# Randomized Algorithms I

# Outline for Today

Randomized algorithms

- Quicksort
- Quickselect
- Majority element

# Randomized Algorithms

# Randomized Algorithms

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

Randomized vs. Deterministic

Deterministic: guaranteed correctness & worst-case runtime

Randomized: either correctness or worst-case runtime is not guaranteed

Why? Trade-off between Accuracy and Efficiency

Often aim for properties like:

- Correctness is not guaranteed but gives good worst-case runtime
- Worst-case runtime is not guaranteed but gives good average runtime

- Getting exact answers with high probability
- Getting answers that are close to the right answer

# Randomized Algorithms

Two types of randomized algorithms

Las Vegas vs Monte Carlo

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 lecture 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 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, we expect to permute **A** n! times before it's sorted.
Each permutation requires $O(n)$-time.

There is a possibility that you never happen to sort the list.

Explain on board with example

# Quicksort

# Quicksort

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

It's pretty smart.

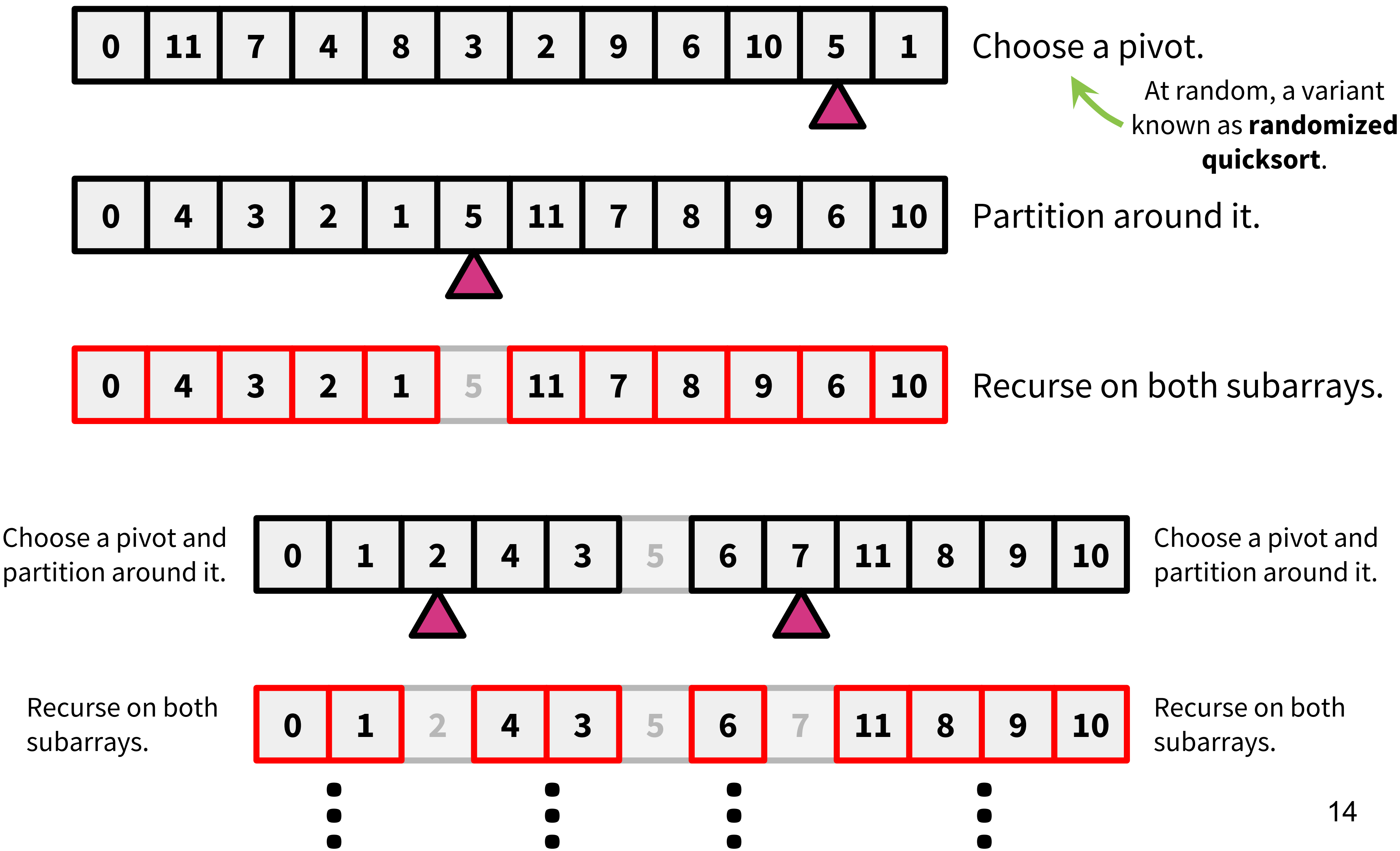
It behaves as follows:

Boundary case: If the list has 0 or 1 elements it's sorted.

Partition: Otherwise, choose a pivot and partition around it.

Recursion: Recursively apply `quicksort` to the sub-lists 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**.
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: $O(n^2)$

# 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 $a = 2$, $b = 2$, $d = 1$.

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

# 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 = 2, b = 2, d = 1.

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

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 or just add-up the total times of number comparisons.

# 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!

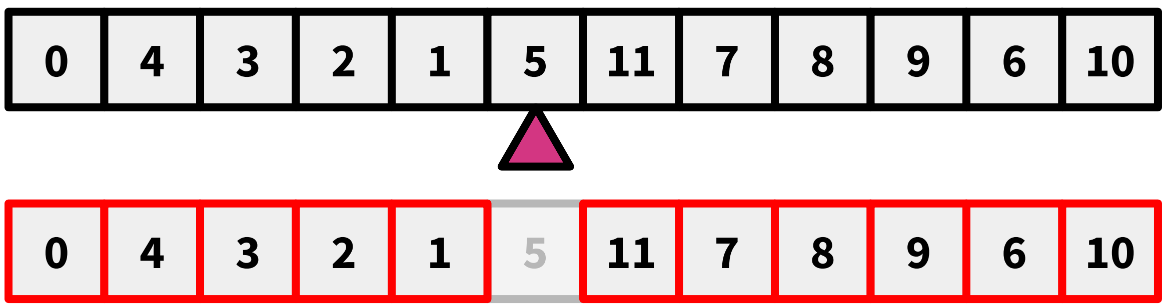
This might not seem intuitive at first, but it's an approach you can use to analyze runtime of randomized algorithms.

Note: We only need to analyze the runtime of soring n different numbers 0~n-1

Why: 1. Sorting an array that contains repeated values is just easier

2. Sorting an array with n different elements is equal to sorting an array with elements 0~n-1

# Expected Runtime of Quicksort

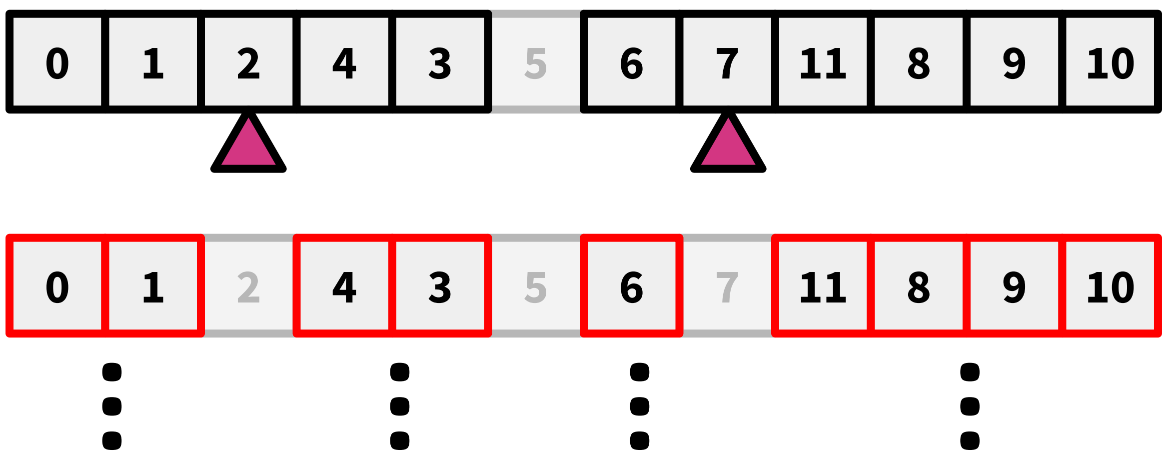



Partition around it.

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?

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?

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

We need to figure out this value!

By linearity of expectation

# Expected Runtime of Quicksort

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

By definition of expectation

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

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** or **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 dice; 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 dice! In this case, we are rolling a 12-side dice and expecting 6 or 10 out of 6~10.

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,

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

$$= \sum_{a=0}^{n-1}\sum_{c=1}^{n-a-1} 2 / (c + 1)$$

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

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

$$= 2n \sum_{c=0}^{n-1} 1 / (c+1) = 2n \sum_{c=1}^{n} 1/c$$

Harmonic series

$$\leq 2n(\ln(n) + 1) = O(n \log n)$$

Harmonic series:

$$\sum_{n=1}^{k} \frac{1}{n} = \ln k + \gamma + \varepsilon_k \leq (\ln k) + 1$$

# 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**

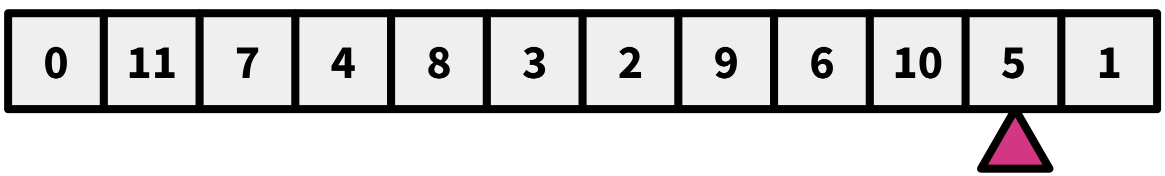
Expected: $O(n\log n)$ Worst-case: $O(n^2)$

Think of this as the adversary chooses the randomness.

# Quicksort vs Mergesort

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

1. **Quicksort requires little additional space**
   Show a smarter manipulation method that requires no extra memory storage




2. **Quicksort exhibits good cache locality**
   It does not have to access locations that are far away frequently

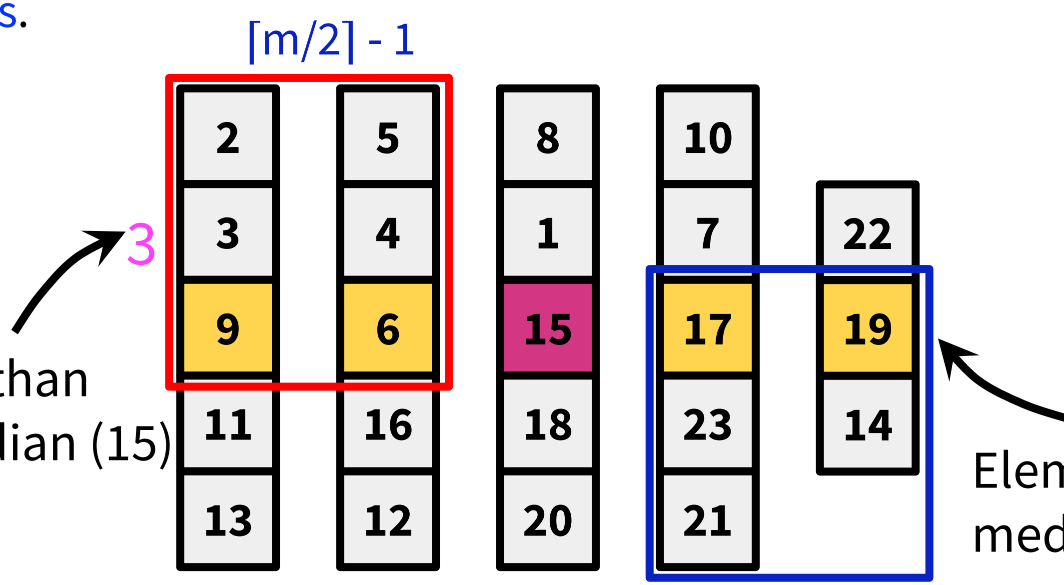
3. **It is easy to avoid worst-case run time of O($n^2$) almost entirely by using randomly selection**
   Verified in many practical applications

# 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.




Elements less than median of median (15)

Elements grater than median of median (15)

$3\cdot(\lceil m/2\rceil - 1) \le |L| \le 7n/10 + 2$

$3\cdot(\lceil m/2\rceil - 1) \le |R| \le 7n/10 + 2$

Complexity of `select_k` is O(n)

# 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.

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

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.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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: (**Lemma 1**) 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).

Proof by contradiction:
If element x is not the majority element for either sub-list, then:
count_left(x) ≤ n_left/2 and count_right(x) ≤ n_right/2;
thus count(x) = count_left(x) + count_right(x) ≤ n_left/2 + n_right/2 = n/2
As a result, x can not possibly be the majority element of the whole list.

# 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:** $O(n\log n)$

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

Count the number of calls to equals.



Explain on board how the algorithm runs using example: 3 2 5 0 2 2 2 2 4 1 2 2 and 0 1 2 3 4 5 2 2 2 2 2 2

# Majority Element

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

**Proof:**

We proceed by induction on i, such that the size of input list is $2^i$

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

Suppose when k=i-1, the `majority_element` is correct for inputs of length $2^{i-1}$. Now consider k=i where the size of input list $2^i$. The algorithm splits the list into two equal-size sub-lists **`A[0:2`**$^{i-1}$**`-1]`** and **`A[2`**$^{i-1}$**`:2`**$^{i}$**`-1]`**. According to Lemma 1, the majority element of the entire array, if it exists, must be the majority element of at least one of **`A[0:2`**$^{i-1}$**`-1]`** or **`A[2`**$^{i-1}$**`:2`**$^{i}$**`-1]`**; otherwise it would occur at most $\lfloor n/2 \rfloor$ times, where n= $2^i$ is the length of the original list. Then the algorithm checks which one of these is the majority element and returns it correctly.

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: $O(\infty)$

# 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(∞)`

The philosophy of trade-off: We trade off between a much better runtime and a much worse runtime, but it will worth the risk if the probability of getting the much better runtime is very high.

# Get Hyped!

The randomized algorithmic paradigm appears everywhere in computer science.

You will see it frequently in the following topics such as graph algorithms!

# QuickSelect

**(Optional Advanced topic)**

# Quickselect

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

You've actually seen it before.

Select the k-th smallest element from a given list.

# Quickselect

```
algorithm quick_select(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 quick_select(L, k)
  else if length(L) < k:
    return quick_select(R, k-length(L)-1)
```

**Runtime:** $O(n^2)$

# Quickselect

```
algorithm quick_select(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 quick_select(L, k)
  else if length(L) < k:
    return quick_select(R, k-length(L)-1)
```

I didn't give you the entire story …

**Runtime:** $O(n^2)$

# Quickselect

```
algorithm quick_select(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 quick_select(L, k)
  else if length(L) < k:
    return quick_select(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`!

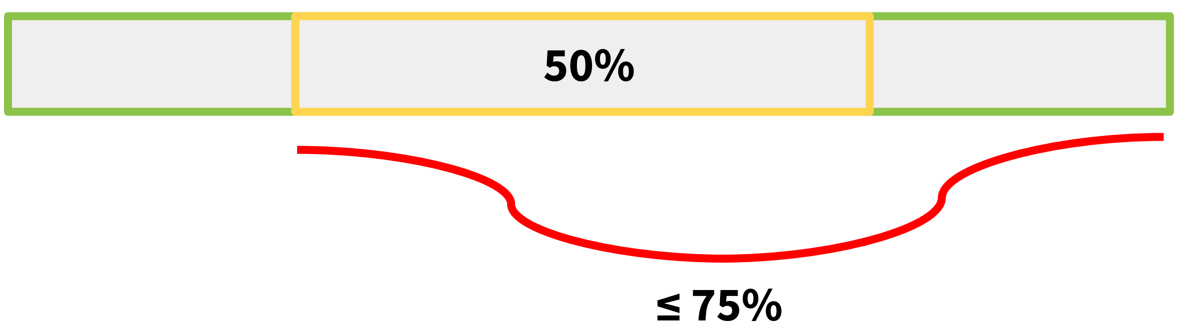
- `select_k` with `smartly_choose_pivot` upper-bounds the length of the list on which it recurses with 7n/10+c.
- Here, let's estimate the expected runtime of shrinking the length of the list to, say, 75% of the original length.

# 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.

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 $\lfloor \log_{4/3} n \rfloor$.

# 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 $[\log_{4/3} n]$.

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:

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

And the expected runtime must be:

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

# Expected Runtime of Quickselect

Simplifying the expression gives …

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

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

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

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

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

# Expected Runtime of Quickselect

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

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 …

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

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

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

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

By the sum of infinite geometric series.

$$= 8cn$$

$$= O(n)$$

# Quickselect

```
algorithm quick_select(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 quick_select(L, k)
  else if length(L) < k:
    return quick_select(R, k-length(L)-1)
```

**Runtime**

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

# Summary

**Randomized Algorithms**

Bogo Sort
Quick Sort
Majority Element
Quick Select

# Summary

**Randomized Algorithms**

Bogo Sort
Quick Sort
Majority Element
Quick Select

Acknowledgement: Part of the materials are adapted from Mary Wootter, Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.