtime of Quicksort

Each pair of elements **a** and **b** is compared 0 or 1 times.
Which is it?

Expected Runtime of Quicksort

Each pair of elements **a** and **b** is compared 0 or 1 times. Which is it?

Let $X_{a,b}$ be random variable that depends on choice of pivots, such that:

$$X_{a,b} = \begin{cases} 1 & \text{if } \mathbf{a} \text{ and } \mathbf{b} \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

Expected Runtime of Quicksort

Each pair of elements **a** and **b** is compared 0 or 1 times. Which is it?

Let $X_{a,b}$ be random variable that depends on choice of pivots, such that:

$$X_{a,b} = \begin{cases} 1 & \text{if } \mathbf{a} \text{ and } \mathbf{b} \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

In the previous example, $X_{3,5} = 1$ since **3** and **5** are compared but $X_{4,6} = 0$ since **4** and **6** are not compared.

Notice that these assignments of $X_{3,5}$ and $X_{4,6}$ both depended on our random choice of pivot **5**.

Expected Runtime of Quicksort

Each pair of elements **a** and **b** is compared 0 or 1 times. Which is it?

Let $X_{a,b}$ be random variable that depends on choice of pivots, such that:

$$X_{a,b} = \begin{cases} 1 & \text{if } \mathbf{a} \text{ and } \mathbf{b} \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

In the previous example, $X_{3,5} = 1$ since **3** and **5** are compared but $X_{4,6} = 0$ since **4** and **6** are not compared.

Notice that these assignments of $X_{3,5}$ and $X_{4,6}$ both depended on our random choice of pivot **5**.

The total number of comparisons?

Expected Runtime of Quicksort

Each pair of elements **a** and **b** is compared 0 or 1 times. Which is it?

Let $X_{a,b}$ be random variable that depends on choice of pivots, such that:

$$X_{a,b} = \begin{cases} 1 & \text{if } \mathbf{a} \text{ and } \mathbf{b} \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

In the previous example, $X_{3,5} = 1$ since **3** and **5** are compared but $X_{4,6} = 0$ since **4** and **6** are not compared.

Notice that these assignments of $X_{3,5}$ and $X_{4,6}$ both depended on our random choice of pivot **5**.

The total number of comparisons?

We need to figure out this value!

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

By linearity of expectation

Expected Runtime of Quicksort

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

By definition of expectation

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

Expected Runtime of Quicksort

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

By definition of expectation

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

To determine $P(X_{a,b} = 1)$, consider an example …

0	**6**	**8**	**11**	**1**	**10**	**2**	**7**	**3**	**9**	**4**	**5**

$P(X_{a,b} = 1)$ is the probability that **a** and **b** are compared.

$P(X_{6,10} = 1)$ is the probability that **6** and **10** are compared.

Expected Runtime of Quicksort

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

By definition of expectation

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

To determine $P(X_{a,b} = 1)$, consider an example …

0	6	8	11	1	10	2	7	3	9	4	5

$P(X_{a,b} = 1)$ is the probability that **a** and **b** are compared.

$P(X_{6,10} = 1)$ is the probability that **6** and **10** are compared.

This is the probability that either **6** and **10** are selected a pivot before **7**, **8**, or **9**. If we selected **7** as a pivot before either **6** or **10**, then **6** and **10** would be partitioned and not be compared.

Expected Runtime of Quicksort

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

By definition of expectation

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

To determine $P(X_{a,b} = 1)$, consider an example …

0	**6**	**8**	**11**	**1**	**10**	**2**	**7**	**3**	**9**	**4**	**5**

$P(X_{a,b} = 1)$ is the probability that **a** and **b** are compared.

$P(X_{6,10} = 1)$ is the probability that **6** and **10** are compared.

This is the probability that either **6** and **10** are selected a pivot before **7**, **8**, or **9**. If we selected **7** as a pivot before either **6** or **10**, then **6** and **10** would be partitioned and not be compared.

$= 2/5$

Expected Runtime of Quicksort

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

By definition of expectation

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

To determine $P(X_{a,b} = 1)$, consider an example …

0	**6**	**8**	**11**	**1**	**10**	**2**	**7**	**3**	**9**	**4**	**5**

$P(X_{a,b} = 1)$ is the probability that **a** and **b** are compared.

$P(X_{6,10} = 1)$ is the probability that **6** and **10** are compared.

This is the probability that either **6** and **10** are selected a pivot before **7**, **8**, or **9**. If we selected **7** as a pivot before either **6** or **10**, then **6** and **10** would be partitioned and not be compared.

$= 2/5$

Why doesn't this depend on the length of the overall list, 12? Consider an analogy: let's say you're playing the game: roll a die; if it's 1 you win, if it's 2 you lose, else roll again. You will win with probability 1/2, regardless of how many sides of the die!

Expected Runtime of Quicksort

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

By definition of expectation

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

To determine $P(X_{a,b} = 1)$, consider an example …

0	6	8	11	1	10	2	7	3	9	4	5

$P(X_{a,b} = 1)$ is the probability that **a** and **b** are compared.

$P(X_{6,10} = 1)$ is the probability that **6** and **10** are compared.

This is the probability that either **6** and **10** are selected a pivot before **7**, **8**, or **9**. If we selected **7** as a pivot before either **6** or **10**, then **6** and **10** would be partitioned and not be compared.

$= 2/5$

Why doesn't this depend on the length of the overall list, 12? Consider an analogy: let's say you're playing the game: roll a die; if it's 1 you win, if it's 2 you lose, else roll again. You will win with probability 1/2, regardless of how many sides of the die!

So, we can see that $\mathbf{P(X_{a,b} = 1) = 2 / (b - a + 1)}$

Expected Runtime of Quicksort

This gives that $E[X_{a,b}] = P(X_{a,b} = 1) = 2 / (b - a + 1)$. Thus,

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 2 / (b - a + 1)$$

Expected Runtime of Quicksort

This gives that $E[X_{a,b}] = P(X_{a,b} = 1) = 2 / (b - a + 1)$. Thus,

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 2 / (b - a + 1)$$

$$= \sum_{a=1}^{n} \sum_{c=1}^{n-a} 2 / (c + 1)$$

Expected Runtime of Quicksort

This gives that $E[X_{a,b}] = P(X_{a,b} = 1) = 2 / (b - a + 1)$. Thus,

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 2 / (b - a + 1)$$

$$= \sum_{a=1}^{n} \sum_{c=1}^{n-a} 2 / (c + 1)$$

$$\leq \sum_{a=1}^{n} \sum_{c=1}^{n} 2 / (c + 1)$$

This is the hard part,
and it's a useful skill.

Expected Runtime of Quicksort

This gives that $E[X_{a,b}] = P(X_{a,b} = 1) = 2 / (b - a + 1)$. Thus,

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 2 / (b - a + 1)$$

$$= \sum_{a=1}^{n} \sum_{c=1}^{n-a} 2 / (c + 1)$$

$$\leq \sum_{a=1}^{n} \sum_{c=1}^{n} 2 / (c + 1)$$

This is the hard part, and it's a useful skill.

$$= 2n \sum_{c=1}^{n} 1 / (c+1)$$

Expected Runtime of Quicksort

This gives that $E[X_{a,b}] = P(X_{a,b} = 1) = 2 / (b - a + 1)$. Thus,

$$\sum_{a=1}^{n}\sum_{b=a+1}^{n} E[X_{a,b}] = \sum_{a=1}^{n}\sum_{b=a+1}^{n} 2 / (b - a + 1)$$

$$= \sum_{a=1}^{n}\sum_{c=1}^{n-a} 2 / (c + 1)$$

$$\leq \sum_{a=1}^{n}\sum_{c=1}^{n} 2 / (c + 1)$$

This is the hard part, and it's a useful skill.

$$= 2n \sum_{c=1}^{n} 1 / (c+1) \leq 2n \sum_{c=1}^{n} 1/c$$

Harmonic series

Expected Runtime of Quicksort

This gives that $E[X_{a,b}] = P(X_{a,b} = 1) = 2 / (b - a + 1)$. Thus,

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 2 / (b - a + 1)$$

$$= \sum_{a=1}^{n} \sum_{c=1}^{n-a} 2 / (c + 1)$$

$$\leq \sum_{a=1}^{n} \sum_{c=1}^{n} 2 / (c + 1)$$

This is the hard part, and it's a useful skill.

$$= 2n \sum_{c=1}^{n} 1 / (c+1) \leq 2n \sum_{c=1}^{n} 1/c$$

Harmonic series

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

Quicksort

```
algorithm quicksort(A):
  if length(A) <= 1:
    return
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  quicksort(L)
  quicksort(R)
```

You can implement this to be in-place; try it out!

Runtime

Expected: $O(n \log n)$ Worst-case: $O(n^2)$

Think of this as the adversary chooses the randomness.

Better Quicksort?

Any ideas to make `quicksort` better? It still has worst-case $O(n^2)$-time.

- Recall that worst-case for randomized algorithms allows the adversary to control the randomness.

Better Quicksort?

Any ideas to make `quicksort` better? It still has worst-case $O(n^2)$-time.

Recall that worst-case for randomized algorithms allows the adversary to control the randomness.

We can borrow ideas from `select_k` and instead partition around the median of medians. It might also be a good idea to partition about the actual median or the median of three.

Quickselect

Quickselect

Our next example of a randomized algorithm is `quickselect`.

Quickselect

Our next example of a randomized algorithm is `quickselect`.

You’ve actually seen it before.

Quickselect

```
algorithm select_k(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: $O(n^2)$

Quickselect

```
algorithm quickselect(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: $O(n^2)$

Quickselect

```
algorithm quickselect(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

I didn't give you the entire story …

Runtime: $O(n^2)$

Quickselect

```
algorithm quickselect(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime

Expected: $O(n)$ Worst-case: $O(n^2)$

Think of this as the adversary chooses the randomness.

Expected Runtime of Quickselect

How do we know the expected runtime of `quickselect` is `O(n)`?

Let's refer to how we bounded the worst-case runtime for `select_k` with `smartly_choose_pivot`!

Expected Runtime of Quickselect

How do we know the expected runtime of `quickselect` is **`O(n)`**?

Let's refer to how we bounded the worst-case runtime for `select_k` with `smartly_choose_pivot`!

- `select_k` with `smartly_choose_pivot` upper-bounds the length of the list on which it recurses with 7n/10+c.

Expected Runtime of Quickselect

How do we know the expected runtime of `quickselect` is `O(n)`?

Let's refer to how we bounded the worst-case runtime for `select_k` with `smartly_choose_pivot`!

- `select_k` with `smartly_choose_pivot` upper-bounds the length of the list on which it recurses with 7n/10+c.
- Here, let's estimate the expected runtime of shrinking the length of the list to, say, 75% of the original length.

Expected Runtime of Quickselect

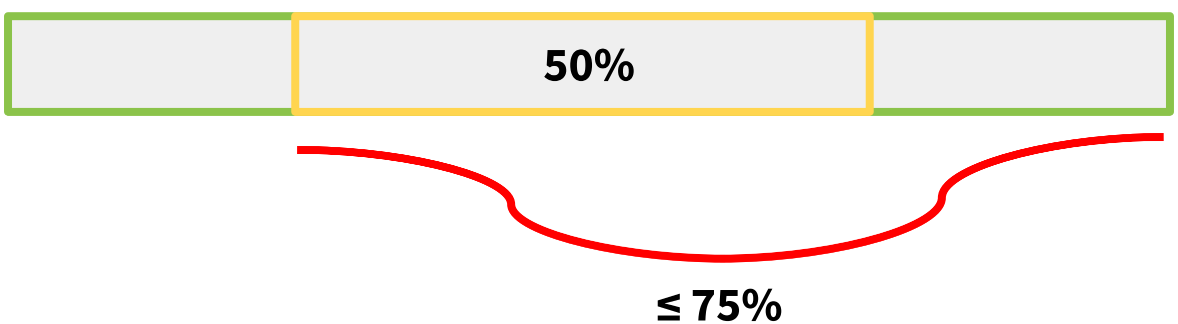
Let's define one "phase" of `quickselect` to be when it decreases the length of the input list to 75% of the original length or less.

Expected Runtime of Quickselect

Let's define one "phase" of `quickselect` to be when it decreases the length of the input list to 75% of the original length or less.

Why 75%?

Selecting a pivot in the middle 50% of all list values guarantees that the length of the input list decreases to below 75%.



A phase ends as soon as quickselect picks a pivot in the middle 50% of values.

Expected Runtime of Quickselect

Why at most?

If we number the phases 0, 1, 2, …

in phase k, the length of the list is at most $n(3/4)^k$ and the last phase is numbered $\lceil \log_{4/3} n \rceil$.

Expected Runtime of Quickselect

Why at most?

If we number the phases 0, 1, 2, …

in phase k, the length of the list is at most $n(3/4)^k$ and the last phase is numbered $\lceil \log_{4/3} n \rceil$.

Let X_k be a random variable equal to the number of recursive calls in phase k, and W be a random variable equal to the runtime.

Expected Runtime of Quickselect

Why at most?

If we number the phases 0, 1, 2, …

in phase k, the length of the list is at most $n(3/4)^k$ and the last phase is numbered $\lceil \log_{4/3} n \rceil$.

Let X_k be a random variable equal to the number of recursive calls in phase k, and W be a random variable equal to the runtime.

The runtime of phase k is at most $X_k \cdot cn(3/4)^k$, so:

$$W \leq \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot cn(3/4)^k = cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot (3/4)^k$$

Expected Runtime of Quickselect

Why at most?

If we number the phases 0, 1, 2, …

in phase k, the length of the list is at most $n(3/4)^k$ and the last phase is numbered $\lceil \log_{4/3} n \rceil$.

Let X_k be a random variable equal to the number of recursive calls in phase k, and W be a random variable equal to the runtime.

The runtime of phase k is at most $X_k \cdot cn(3/4)^k$, so:

$$W \leq \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot cn(3/4)^k = cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot (3/4)^k$$

And the expected runtime must be:

$$E[W] \leq E[cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot (3/4)^k]$$

Expected Runtime of Quickselect

Simplifying the expression gives …

$$E[W] \leq E[cn \sum_{k=0}^{[\log_{4/3} n]} X_k \cdot (3/4)^k]$$

Expected Runtime of Quickselect

Simplifying the expression gives …

$$E[W] \le E[cn \sum_{k=0}^{[\log_{4/3} n]} X_k \cdot (3/4)^k]$$

$$= cn \cdot E[\sum_{k=0}^{[\log_{4/3} n]} X_k \cdot (3/4)^k]$$

Expected Runtime of Quickselect

Simplifying the expression gives …

$$
\begin{aligned}
E[W] &\le E\left[cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot (3/4)^k\right] \\
&= cn \cdot E\left[\sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \cdot (3/4)^k\right] \\
&= cn \cdot \sum_{k=0}^{\lceil \log_{4/3} n \rceil} E[X_k \cdot (3/4)^k]
\end{aligned}
$$

Expected Runtime of Quickselect

Simplifying the expression gives …

$$E[W] \le E[cn \sum_{k=0}^{[\log_{4/3} n]} X_k \cdot (3/4)^k]$$

$$= cn \cdot E[\sum_{k=0}^{[\log_{4/3} n]} X_k \cdot (3/4)^k]$$

$$= cn \cdot \sum_{k=0}^{[\log_{4/3} n]} E[X_k \cdot (3/4)^k]$$

$$= cn \cdot \sum_{k=0}^{[\log_{4/3} n]} E[X_k](3/4)^k$$

The important part: How might we solve for $E[X_k]$?

Expected Runtime of Quickselect

How might we solve for $E[X_k]$?

Expected Runtime of Quickselect

How might we solve for $E[X_k]$?

- Recall X_k represents a random variable equal to the number of recursive calls in phase k.

Expected Runtime of Quickselect

How might we solve for $E[X_k]$?

Recall X_k represents a random variable equal to the number of recursive calls in phase k.

Since all pivot choices are independent, we have a geometric random variable with probability of success of ≥1/2 (since a phase ends as soon as `quickselect` picks a pivot in the middle 50% of values).

The first trial, probability of success is 1/2. If it fails, then the probability of success will be > 1/2 thereafter.

Expected Runtime of Quickselect

How might we solve for $E[X_k]$?

- Recall X_k represents a random variable equal to the number of recursive calls in phase k.
- Since all pivot choices are independent, we have a geometric random variable with probability of success of ≥1/2 (since a phase ends as soon as `quickselect` picks a pivot in the middle 50% of values). $E[X_k] \leq 1/(1/2) = 2$.

Expected Runtime of Quickselect

Simplifying the expression gives …

$$E[W] \leq cn \cdot \sum_{k=0}^{\lceil \log_{4/3} n \rceil} E[X_k](3/4)^k$$

Expected Runtime of Quickselect

Simplifying the expression gives …

$$E[W] \leq cn \cdot \sum_{k=0}^{[\log_{4/3} n]} E[X_k](3/4)^k$$

$$\leq cn \cdot \sum_{k=0}^{[\log_{4/3} n]} 2(3/4)^k$$

Expected Runtime of Quickselect

Simplifying the expression gives …

$$E[W] \le cn \cdot \sum_{k=0}^{\lceil \log_{4/3} n \rceil} E[X_k](3/4)^k$$

$$\le cn \cdot \sum_{k=0}^{\lceil \log_{4/3} n \rceil} 2(3/4)^k$$

$$\le cn \cdot \sum_{k=0}^{\infty} 2(3/4)^k$$

Expected Runtime of Quickselect

Simplifying the expression gives ...

$$E[W] \le cn \cdot \sum_{k=0}^{\lceil \log_{4/3} n \rceil} E[X_k](3/4)^k$$

$$\le cn \cdot \sum_{k=0}^{\lceil \log_{4/3} n \rceil} 2(3/4)^k$$

$$\le cn \cdot \sum_{k=0}^{\infty} 2(3/4)^k$$

This is the hard part, and it's a useful skill.

By the sum of infinite geometric series.

$$= 8cn$$

$$= \mathbf{O(n)}$$

Quickselect

```
algorithm quickselect(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime

Expected: $O(n)$ Worst-case: $O(n^2)$

3 min break

Majority Element

Majority Element

The **majority element problem** is the following: Given an input list A, find the element that occurs at least $\lfloor n/2 \rfloor + 1$ times, provided one exists.

Input accepts a list **A** and its length n.

Try to solve the same problem, but return NIL when one doesn't exist.

Majority Element

The **majority element problem** is the following: Given an input list A, find the element that occurs at least $\lfloor n/2 \rfloor + 1$ times, provided one exists.

Input accepts a list **A** and its length n.

Let's assume n is a power of 2 since dealing with this edge case isn't the point of the example.

Majority Element

The **majority element problem** is the following: Given an input list A, find the element that occurs at least $\lfloor n/2 \rfloor + 1$ times, provided one exists.

Input accepts a list **A** and its length n.

Let's assume n is a power of 2 since dealing with this edge case isn't the point of the example.

Additionally, suppose we can only perform the `equals` operation on the list, which accepts two values **a** and **b** and returns True if **a** equals **b**; otherwise returns False.

1	0	3	1	1	5	2	1	1	1	4	1

Majority Element

The **majority element problem** is the following: Given an input list A, find the element that occurs at least $\lfloor n/2 \rfloor + 1$ times, provided one exists.

Input accepts a list **A** and its length n.

Let's assume n is a power of 2 since dealing with this edge case isn't the point of the example.

Additionally, suppose we can only perform the `equals` operation on the list, which accepts two values **a** and **b** and returns True if **a** equals **b**; otherwise returns False.

1	0	3	1	1	5	2	1	1	1	4	1

`equals(A[0], A[2])` returns False

Majority Element

The **majority element problem** is the following: Given an input list A, find the element that occurs at least $\lfloor n/2 \rfloor + 1$ times, provided one exists.

Input accepts a list **A** and its length n.

Let's assume n is a power of 2 since dealing with this edge case isn't the point of the example.

Additionally, suppose we can only perform the `equals` operation on the list, which accepts two values **a** and **b** and returns True if **a** equals **b**; otherwise returns False.

1	**0**	**3**	**1**	**1**	**5**	**2**	**1**	**1**	**1**	**4**	**1**

`equals(A[0], A[2])` returns False

`equals(A[0], A[3])` returns True

Majority Element

The **majority element problem** is the following: Given an input list A, find the element that occurs at least $\lfloor n/2 \rfloor + 1$ times, provided one exists.

Input accepts a list **A** and its length n.

Let's assume n is a power of 2 since dealing with this edge case isn't the point of the example.

Additionally, suppose we can only perform the `equals` operation on the list, which accepts two values **a** and **b** and returns True if **a** equals **b**; otherwise returns False.

1	0	3	1	1	5	2	1	1	1	4	1

`equals(A[0], A[2])` returns False

`equals(A[0], A[3])` returns True

`equals(A[0], 1)` returns True

Majority Element

We will visit two solutions to this problem.

The first will be a divide-and-conquer algorithm; the second will be a randomized algorithm.

Majority Element

The divide-and-conquer approach …

- Recursive calls should return the majority element of a list's sublists.
- How might we merge two majority elements into a single majority element for this list? 🤔

Majority Element

The divide-and-conquer approach …

Recursive calls should return the majority element of a list’s sublists.

How might we merge two majority elements into a single majority element for this list? 🤔

0	**1**	**2**	**3**	**4**	**5**	**2**	**2**	**2**	**2**	**2**	**2**

`m_left = 5` `m_right = 2`

Majority Element

The divide-and-conquer approach …

Recursive calls should return the majority element of a list's sublists.

How might we merge two majority elements into a single majority element for this list? 🤔

0	**1**	**2**	**3**	**4**	**5**	**2**	**2**	**2**	**2**	**2**	**2**

`m_left = 5`   `m_right = 2`

Key insight: The majority element of entire list (if it exists) must be the same as the majority element as one of the sublists (otherwise it would occur at most $\lfloor n/2 \rfloor$ times).

Majority Element

The divide-and-conquer approach …

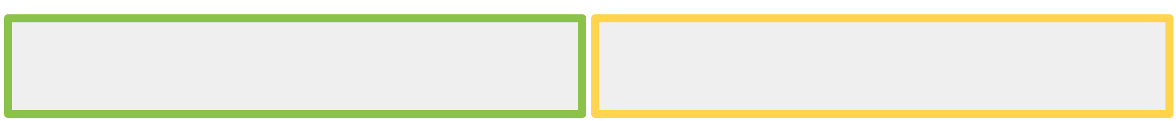
Recursive calls should return the majority element of a list’s sublists.

How might we merge two majority elements into a single majority element for this list? 🤔

0	1	2	3	4	5	2	2	2	2	2	2

`m_left = 5` `m_right = 2`

Key insight: The majority element of entire list (if it exists) must be the same as the majority element as one of the sublists (otherwise it would occur at most $\lfloor n/2 \rfloor$ times). To convince yourself of this case, consider if it’s possible for recursive calls to return these sublists if the majority element of the entire list isn’t 5 or 2.



`m_left = 5` `m_right = 2`

Majority Element

```
algorithm majority_element(A):
  # divide and conquer
  n = length(A), mid = (n-1)/2
  if n <= 1:
    return A[0]
  m1 = majority_element(A[0:mid])
  m2 = majority_element(A[mid+1:n-1])
  count = 0
  for a in A:
    if equals(m1, a): count += 1
  if count > n/2+1: return m1
  else: return m2
```

int division

Runtime: `O(nlogn)`

Count the number of calls to equals.

Recurrence: **$T(n) = 2T(n/2) + O(n)$**

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

We proceed by induction on i, such that $n = 2^i$.

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

We proceed by induction on i, such that $n = 2^i$.

Our base case, when i = 0, is trivially satisfied since `majority_element` returns **`A[0]`**.

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

We proceed by induction on i, such that $n = 2^i$.

Our base case, when i = 0, is trivially satisfied since `majority_element` returns **`A[0]`**.

Suppose `majority_element` is correct for inputs of length $n/2 = 2^{i-1}$. Now consider an input of length $n = 2^i$.

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

We proceed by induction on i, such that $n = 2^i$.

Our base case, when i = 0, is trivially satisfied since `majority_element` returns **`A[0]`**.

Suppose `majority_element` is correct for inputs of length $n/2 = 2^{i-1}$. Now consider an input of length $n = 2^i$. The majority element of the entire array, if it exists, must be the majority element of at least one of **`A[0:mid]`** or **`A[mid+1:n-1]`**; otherwise it would occur at most $\lfloor n/2 \rfloor$ times.

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

We proceed by induction on i, such that $n = 2^i$.

Our base case, when i = 0, is trivially satisfied since `majority_element` returns **`A[0]`**.

Suppose `majority_element` is correct for inputs of length $n/2 = 2^{i-1}$. Now consider an input of length $n = 2^i$. The majority element of the entire array, if it exists, must be the majority element of at least one of **`A[0:mid]`** or **`A[mid+1:n-1]`**; otherwise it would occur at most $\lfloor n/2 \rfloor$ times. Then the algorithm checks which one of these is the majority element and returns it.

Majority Element

Theorem: `majority_element` correctly finds the majority element of **`A`**, provided one exists.

Proof:

We proceed by induction on i, such that $n = 2^i$.

Our base case, when i = 0, is trivially satisfied since `majority_element` returns **`A[0]`**.

Suppose `majority_element` is correct for inputs of length $n/2 = 2^{i-1}$. Now consider an input of length $n = 2^i$. The majority element of the entire array, if it exists, must be the majority element of at least one of **`A[0:mid]`** or **`A[mid+1:n-1]`**; otherwise it would occur at most $\lfloor n/2 \rfloor$ times. Then the algorithm checks which one of these is the majority element and returns it.

Since the `majority_element` is called on the entire array, it can correctly find It, given that one exists. ∎

Majority Element

The randomized approach …

- Think about low-hanging fruit: will an algorithm similar to `bogosort` work?

Majority Element

The randomized approach …

- Think about low-hanging fruit: will an algorithm similar to `bogosort` work?
  - Choose a random index from 1 to n.
  - Is the element at that index the majority element?

Majority Element

```
algorithm majority_element(A):
  # randomized
  while True:
    i = random_int(0, n-1)  # random int {0,...,n-1}
    count = 0
    for a in A:
      if equals(A[i], a): count += 1
    if count > n/2+1: return A[i]
```

Runtime

Expected: 🤔 Worst-case: 🤔

Majority Element

```
algorithm majority_element(A):
  # randomized
  while True:
    i = random_int(0, n-1)  # random int {0,...,n-1}
    count = 0
    for a in A:
      if equals(A[i], a): count += 1
    if count > n/2+1: return A[i]
```

Runtime

Expected: $O(n)$ Worst-case: $O(\infty)$

Not all randomized algorithms have expected runtime O(n log n)!!! I don't want to see this everrr.

Expected Runtime of Majority Element

Provided there exists a majority element, this element must occur at least $\lfloor n/2 \rfloor + 1$ times.

- Let X be a geometric random variable for which success corresponds to finding the majority element; otherwise, failure.
- Since the algorithm finds the majority element with $p > 1/2$,
- E[# iterations through the while loop] $= 1/p < 2$.
- Each iteration requires n `equals` queries, so the expected runtime is **`O(n)`**.

Majority Element

Divide and Conquer Runtime

Expected & Worst-case: `O(nlogn)`

Randomized Runtime

Expected: `O(n)` Worst-case: `O(∞)`

Can you think of a deterministic algorithm that finds the majority element and only uses at most n - 1 calls to `equals`?

Get Hyped!

The randomized algorithmic paradigm appears everywhere in computer science.

As such, it will reappear throughout the quarter, starting next week with graph algorithms!