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.

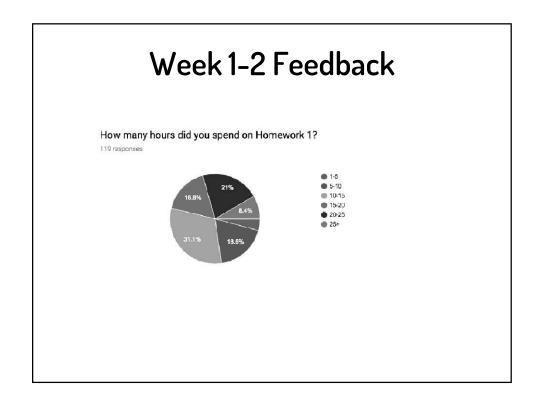
Outline for Today

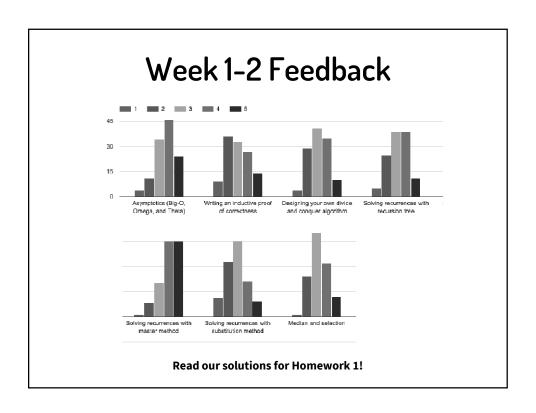
Randomized algorithms

Quicksort

Quickselect

Majority element





Week 1-2 Feedback

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:

E[aX + b] = aE[X] + b

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

Bogosort

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

Runtime

Expected: **O(n·n!)** Worst-case: **O(∞)**

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)**-

Think of this as the adversary chooses the randomness.

Quicksort

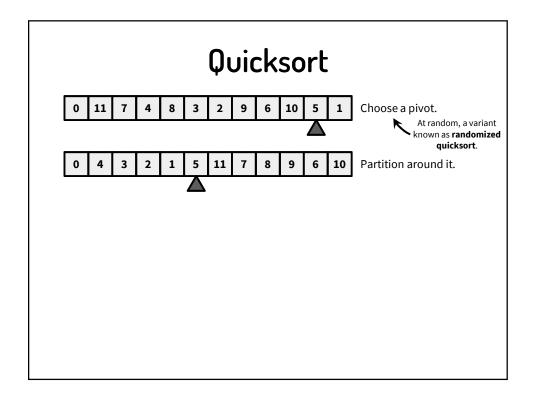
Quicksort

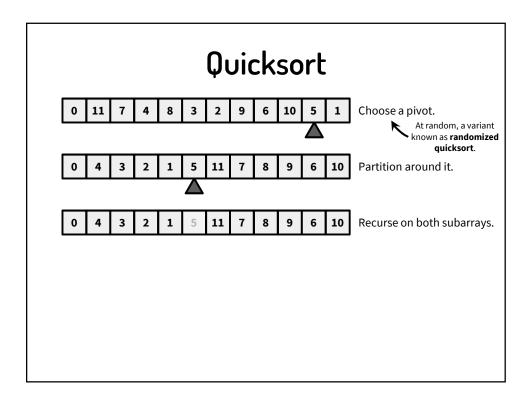
Our next example of a randomized algorithm is quicksort. It's pretty smart.

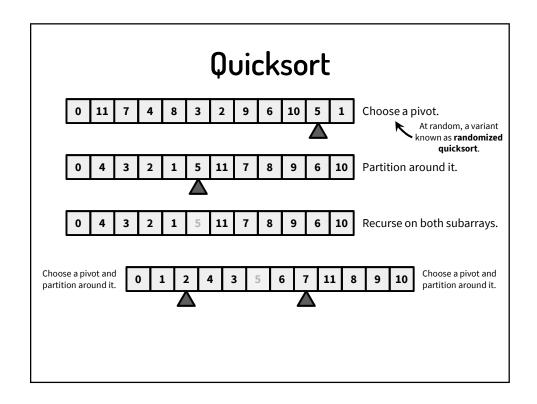
It behaves as follows:

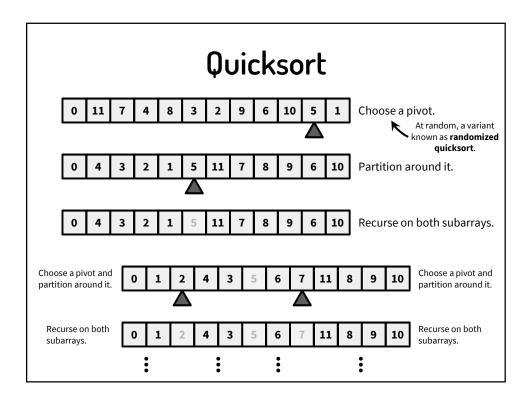
- If the list has 0 or 1 elements it's sorted.
- Otherwise, choose a pivot and partition around it.
- Recursively apply quicksort to the sublists to the left and right of the pivot.

Quicksort 0 11 7 4 8 3 2 9 6 10 5 1 Choose a pivot. At random, a variant known as randomized quicksort.









Quicksort

Runtime

Expected:

Worst-case:

Quicksort

Runtime

Expected: **O(nlogn)** Worst-case: **O(n²)**



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? (0) = T(1) = \Theta(1)
T(n) = 2T([n/2]) + \Theta(n) \qquad \text{Runtime of partition.}
= O(n \log n) \qquad \text{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? (0) = T(1) = \Theta(1)
T(n) = 2T([n/2]) + \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?

Initial Observations

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

```
What's the recurrence relation? (1) = T(1) = \Theta(1)
T(n) = 2T([n/2]) + \Theta(n) \qquad \text{Runtime of partition.}
= O(n \log n) \qquad \text{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.

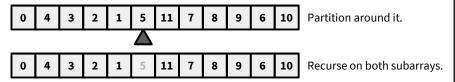
```
What's the recurrence relation? T(0) = T(1) = \Theta(1)
T(n) = T(n-1) + \Theta(n)
= O(n^2) \qquad \text{Draw the recursion tree.}
```

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

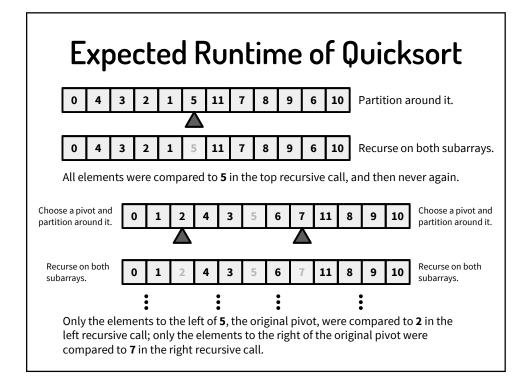
To answer this question, let's count the number of times two elements get compared!

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



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



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

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:

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

Expected Runtime of Quicksort

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

Let $\boldsymbol{X}_{a,b}$ be random variable that depends on choice of pivots, such that:

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

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**.

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:

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

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 ${\bf 5}.$

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:

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

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 $\bf 5$.

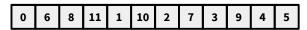
The total number of comparisons? We need to figure out this value!
$$E[\sum_{a=1}^{n}\sum_{b=a+1}^{n}X_{a,b}]=\sum_{a=1}^{n}\sum_{b=a+1}^{n}E[X_{a,b}]$$
 We need to figure out this value!

So what's
$$E[X_{a,b}]$$
?
$$E[X_{a,b}] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$$
By definition of expectation

Expected Runtime of Quicksort

So what's $E[X_{a,b}]$? $E[X_{a,b}] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$ By definition of expectation

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



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

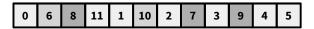
 $P(X_{6,10} = 1)$ is the probability that **6** and **10** are compared.

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 ...



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

 $P(X_{6.10} = 1)$ is the probability that **6** and **10** are compared.

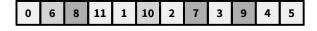
This is the probability that either 6 and 10 are selected a pivot before 7, 8, or 9. If we selected 7 as a pivot before either 6 or 10, then 6 and 10 would be partitioned and not be compared.

Expected Runtime of Quicksort

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

O What's
$$E[X_{a,b}]$$
?
$$E[X_{a,b}] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$$
By definition of expectation

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



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

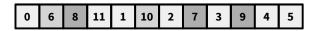
 $P(X_{6.10} = 1)$ is the probability that **6** and **10** are compared.

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

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 ...



 $P(X_{a,b} = 1)$ 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.

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 ...

11 1 10

 $P(X_{a,b} = 1)$ 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.

Why doesn't this depend on the length of the overall list, 12? Consider an analogy: Why doesn't this depend on the length of the overall tist, 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 $P(X_{a,b} = 1) = 2 / (b - a + 1)$

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,

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

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

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

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

Expected Runtime of Quicksort

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

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

$$\leq \sum_{a=1}^{n} \sum_{c=1}^{n} 2 / (c + 1)$$
This is the hard part, and it's a useful skill.
$$= 2n \sum_{c=1}^{n} 1 / (c+1) \leq 2n \sum_{c=1}^{n} 1/c$$

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

Quicksort

Runtime

Expected: **O(nlogn)** Worst-case: **O(n²)**



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)</pre>
```

Runtime: O(n2)

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)</pre>
```

Runtime: O(n²)

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)</pre>
```

Runtime: O(n²)

I didn't give you the entire story ...

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)</pre>

Runtime

Expected: O(n) Worst-case: O(n²)
```

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.

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.

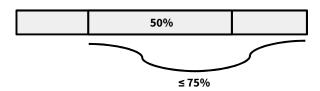
Expected Runtime of Quickselect

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.

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

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 $\lceil \log_{4/3} n \rceil$.

If we number the phases 0, 1, 2, ... Why at most?

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

If we number the phases 0, 1, 2, ... Why at most?

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:

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

If we number the phases $0, 1, 2, \dots$

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

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:

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

And the expected runtime must be:

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

Expected Runtime of Quickselect

Simplifying the expression gives ...

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

Simplifying the expression gives ...

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

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

Expected Runtime of Quickselect

Simplifying the expression gives ...

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

Simplifying the expression gives ...

$$\begin{split} E[W] &\leq E[cn\sum_{k=0}^{\lceil \log_{x/3} n \rceil} X_k \cdot (3/4)^k] \\ &= cn \cdot E[\sum_{k=0}^{\lceil \log_{x/3} n \rceil} X_k \cdot (3/4)^k] \\ &= cn \cdot \sum_{k=0}^{\lceil \log_{x/3} n \rceil} E[X_k \cdot (3/4)^k] \\ &= cn \cdot \sum_{k=0}^{\lceil \log_{x/3} n \rceil} E[X_k] (3/4)^k \end{split}$$
The important part: How might we solve for E[X_k]?

Expected Runtime of Quickselect

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

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 $\mathbf{X}_{\mathbf{k}}$ represents a random variable equal to the number of recursive calls in phase $\mathbf{k}.$

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.

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

Recall $\mathbf{X}_{\mathbf{k}}$ represents a random variable equal to the number of recursive calls in phase $\mathbf{k}.$

Since all pivot choices are independent, we have a geometric random variable with probability of success of $\geq 1/2$ (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 ...

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

Simplifying the expression gives ...

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

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

Expected Runtime of Quickselect

Simplifying the expression gives ...

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

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

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

Simplifying the expression gives ...

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

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

$$\le cn \cdot \sum_{k=0}^{\infty} 2(3/4)^k$$
This is the hard part, and it's a useful skill.
$$= 8cn$$

$$= 8cn$$

$$= 0(n)$$

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)</pre>
```

Runtime

Expected: **O(n)** Worst-case: **O(n²)**

3 min break **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.

Let's assume n is a power of 2

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.

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.

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.



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.

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.



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

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.

Let's assume n is a power of 2

Input accepts a list A and its length n.

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.



equals(A[0], A[2]) returns False equals(A[0], A[3]) returns True

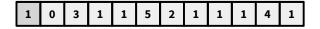
Majority Element

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.

Let's assume n is a power of 2

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

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.



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

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

equals(A[0], 1) returns True

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 ...

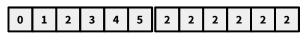
Recursive calls should return the majority element of a list's sublists.

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

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? \$



 $m_left = 5$

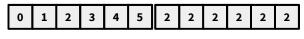
 $m_right = 2$

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? \$



 $m_left = 5$

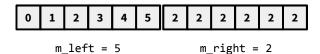
 $m_right = 2$

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).

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?



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.



Majority Element

```
Runtime: O(nlogn) Count the number of calls to equals.

Recurrence: T(n) = 2T(n/2) + O(n)
```

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 $n = 2^{i}$.

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 $n = 2^{i}$.

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 [n/2] times. Then the algorithm checks which one of these is the majority element and returns it.

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. \square

Majority Element

The randomized approach ...

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

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: (5) Worst-case: (5)

```
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: **0(n)** Worst-case: **0(∞)**

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

Expected Runtime of Majority Element

Provided there exists a majority element, this element must occur at least |n/2| + 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)**.

Divide and Conquer Runtime Expected & Worst-case: **O(nlogn)**

Randomized Runtime
Expected: **O(n)** Worst-case:

1 0(∞)

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!