

CSE301 Introduction to Algorithms

Algorithmic Analysis I

Fall 2022



Instructor : Hoon Sung Chwa

Last time

- Course Goals

- What is an algorithm?
- Why we study algorithms?
 - Fundamental, useful, fun
- Goal: **design** and **analysis** of algorithms
- Algorithm designer's questions

- Course Topics

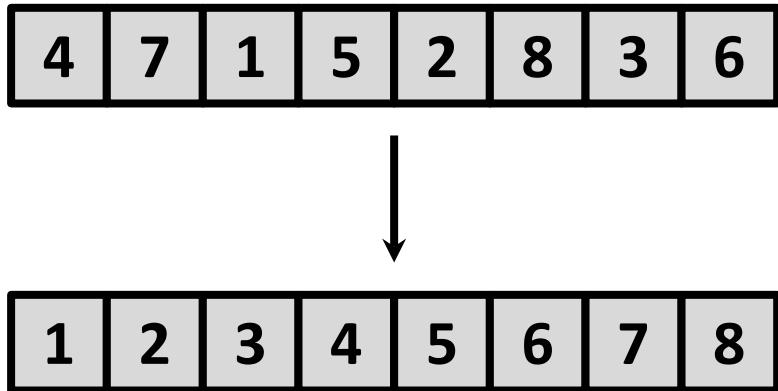
Good Algorithms → ① correctness
 [
 ② efficiency

Outline

- Techniques to analyze correctness and runtime
 - Proving **correctness with induction** **Today!**
 - Proving **runtime with asymptotic analysis** **Next time**
 - *Problems: Comparison-sorting*
 - *Algorithms: Insertion sort*
 - Reading: CLRS 2.1, 2.2, 3

Sorting

- Sorting algorithms order sequences of values.
 - For the sake of clarity, we'll pretend all elements are distinct.



A sorting algorithm

```
def sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

$i=3$
~~5 4 3 2~~ $j=2$
1 2 ~~4~~ 4 $cur=3$

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list. ↳ sorted 된 list를 정렬시키는 방식
- You might have two questions at this point...

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?**

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. Does this actually work?
정작동하는가?
 2. Is it fast? 빠른가?

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

모든 element 를 놓지 않고
정렬할수 있는지
알수 있는가?

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. Does this actually work?
 2. Is it fast?

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!

정렬을 위한 방법



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



Move A[1] leftwards until you find something smaller (or can't go move it any further).

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

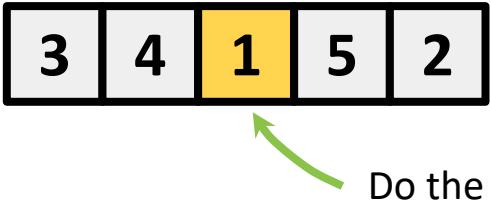
1. Does this actually work? Let's see an example!



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!

1	3	4	5	2
---	---	---	---	---

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



And also for **A[3]** (it's already in the right position).

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!

1	3	4	5	2
---	---	---	---	---

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



And lastly for $A[4]$.

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

1. Does this actually work? Let's see an example!

1	2	3	4	5
---	---	---	---	---

Then we're done!

한번의 Input에서
작동하는 것은 확인.

then How we process
all input

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?**
 2. **Is it fast?**

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?** Ok, obviously it works.
 2. **Is it fast?**

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?** Ok, obviously it works.
 2. **Is it fast?**



But it won't be so obvious later, so let's take some time now to see how to prove that it works rigorously.

How algorithms work

- Algorithms often initialize, modify, or delete new data.
 - Is there a way to prove the algorithm works, without checking it for all (infinitely many) input lists?

How algorithms work

- Algorithms often initialize, modify, or delete new data.
 - Is there a way to prove the algorithm works, without checking it for all (infinitely many) input lists?
- **Key Insight** To reason about the behavior of algorithms, it often helps to look for things that **don't** change. \Rightarrow *loop invariant*

Iteration이 끝에 한걸 같이 유지되는가, 성장이
무엇인가?

Insertion sort

Suppose you have a sorted list,

1	3	4	5
---	---	---	---

Insertion sort

Suppose you have a sorted list,

1	3	4	5
---	---	---	---

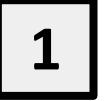
, and another element

2

.

Insertion sort

Suppose you have a sorted list,  , and another element  .

Inserting  immediately to the right of the largest element from the original list that's smaller than  (i.e. right of ) produces another sorted list.

Insertion sort

Suppose you have a sorted list,  , and another element  .

Inserting  immediately to the right of the largest element from the original list that's smaller than  (i.e. right of ) produces another sorted list. Notice that this new list is longer than the original one by one element:  .

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
---	---	---	---	---

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
---	---	---	---	---

The first element, [4], is a sorted list.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
---	---	---	---	---

The first element, [4], is a sorted list. 3 is our other element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
---	---	---	---	---

The first two elements, [3, 4], are a sorted list.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
---	---	---	---	---

The first two elements, [3, 4], are a sorted list. 1 is our other element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

1	3	4	5	2
---	---	---	---	---

The first three elements, [1, 3, 4], are a sorted list.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

1	3	4	5	2
---	---	---	---	---

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

1	3	4	5	2
1	3	4	5	2

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element.
Correctly inserting 5 into the sorted list [1, 3, 4] produces another sorted list [1, 3, 4, 5] that's longer by one element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

1	3	4	5	2
1	3	4	5	2

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element.
Correctly inserting 5 into the sorted list [1, 3, 4] produces another sorted list [1, 3, 4, 5] that's longer by one element.

1	3	4	5	2
---	---	---	---	---

The first four elements, [1, 3, 4, 5], are a sorted list.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

1	3	4	5	2
1	3	4	5	2

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element.
Correctly inserting 5 into the sorted list [1, 3, 4] produces another sorted list [1, 3, 4, 5] that's longer by one element.

1	3	4	5	2
---	---	---	---	---

The first four elements, [1, 3, 4, 5], are a sorted list. 2 is our other element.

Insertion sort

- We can apply this logic at every step.

4	3	1	5	2
3	4	1	5	2

The first element, [4], is a sorted list. 3 is our other element.
Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element.
Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

1	3	4	5	2
1	3	4	5	2

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element.
Correctly inserting 5 into the sorted list [1, 3, 4] produces another sorted list [1, 3, 4, 5] that's longer by one element.

1	3	4	5	2
1	2	3	4	5

The first four elements, [1, 3, 4, 5], are a sorted list. 2 is our other element.
Correctly inserting 2 into the sorted list [1, 3, 4, 5] produces another sorted list [1, 2, 3, 4, 5] that's longer by one element.

Proving Correctness

- There's a name for a condition that is true before and after each iteration of a loop: **a loop invariant**. general하게 항상 유지되는 property를 **loop invariant**라고 한다.

위에서 봤 insertion sort의 loop invariant는?

→ 각 iteration에서 sublist가 (앞 list) 정렬됨.

the sublist $A[0:i]$

Proving Correctness

- There's a name for a condition that is true before and after each iteration of a loop: **a loop invariant**.
 - To prove the correctness of insertion sort, we will use our loop invariant to proceed by **induction**.
 - In this case, our loop invariant (the thing that's not changing) seems to be at the beginning of iteration i (the iteration where we try to insert element $A[i+1]$ into the sorted list), the sublist $A[:i+1]$ is sorted.

mathematics
induction → 틀림없이 정렬된다

Steps

Proving Correctness

- Recall, there are four components in a proof by induction.
 - Inductive Hypothesis** The loop invariant holds after the i th iteration.
 - Base case** The loop invariant holds before the first iteration.
 - Inductive step** If the loop invariant holds after the i th iteration, then it holds after the $(i+1)$ st iteration.
 - Conclusion** If the loop invariant holds after the last iteration, then the algorithm is correct!

이전까지 $P(n)=1$. (1) base case) $n=1$ 일 때 참이고
 (2) (inductive step) $n=i$ 일 때 참이면
 $n=i+1$ 일 때도 참이다.

(3) n 이 증가할 때마다 증명

Proving Correctness

- Loop invariant(i): $A[: i+1]$ is sorted.

Proving Correctness

- Loop invariant(i): $A[: i+1]$ is sorted.
- Formally, for insertion sort...

Proving Correctness

- Loop invariant(i): $A[: i+1]$ is sorted.
- Formally, for insertion sort...
 - **Inductive Hypothesis** The loop invariant(i) holds at the end of iteration i of the outer loop (i.e. $A[: i+1]$ is sorted).

Proving Correctness

- Loop invariant(i): $A[: i+1]$ is sorted.
- Formally, for insertion sort...
 - **Inductive Hypothesis** The **loop invariant(i)** holds at the end of iteration i of the outer loop i.e. $A[: i+1]$ is sorted.
 - **Base case ($i=0$)** The loop invariant(i) holds before the algorithm starts when $i = 0$ i.e. $A[: 1]$ contains only one element, and this is sorted.

Proving Correctness

- Loop invariant(i): $A[: i+1]$ is sorted.
- Formally, for insertion sort...
 - **Inductive Hypothesis** The loop invariant(i) holds at the end of iteration i of the outer loop i.e. $A[: i+1]$ is sorted.
 - **Base case** The loop invariant(i) holds before the algorithm starts when $i = 0$ i.e. $A[: 1]$ contains only one element, and this is sorted.
 - **Inductive step** Recall logic from the animation.

~~If $A[: i]$ is sorted, prove that $A[: i+1]$ is sorted~~

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element. Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

Proving Correctness

- Loop invariant(i): $A[: i+1]$ is sorted.
- Formally, for insertion sort...
 - **Inductive Hypothesis** The loop invariant(i) holds at the end of iteration i of the outer loop i.e. $A[: i+1]$ is sorted.
 - **Base case** The loop invariant(i) holds before the algorithm starts when $i = 0$ i.e. $A[: 1]$ contains only one element, and this is sorted.
 - **Inductive step** Recall logic from the animation. $A[: i]$ is sorted, prove that $A[: i+1]$ is sorted.

3	4	1	5	2
1	3	4	5	2

The first two elements, [3, 4], are a sorted list. 1 is our other element. Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.

- **Conclusion** At the end of the $n-1$ 'st iteration (at the end of the algorithm) $A[: n]$ is sorted. Since $A[: n]$ is the whole list A , so we're done!

Proving Correctness

- Another way to think of proofs by induction for iterative algorithms...
 - **Inductive Hypothesis** The loop invariant holds after the i th iteration.
 - **Base case** The loop invariant holds before the first iteration.
 - “Initialization”
 - **Inductive step** If the loop invariant holds after the i th iteration, then it holds after the $(i+1)$ st iteration.
 - “Maintenance”
 - **Conclusion** If the loop invariant holds after the last iteration, then the algorithm is correct!
 - “Termination”

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?** Ok, obviously it works.
 2. **Is it fast?**

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?** ~~Ok, obviously it works.~~
Yes, and I promise to write a proof by induction if asked to prove correctness formally...
 2. **Is it fast?**

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?** ~~Ok, obviously it works.~~
Yes, and I promise to write a proof by induction if asked to prove correctness formally...
 2. **Is it fast?**

Insertion sort

- **Intuition** Maintain a growing sorted list. For each element, put it into the “right place” in this growing list.
- You might have two questions at this point...
 1. **Does this actually work?** ~~Ok, obviously it works.~~
Yes, and I promise to write a proof by induction if asked to prove correctness formally...
 2. **Is it fast?** Well, what does it mean to be fast?

running time.

Analyzing Runtime

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

At most n inner iters per outer iter

At most n outer iters

Total runtime at most n^2 iters

Outline

- Techniques to analyze correctness and runtime
 - Proving **correctness with induction** **Done!**
 - Skill: analyzing correctness of iterative algorithms
 - Concept: loop invariant, proof by induction
 - Proving **runtime with asymptotic analysis** **Next time!**
 - *Problems: Comparison-sorting*
 - *Algorithms: Insertion sort*
 - Reading: CLRS 2.1, 2.2, 3

Any Question?

CSE301 Introduction to Algorithms

Algorithmic Analysis II

Fall 2022



Instructor : Hoon Sung Chwa

Outline

- Techniques to analyze correctness and runtime
 - ~~Proving correctness with induction~~ Done!
 - Proving runtime with asymptotic analysis
 - *Problems: Comparison-sorting*
 - *Algorithms: Insertion sort*
 - Reading: CLRS 2.1, 2.2, 3

알고리즘이다
running time이라도.
input 따라 달라진다

How do we measure the runtime of an algorithm?

Runtime Analysis

We have 3 ways.

Runtime
실행방법

- We might care about the runtime of an algorithm in a few cases.

~~Focus~~ **Worst-case analysis** What is the runtime of the algorithm on the worst possible input? *like upper bound*

■ We'll focus on this type of analysis since it tells us that an algorithm performs at least this fast for *every* input.

- **Best-case analysis** What is the runtime of the algorithm on the best possible input?
- **Average-case analysis** What is the runtime of the algorithm on the average input?

Asymptotic Analysis

어디까지 측정할까?

- What does it mean to measure “runtime” of an algorithm?
 - Engineers probably care most about the “real-world time”: how long does the algorithm take in seconds, minutes, hours, days, etc.? → 실제로 걸려온다
 - This heavily depends on computer hardware, programming language, etc. 컴퓨터성능, 언어의 영향 받음.
 - While important, it will not be the emphasis of this course.
 - Instead, we want to use a universal measure of runtime that's independent of these considerations.

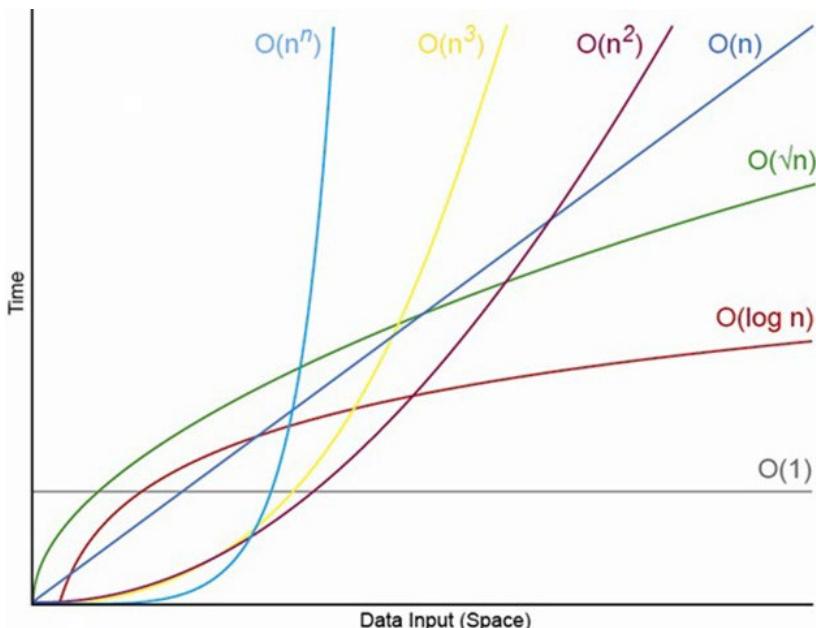
We

\Rightarrow 상수한 알고리즘 만을 기준으로 비교하고 싶다

Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).

One algorithm is “faster” than another if its runtime scales better with the size of the input.



Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).

One algorithm is “faster” than another if its runtime scales better with the size of the input.

Input size에 따른 scale을 비교한다.

- **Pros**
 - Abstracts away from hardware- and language- specific issues
 - Make algorithm analysis much more tractable
- **Cons**
 - Only makes sense if n is large (compared to the constant factors).



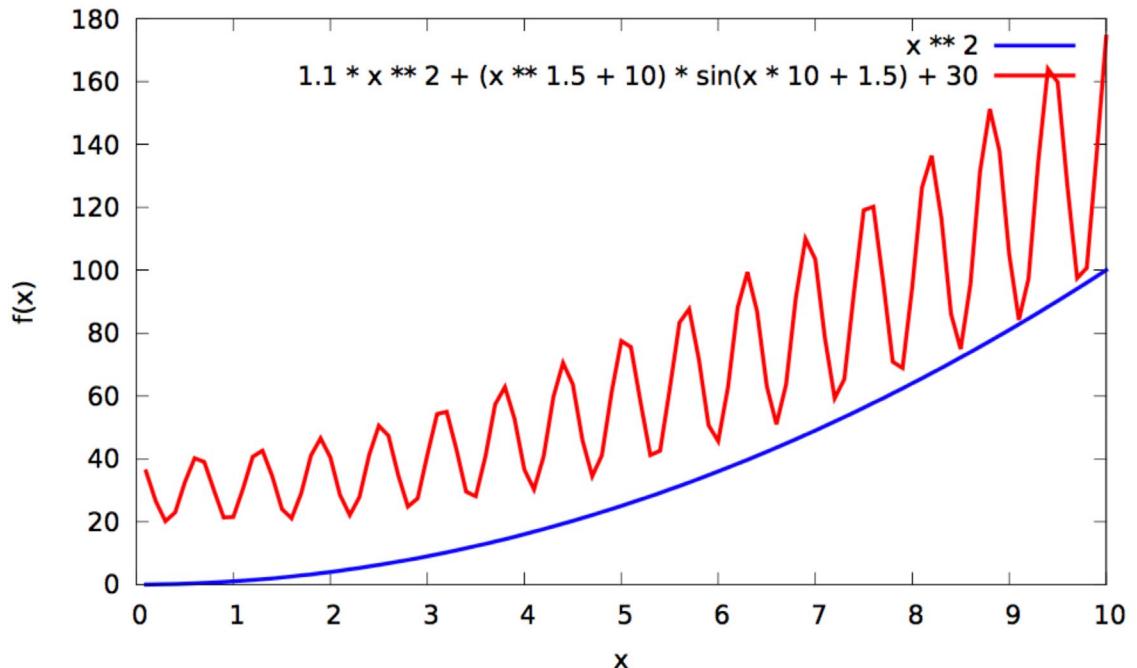
9,999,999,999,999 n
is “better” than n^2 ?!?!?

Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).
 - Are the following functions similar or not?

$$f_1(x) = x^2$$

$$f_2(x) = 1.1x^2 + (x^{1.9} + 10) \sin(10x + 1.5) + 30$$



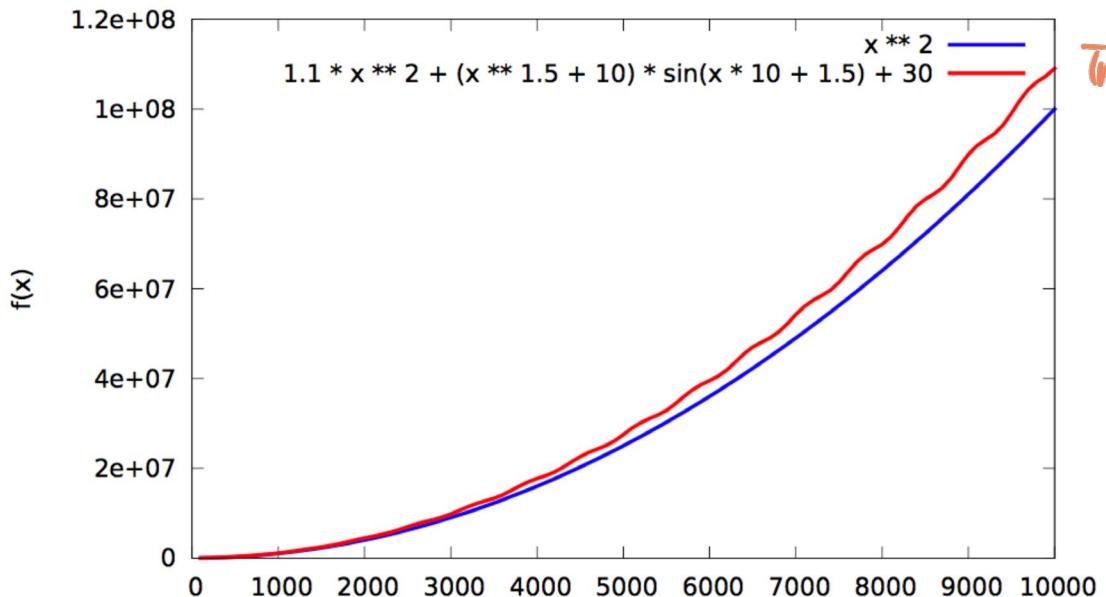
Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).
 - Are the following functions similar or not?

$$f_1(x) = x^2$$

$$f_2(x) = 1.1x^2 + (x^{1.9} + 10) \sin(10x + 1.5) + 30$$

Ignore lower-order terms



Input이 커지면
두 그래프는
더욱 차이난다.

Informally, it can be determined by ignoring constants and non-dominant growth terms.

Running times for $f_1(x)$ and $f_2(x)$ are both n^2

Asymptotic analysis
Big-O term

($O()$, $\Omega()$, $\Theta()$)
lower bound both (up and low) bound

Big-O ($O(\dots)$) Means Upper-Bound

- Big-O notation is a mathematical notation for upper-bounding a function's rate of growth.

Big-O Notation

- Let $T(n)$, $g(n)$ be functions of positive integers.
 - You can think of $T(n)$ as being a runtime: positive and increasing as a function of n .
- We say " $T(n)$ is $O(g(n))$ " if $g(n)$ grows at least as fast as $T(n)$ as n gets large.
- Formally,

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

iff

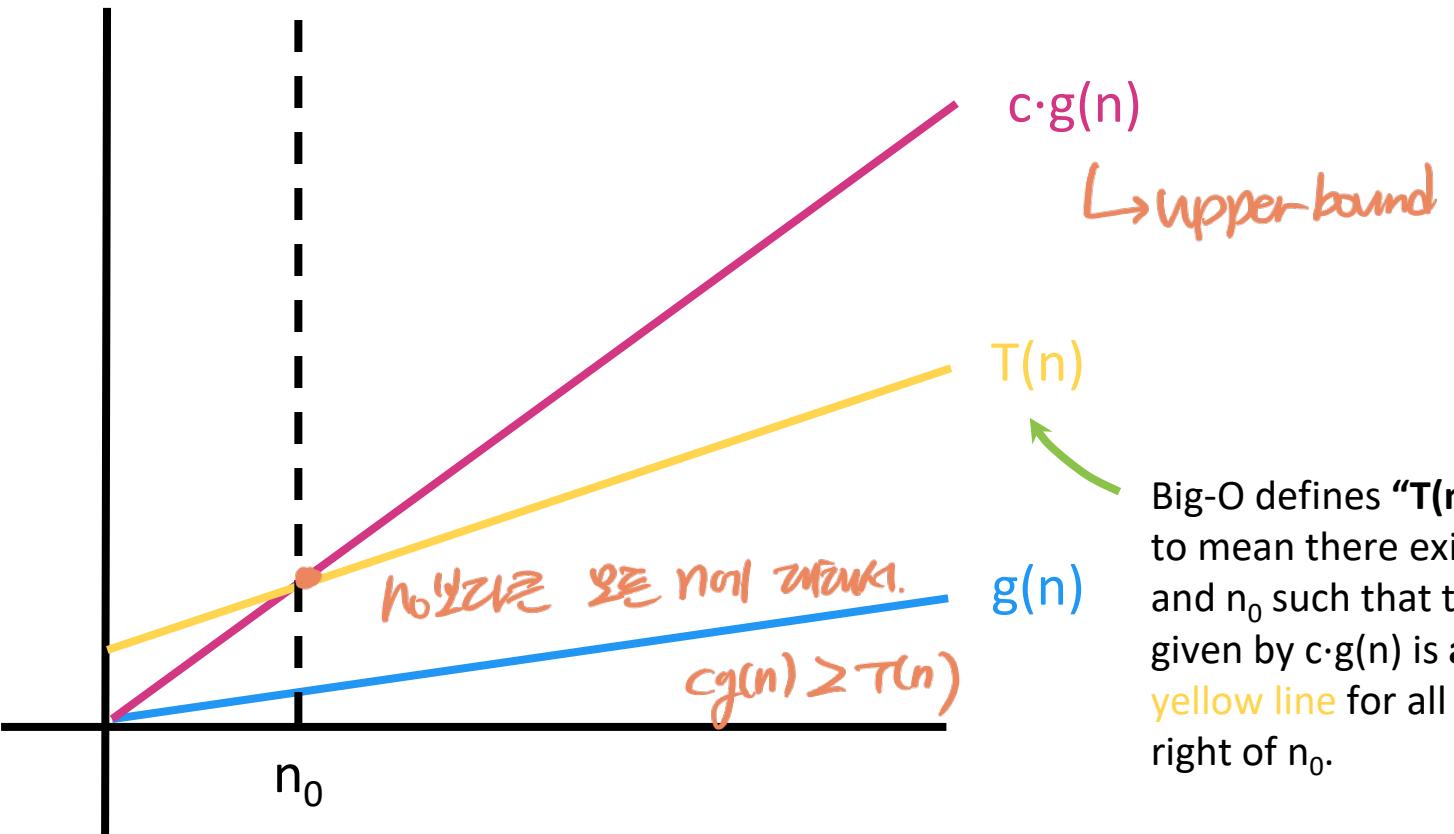
$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$\underline{0 \leq T(n) \leq c \cdot g(n)} \text{ 이 조건 만족}$$

$T(n) = O(g(n))$
 iff

 $\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$
 $0 \leq T(n) \leq c \cdot g(n)$

- Graphically,



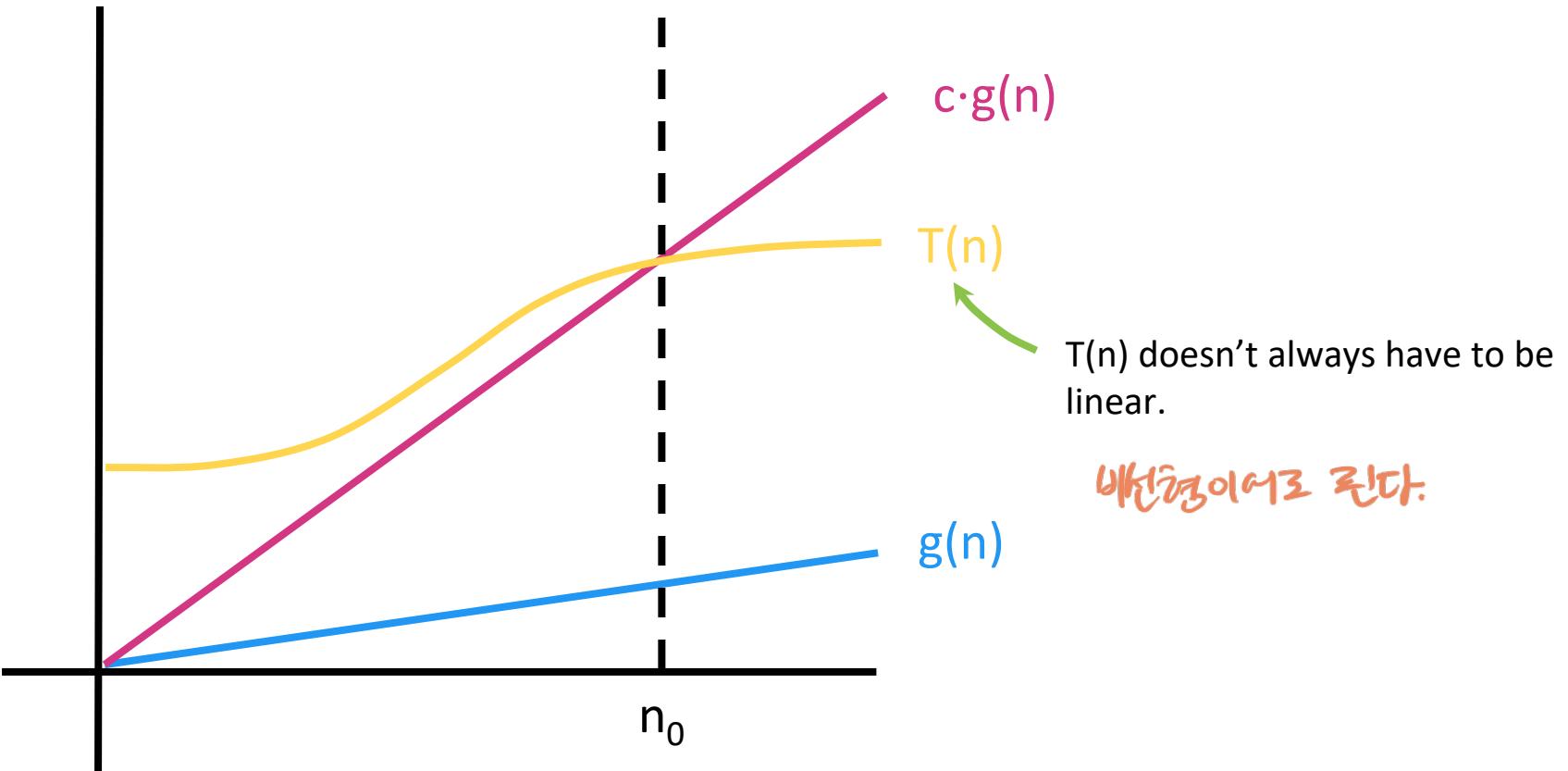
Big-O defines " $T(n) = O(g(n))$ " to mean there exists some c and n_0 such that the pink line given by $c \cdot g(n)$ is **above** the yellow line for all values to the right of n_0 .

Big-O Notation

$T(n) = O(g(n))$
 iff

 $\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$
 $0 \leq T(n) \leq c \cdot g(n)$

- Graphically,

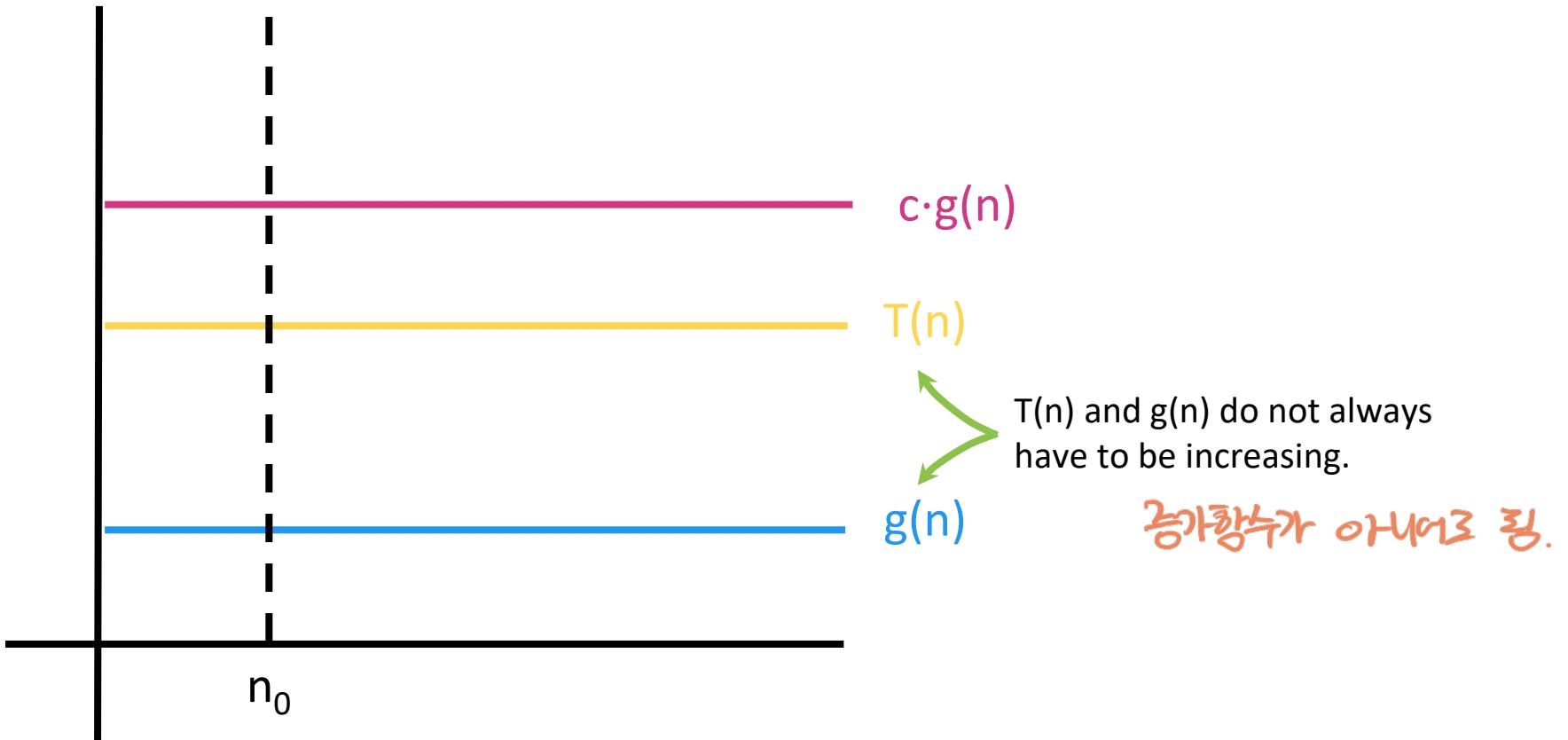


Big-O Notation

$T(n) = O(g(n))$
 iff

 $\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$
 $0 \leq T(n) \leq c \cdot g(n)$

- Graphically,



Big-O Notation

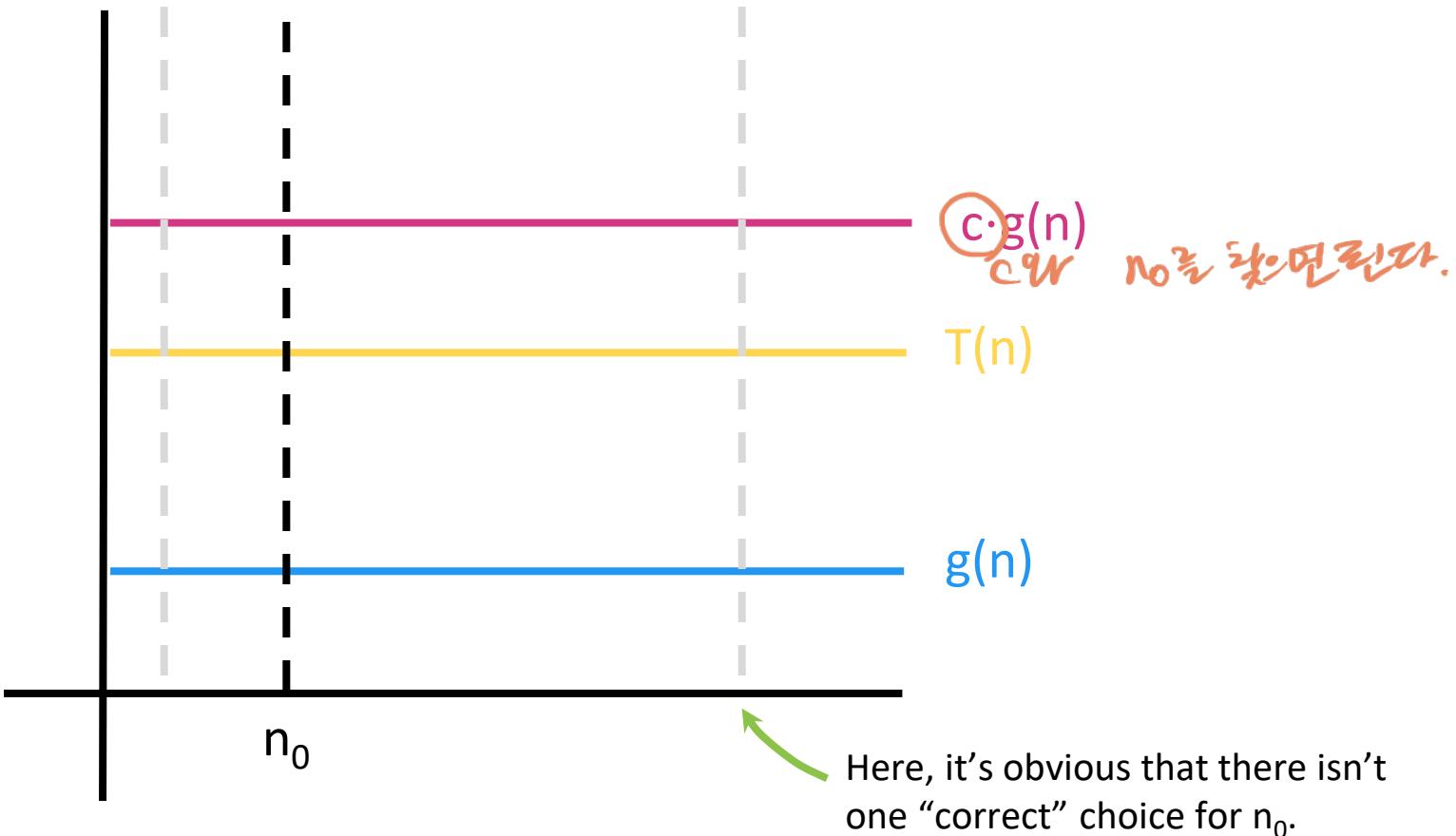
$$T(n) = O(g(n))$$

iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$

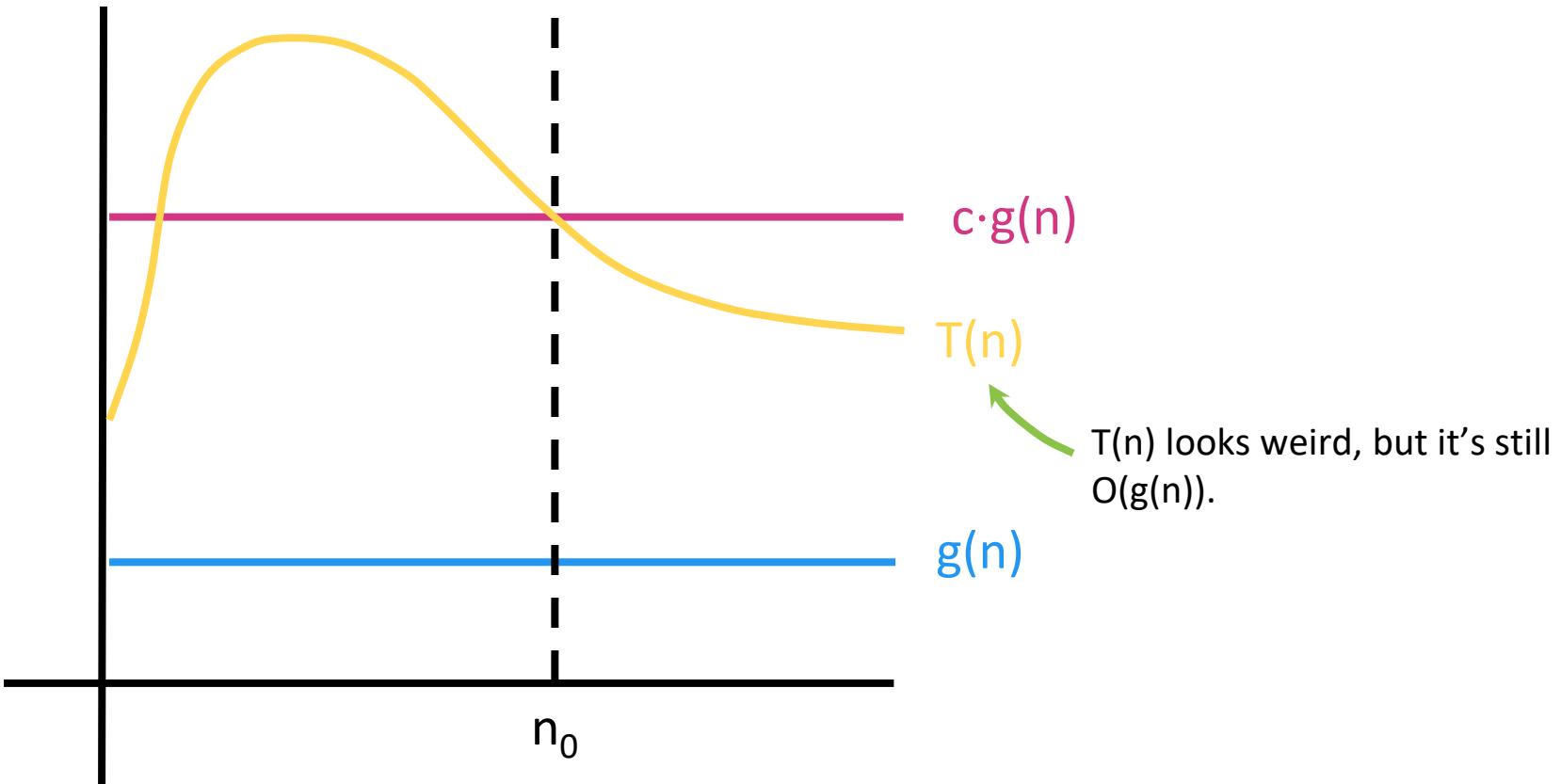
- Graphically,



Big-O Notation

$T(n) = O(g(n))$
iff $\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$
 $0 \leq T(n) \leq c \cdot g(n)$

- Graphically,



$T(n) = O(g(n))$
iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,
 $0 \leq T(n) \leq c \cdot g(n)$

- To prove $T(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

$$T(n) \leq g(n)$$
$$n_0 > 0, n \geq n_0$$

$T(n) = O(g(n))$
 iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot g(n)$$

- To prove $T(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

- For example,

Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.

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

\Leftrightarrow prove $\exists c, n_0$ s.t. $\forall n \geq n_0, 0 \leq n \leq c \cdot n \log n$

assign $c = 1, n_0 = 2$

$n \leq n \log n \quad \forall n \geq 2$

($\because \log n \geq 1 \text{ for } n \geq 2$)

$T(n) = O(g(n))$
 iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot g(n)$$

- To prove $T(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

- For example,

Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.

- To prove $T(n) \neq O(g(n))$, proceed by contradiction.

극류법

Big-O Notation

$T(n) = O(g(n))$
 iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot g(n)$$

- To prove $T(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

- For example,

Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since n is positive and $1 \leq \log(n)$ for $n \geq 2$.

- To prove $T(n) \neq O(g(n))$, proceed by contradiction.

- For example,

Suppose $T(n) = n^2$ and $g(n) = n$. We prove that $T(n) \neq O(g(n))$.

Suppose there exists some c and n_0 such that for all $n \geq n_0$, $n^2 \leq c \cdot n$.

Consider $n = \max\{c, n_0\} + 1$. By construction, we have both $n \geq n_0$ and $n > c$, which implies that $n^2 > c \cdot n$.

Prove $n^2 \neq O(n)$ suppose $\exists c, n_0 \text{ st } \forall n \geq n_0, \underline{n^2 \leq c \cdot n}$

$n = \max(c, n_0) + 1$

By (def) $n > n_0, n > c \rightarrow n^2 > cn_0 \quad \underline{n^2 > cn_0 \geq cn_0}$

contradiction

Big-O Notation

$T(n) = O(g(n))$
 iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot g(n)$$

- To prove $T(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

- For example,**

Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since n is positive and $1 \leq \log(n)$ for $n \geq 2$.

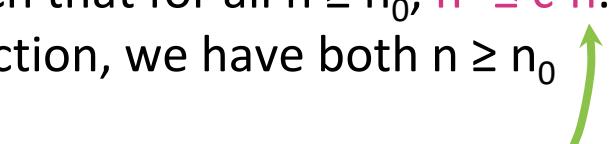
- To prove $T(n) \neq O(g(n))$, proceed by contradiction.

- For example,**

Suppose $T(n) = n^2$ and $g(n) = n$. We prove that $T(n) \neq O(g(n))$.

Suppose there exists some c and n_0 such that for all $n \geq n_0$, $n^2 \leq c \cdot n$.

Consider $n = \max\{c, n_0\} + 1$. By construction, we have both $n \geq n_0$ and $n > c$, which implies that $n^2 > c \cdot n$.



Here's the contradiction:
 assuming $n^2 \leq c \cdot n$ implies
 $n^2 > c \cdot n$ (the opposite)

Big-O Notation

Ω

Big- Ω Means Lower-Bound



- Big- Ω notation is a mathematical notation for **lower-bounding** a function's rate of growth.
 - Informally, it can be determined by ignoring constants and non-dominant growth terms.

Big- Ω Notation

- Let $T(n)$, $g(n)$ be functions of positive integers.
 - You can think of $T(n)$ as being a runtime: positive and increasing as a function of n .
- We say " $T(n)$ is $\Omega(g(n))$ " if $g(n)$ grows at most as fast as $T(n)$ as n gets large.
- Formally,

$$T(n) = \Omega(g(n))$$

iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!

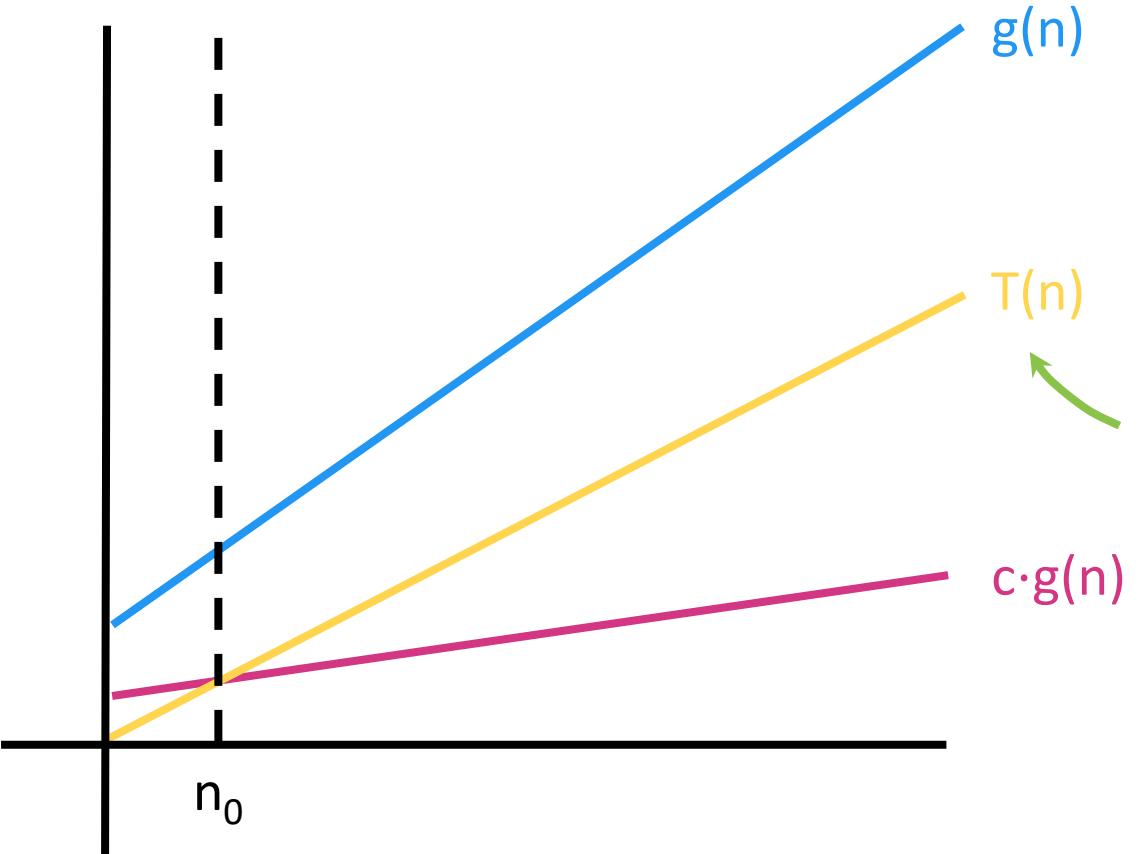
Big- Ω Notation

$$T(n) = \Omega(g(n))$$

iff

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \\ 0 \leq c \cdot g(n) \leq T(n)$$

- Graphically,



Big- Ω defines " $T(n) = \Omega(g(n))$ " to mean there exists some c and n_0 such that the pink line given by $c \cdot g(n)$ is **below** the yellow line for all values to the right of n_0 .

Big-Θ Means Upper and Lower-Bound

- We say “ $T(n)$ is $\Theta(g(n))$ ” iff

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

AND

$T(n)$ is $\Omega(g(n))$

) 듀리만족
tight한 case

More examples

- Claim I:

- Prove that $\frac{3}{2}n^2 + \frac{5}{2}n - 3 = O(n^2)$
 - Equivalent: find c and n_0 such that $\frac{3}{2}n^2 + \frac{5}{2}n - 3 \leq cn^2$
 - Example proof

$$\frac{3}{2}n^2 + \frac{5}{2}n - 3 < \frac{3}{2}n^2 + 5n^2 + n^2 = \frac{15}{2}n^2 \quad \forall n \geq 1$$

$$C = \frac{15}{2}, \quad n_0 = 1$$

More examples

- Claim II:

- Prove that $\frac{1}{2}n^2 - 3n = \underline{\Theta(n^2)}$

- Equivalent: find c_1, c_2 and n_0 such that $c_1 n^2 \leq \underline{\frac{1}{2}n^2 - 3n} \leq \overline{c_2 n^2}$
- Example proof

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$\frac{1}{2} - \frac{3}{n} \leq c_2 \rightarrow n \geq 1, c_2 \geq \frac{1}{2}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \rightarrow n \geq 7, c_1 \leq \frac{1}{14}$$

$$c_1 = \frac{1}{14}, c_2 = \frac{1}{2}, n_0 = 7$$

More examples

- Claim III:

- Let $f(n) = 5n + 12$. Which of the following statements are true?

- | | | |
|-------------------------|---|-------------------|
| 1. $f(n) = O(n)$ | T | tight upper bound |
| 2. $f(n) = O(n \log n)$ | T | |
| 3. $f(n) = O(n^2)$ | T | |
| 4. $f(n) = O(2^n)$ | T | |

Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
 - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
 - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
 - **Average-case analysis** What is the runtime of the algorithm on the average input?

Worst-Case Analysis

- What is the worst possible input for insertion sort?
 - Notice it's possible for the inner **while** loop to iterate anywhere between 1 and i times. What if it iterated i times every single time? What input causes this pattern?

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Worst-Case Analysis

- What is the worst possible input for insertion sort?
 - Notice it's possible for the inner **while** loop to iterate anywhere between 1 and i times. What if it iterated i times every single time? What input causes this pattern?



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
 - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
 - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
 - **Average-case analysis** What is the runtime of the algorithm on the average input?

Best-Case Analysis

- What is the best possible input for insertion sort?
 - Notice it's possible for the inner **while** loop to iterate anywhere between 1 and i times. What if it iterated 1 time every single time? What input causes this pattern?

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Best-Case Analysis

- What is the best possible input for insertion sort?
 - Notice it's possible for the inner **while** loop to iterate anywhere between 1 and i times. What if it iterated 1 time every single time? What input causes this pattern?



```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Worst-Case vs. Best-Case Analysis

- We might care about the runtime of an algorithm in a few cases.
 - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
 - The worst-case runtime of insertion sort is $O(n^2)$.
 - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
 - The best-case runtime of insertion sort is $\Omega(n)$.
 - **Average-case analysis** What is the runtime of the algorithm on the average input?

Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
 - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
 - The worst-case runtime of insertion sort is $O(n^2)$.
 - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
 - The best-case runtime of insertion sort is $\Omega(n)$.
 - **Average-case analysis** What is the runtime of the algorithm on the average input?
 - We will learn this type of analysis when we cover Randomized Algorithms!

Analyzing Runtime

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Upper-bound for worst-case runtime $O(n^2)$

Analyzing Runtime

```
def insertion_sort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

Lower-bound for best-case runtime $\Omega(n)$

Outline

- Techniques to analyze correctness and runtime
 - Proving **correctness with induction**
 - Analyzing correctness of iterative algorithms
 - ✓ Loop invariant, proof by induction
 - Proving **runtime with asymptotic analysis**
 - Insertion sort
 - ✓ Worst-case analysis $O(n^2)$
 - ✓ Best-case analysis $\Omega(n)$
 - *Problems: Comparison-sorting*
 - *Algorithms: Insertion sort*
 - Reading: CLRS 2.1, 2.2, 3

CSE301 Introduction to Algorithms

Divide and Conquer I

Fall 2022

CLIMLLIIS



Instructor : Hoon Sung Chwa

Course Overview

- Algorithmic Analysis
- **Divide and Conquer**
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Divide and Conquer I
 - Proving correctness with induction
 - Proving runtime with **recurrence relations**
 - How do we measure the runtime of a recursive algorithm?
 - Proving the **Master method**
 - A useful theorem so we do not have to answer this question from scratch each time
 - Learn the **Substitution method**
 - It can be used when the master method doesn't work
 - *Problems: Comparison-sorting*
 - *Algorithms: Mergesort*
 - Reading: CLRS 2.3, 4.3-4.6

Recall

- Insertion sort
 - Does this actually work? Yes!
 - We talked about loop invariants and proofs by induction.
 - Is it fast? Eh, nah.
 - We talked about worst-case, best-case, and average case analysis.
 - We talked about Big-O, Big- Ω and Big- Θ notation to describe upper-bounds, lower-bounds, and tight-bounds.
 - Upper-bound for worst-case runtime $O(n^2)$
 - Lower-bound for best-case runtime $\Omega(n)$

Another way of thinking

- Can we do better than insertion sort?
- Mergesort uses divide-and-conquer.

$O(n^2)$

→ sorted list \rightarrow 정렬된 목록

- Does this actually work?
 - We will revisit proofs by induction.
- Is it fast?
 - We will talk about recurrence relations

recursive algorithm

Another way of thinking

- Can we do better than insertion sort?
- Mergesort uses divide-and-conquer.

- **Does this actually work?**

- We will revisit proofs by induction.

- **Is it fast?**

- We will talk about recurrence relations.

These are the same questions we asked about insertion sort!

Mergesort

Divide and Conquer



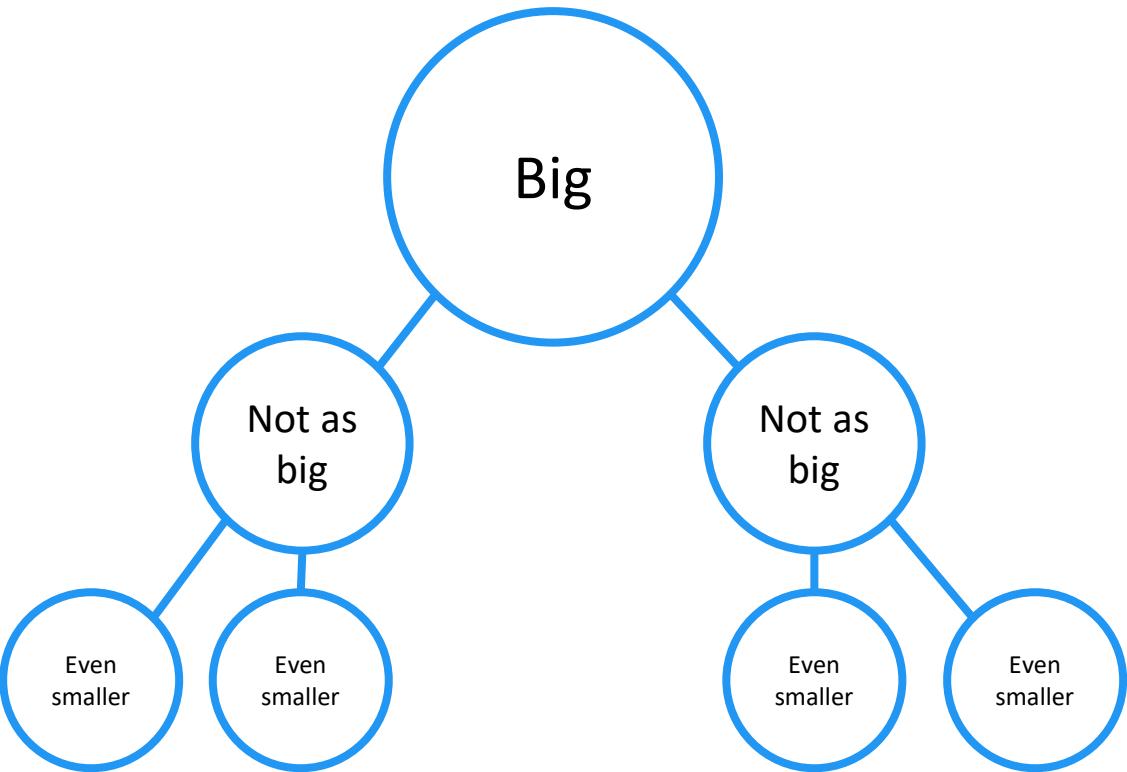
Divide Break the current problem into smaller (easier) sub-problems.



Conquer Solve the smaller problems and collect the results to solve the current problem.



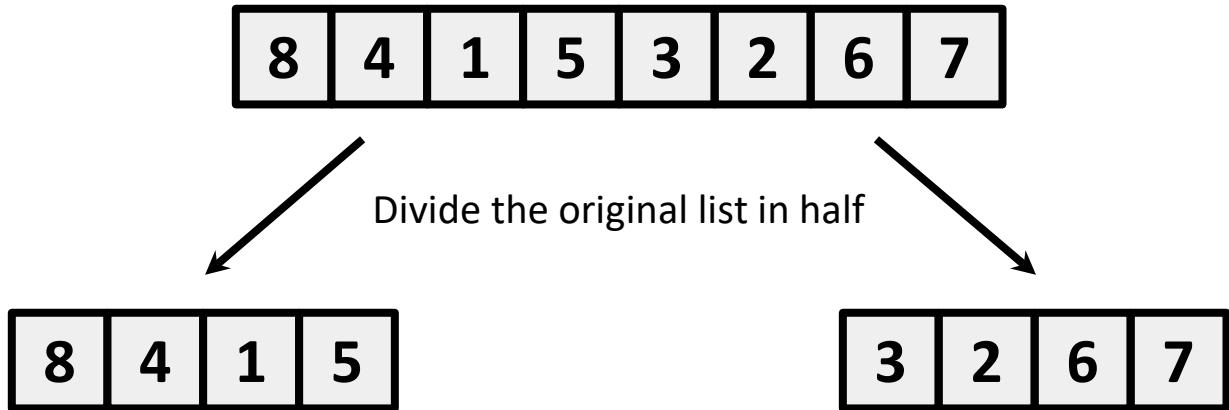
Combine combine solutions to make a solution for the original problem



Mergesort

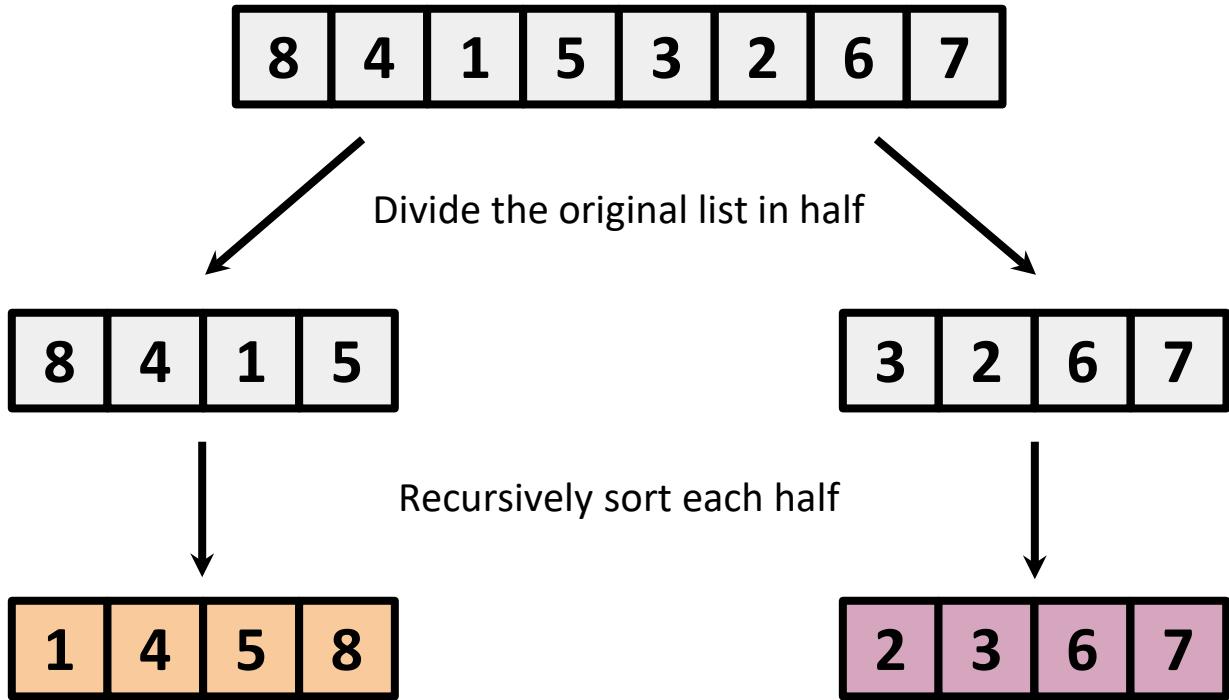
8	4	1	5	3	2	6	7
---	---	---	---	---	---	---	---

Mergesort

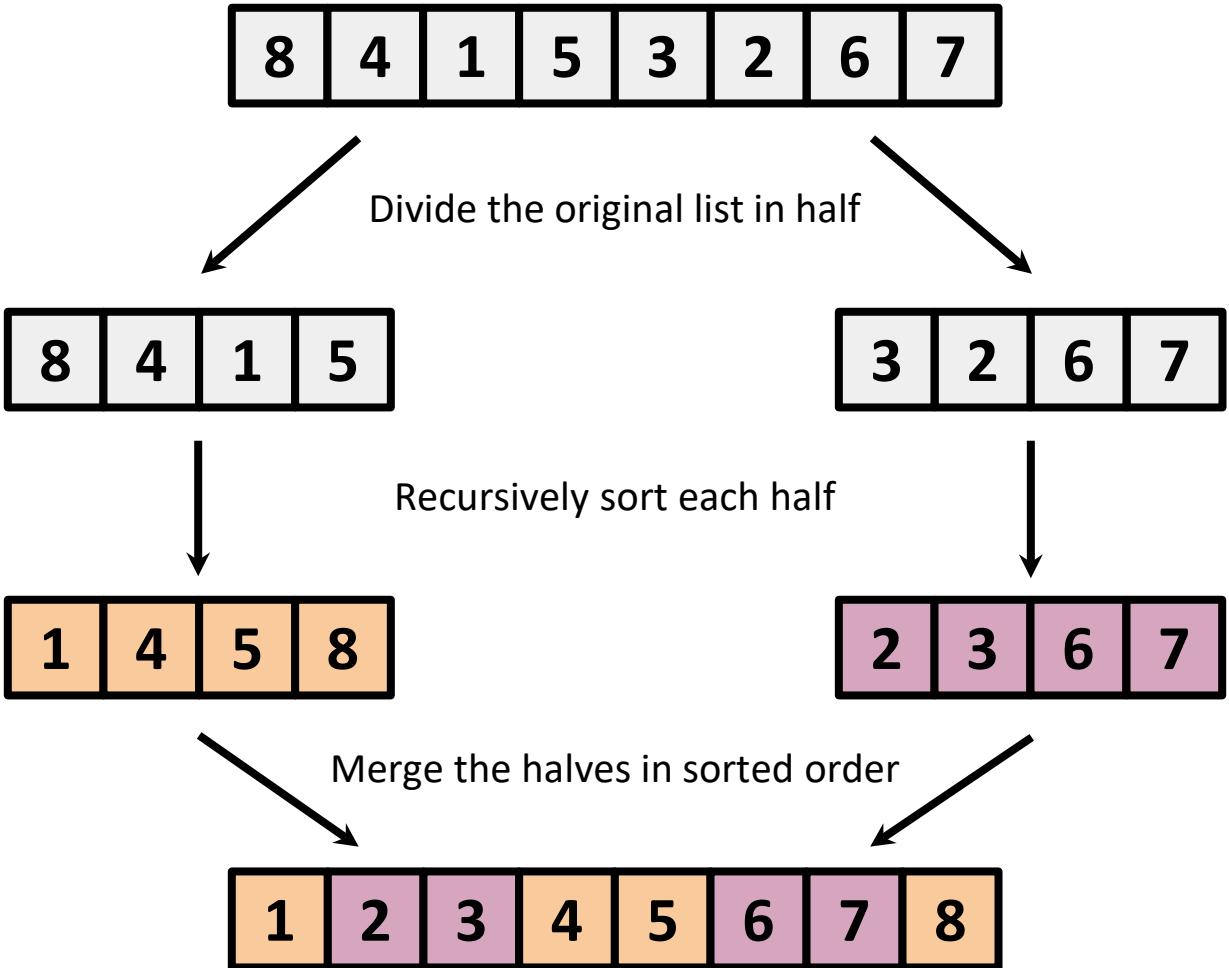


Divide the original list in half

Mergesort



Mergesort



Mergesort

```
def mergesort(A):
```

Mergesort

```
def mergesort(A):  
    if len(A) <= 1:  
        return A
```

Mergesort

```
def mergesort(A):
    if len(A) <= 1:
        return A
    L = mergesort(A[0:n/2])
```

Mergesort

```
def mergesort(A):
    if len(A) <= 1:
        return A
    L = mergesort(A[0:n/2])
    R = mergesort(A[n/2:n])
```

Mergesort

```
def mergesort(A):
    if len(A) <= 1:           ← Conquer
        return A
    L = mergesort(A[0:n/2])   ) Divide
    R = mergesort(A[n/2:n])
    return merge(L, R)       ← combine.
```

Mergesort

```
def merge(L, R):
```

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
```

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
```

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
```

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
        else:
            result.append(R[r_idx])
            r_idx += 1
```

Mergesort

Sorted list 가짐.

```
def merge(L, R): → 두개의 sorted된 결과를 하나로 합치는 과정
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
        else:
            result.append(R[r_idx])
            r_idx += 1
    result.extend(L[l_idx:len(L)])
    result.extend(R[r_idx:len(R)])
```

→ 마지막 값 넣기

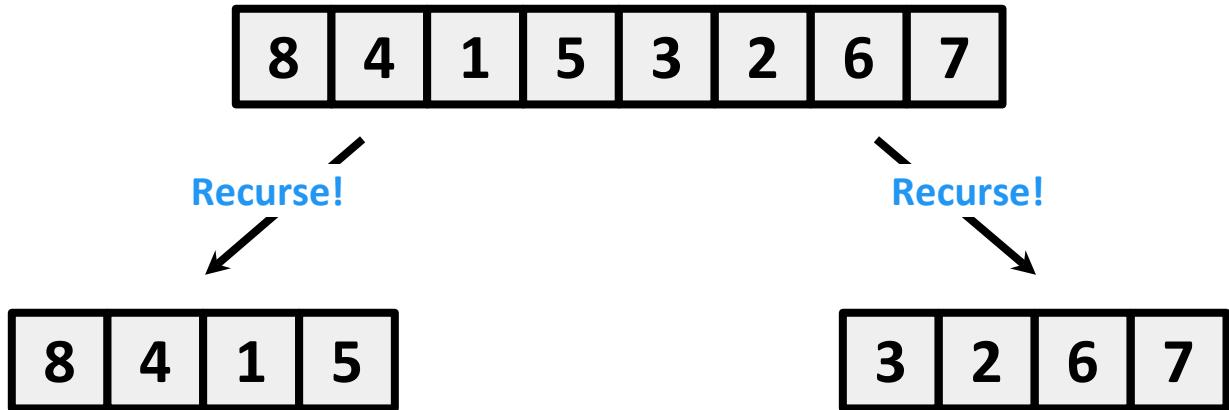
Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
        else:
            result.append(R[r_idx])
            r_idx += 1
    result.extend(L[l_idx:len(L)])
    result.extend(R[r_idx:len(R)])
    return result
```

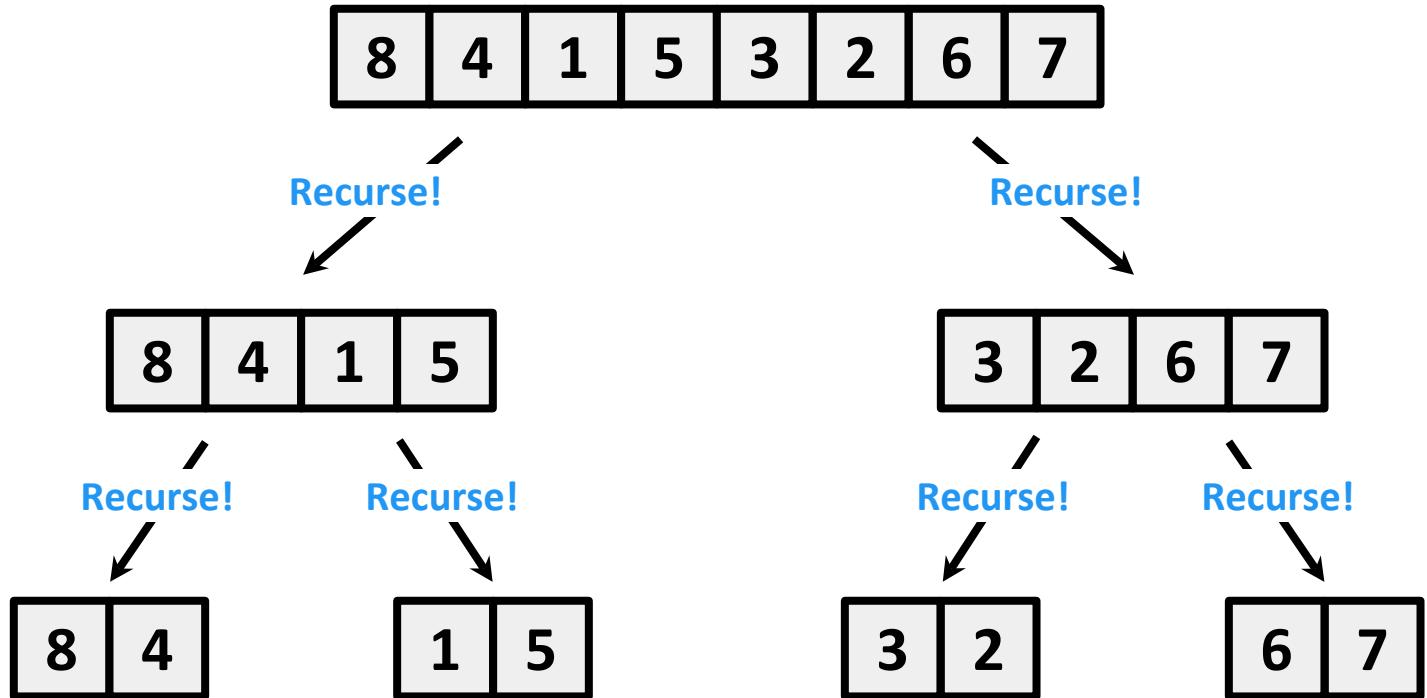
Mergesort

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

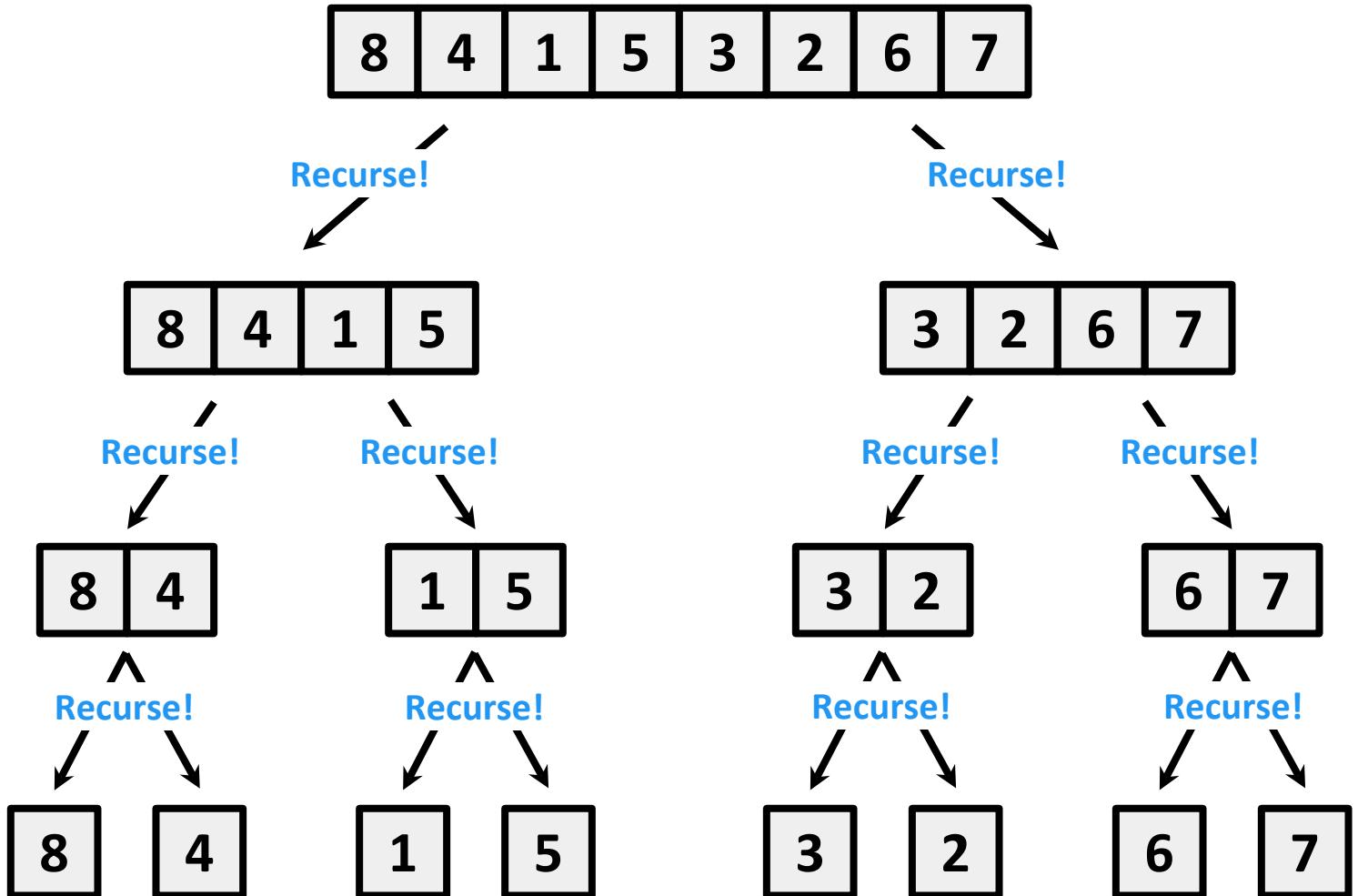
Mergesort



Mergesort



Mergesort



Mergesort

8

4

1

5

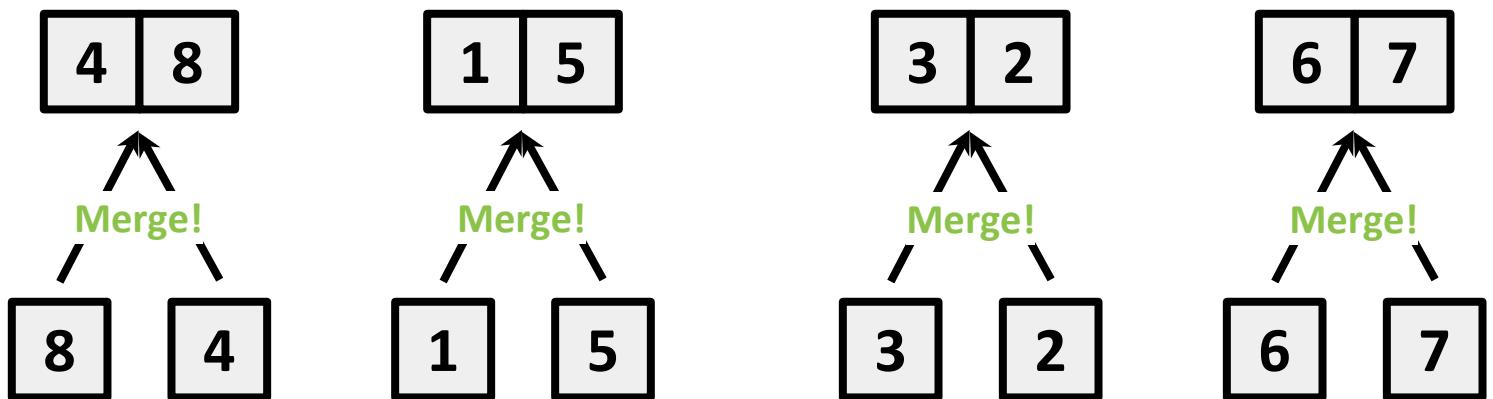
3

2

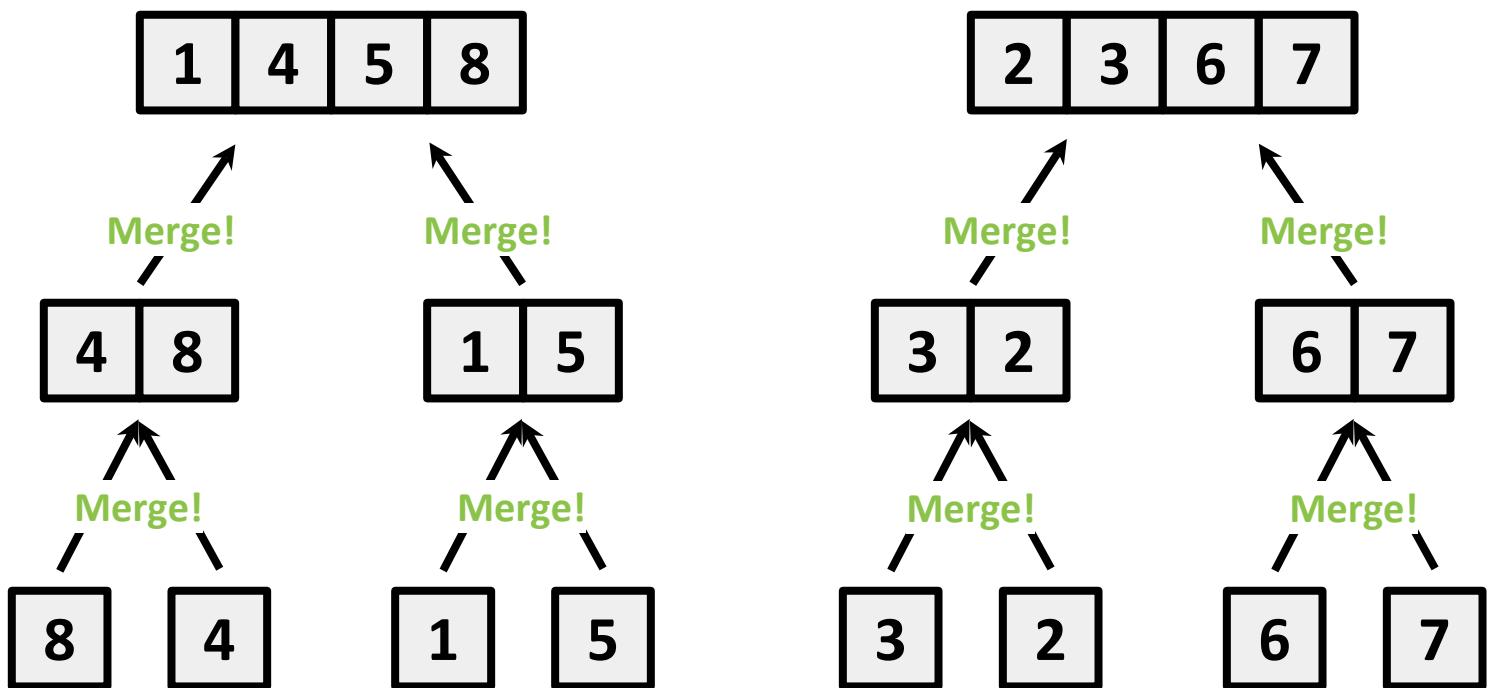
6

7

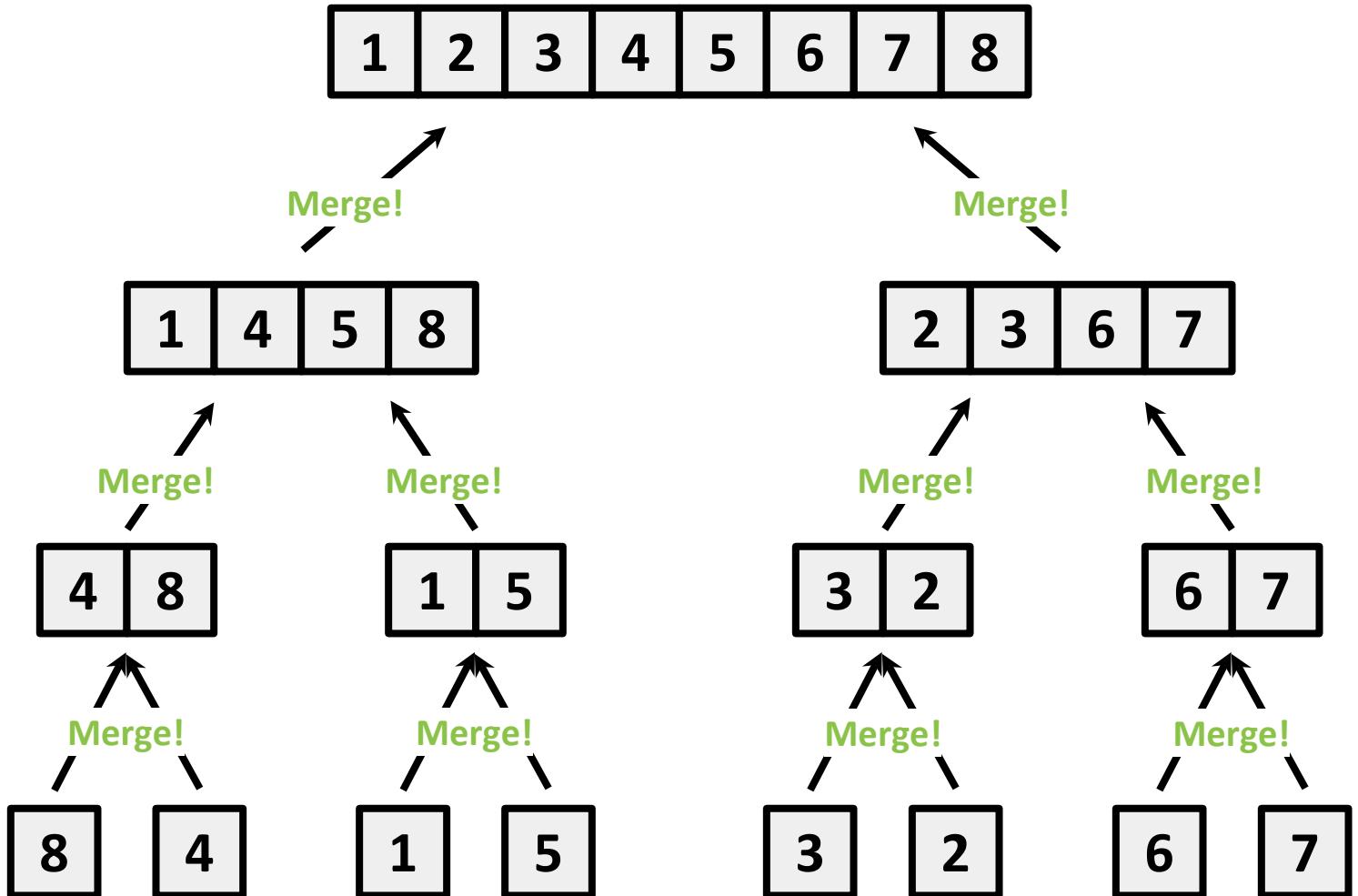
Mergesort



Mergesort



Mergesort



Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point...
 1. **Does this actually work?**
 2. **Is it fast?**

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point...
 1. **Does this actually work?**
 2. **Is it fast?**

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Mergesort

1. Does this actually work? We've already seen an example!

- Formally, similar to last time, we proceed by induction. However, rather than inducting on the loop iteration, we induct on the length of the input list.

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

loop invariant \rightarrow iteration
recursion invariant \rightarrow input length

Proving Correctness

- Recall, there are four components in a proof by induction.
 - **Inductive Hypothesis** The algorithm works on input lists of length 1 to i.
 - **Base case** The algorithm works on input lists of length 1.
 - **Inductive step** If the algorithm works on input lists of length 1 to i, then it works on input lists of length i+1.
 - **Conclusion** If the algorithm works on input lists of length n, then it works on the entire list.

Proving Correctness

- Formally, for `mergesort`...
 - **Inductive Hypothesis** `mergesort` correctly sorts input lists of length i .
(In every recursive call, `mergesort` returns a sorted list.)
 - **Base case** ($i=1$) `mergesort` correctly sorts input lists of length 1; it returns a 1-element list, which is trivially sorted.
 - **Inductive step** Suppose the algorithm works on input lists of length 1 to i . Calling `mergesort` on an input list of length $i+1$ recursively calls `mergesort` on the left and right halves, which have lengths between 1 and i ; therefore, `left` and `right` contain the elements originally in the left and right halves of the list, but sorted. Given two sorted lists, `merge` returns a single sorted list with all of the elements from the original two lists.
- **Conclusion** The inductive hypothesis holds for all i . In particular, given an input list of any length n , `mergesort` returns a sorted version of that list!

Proving Correctness

- Formally, for **mergesort**...

- **Inductive Hypothesis** **mergesort** correctly sorts input lists of length i .
- **Base case** **mergesort** correctly sorts input lists of length 1; it returns a 1-element list, which is trivially sorted.
- **Inductive step** Suppose the algorithm works on input lists of length 1 to i . Calling **mergesort** on an input list of length $i+1$ recursively calls **mergesort** on the left and right halves, which have lengths between 1 and i ; therefore, **left** and **right** contain the elements originally in the left and right halves of the list, but sorted. Given two sorted lists, **merge** returns a single sorted list with all of the elements from the original two lists.

Merge (loop invariant)

- result sorted list!
- right index smallest one ok!



Proving this statement requires another proof by induction, with a loop invariant!

- **Conclusion** The inductive hypothesis holds for all i . In particular, given an input list of any length n , **mergesort** returns a sorted version of that list!

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
        else:
            result.append(R[r_idx])
            r_idx += 1
    result.extend(L[l_idx:len(L)])
    result.extend(R[r_idx:len(R)])
    return result
```

Proving Correctness

```
def proof_of_correctness_helper(algorithm):
```

Proving Correctness

```
def proof_of_correctness_helper(algorithm):  
    if algorithm.type == "iterative":  
        # 1) Find the loop invariant  
        # 2) Define the inductive hypothesis  
        #     (internal state at iteration i)  
        # 3) Prove the base case (i=0)  
        # 4) Prove the inductive step (i => i+1)  
        # 5) Prove the conclusion (i=n => correct)
```

Proving Correctness

```
def proof_of_correctness_helper(algorithm):  
    if algorithm.type == "iterative":  
        # 1) Find the loop invariant  
        # 2) Define the inductive hypothesis  
        #     (internal state at iteration i)  
        # 3) Prove the base case (i=0)  
        # 4) Prove the inductive step (i => i+1)  
        # 5) Prove the conclusion (i=n => correct)  
  
    elif algorithm.type == "recursive":  
        # 1) Define the inductive hypothesis  
        #     (correct for inputs of sizes 1 to i)  
        # 2) Prove the base case (i < small constant)  
        # 3) Prove the inductive step (i => i+1 OR  
        #     {1,2,...,i} => i+1)  
        # 4) Prove the conclusion (i=n => correct)
```

Today's Outline

- Divide and Conquer I
 - ~~Proving correctness with induction~~ Done!
 - Proving runtime with recurrence relations
 - Proving the Master method
 - *Problems: Comparison-sorting*
 - *Algorithms: Mergesort*
 - Reading: CLRS 2.3, 4.3-4.6

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point...
 1. **Does this actually work?**
 2. **Is it fast?**

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point...
 1. Does this actually work? Yes!
 2. Is it fast?

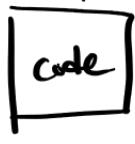
```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point...
 1. Does this actually work? Yes!
 2. Is it fast?

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Asymptotic Analysis

 $T(n) \rightarrow O(?)$
implementation only analysis.

Analyzing Runtime

2. Is it fast?

- Let $T(n)$ represent the runtime of **mergesort** on a list of length n .
 - Extending this notation, $T(n/2)$ is the runtime of **mergesort** on a list of length $n/2$ and $T(1000)$ is the runtime of **mergesort** on a list of length 1000.
 - Calling **mergesort** on a list of length n calls **mergesort** once for each half, a total runtime of $T([n/2]) + T([n/2])$.
 - What is the runtime of **merge**? 

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    loop while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
        else:
            result.append(R[r_idx])
            r_idx += 1
    result.extend(L[l_idx:len(L)])
    result.extend(R[r_idx:len(R)])
    return result
```

Mergesort

```
def merge(L, R):
    result = []
    l_idx, r_idx = (0, 0)
    while l_idx < len(L) and r_idx < len(R):
        if L[l_idx] < R[r_idx]:
            result.append(L[l_idx])
            l_idx += 1
        else:
            result.append(R[r_idx])
            r_idx += 1
    result.extend(L[l_idx:len(L)])
    result.extend(R[r_idx:len(R)])
    return result
```

At most $\text{len}(L) + \text{len}(R)$,
which is n iters

Analyzing Runtime

2. Is it fast?

- Let $T(n)$ represent the runtime of `mergesort` on a list of length n .
 - Extending this notation, $T(n/2)$ is the runtime of `mergesort` on a list of length $n/2$ and $T(1000)$ is the runtime of `mergesort` on a list of length 1000.
 - Calling `mergesort` on a list of length n calls `mergesort` once for each half, a total runtime of $T([n/2]) + T([n/2])$.
 - What is the runtime of `merge`?

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Analyzing Runtime

2. Is it fast?

- Let $T(n)$ represent the runtime of `mergesort` on a list of length n .
 - Extending this notation, $T(n/2)$ is the runtime of `mergesort` on a list of length $n/2$ and $T(1000)$ is the runtime of `mergesort` on a list of length 1000.
 - Calling `mergesort` on a list of length n calls `mergesort` once for each half, a total runtime of $T([n/2]) + T([n/2])$.
 - What is the runtime of `merge`? $\Theta(n)$.

```
def mergesort(A):
    if len(A) <= 1:
        return A
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    return merge(left, right)
```

Analyzing Runtime

2. Is it fast?

- Let $T(n)$ represent the runtime of `mergesort` on a list of length n .
 - Extending this notation, $T(n/2)$ is the runtime of `mergesort` on a list of length $n/2$ and $T(1000)$ is the runtime of `mergesort` on a list of length 1000.
 - Calling `mergesort` on a list of length n calls `mergesort` once for each half, a total runtime of $T([n/2]) + T([n/2])$.
 - What is the runtime of `merge`? $\Theta(n)$.
- Here's our first **recurrence relation**!
 - $T(0) = \Theta(1)$
 - $T(1) = \Theta(1)$
 - $T(n) = T([n/2]) + T([n/2]) + \Theta(n)$

merge

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.
 - A well-known recurrence relation defines the Fibonacci sequence:
 $T(n) = T(n-1) + T(n-2)$.
피보나치수열

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.
 - A well-known recurrence relation defines the Fibonacci sequence:
 $T(n) = T(n-1) + T(n-2)$.
- Our recurrence relation for the runtime of **mergesort** isn't very useful unless we can determine the runtime as closed-form expression.

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.
 - A well-known recurrence relation defines the Fibonacci sequence:
 $T(n) = T(n-1) + T(n-2)$.
- Our recurrence relation for the runtime of **mergesort** isn't very useful unless we can determine the runtime as closed-form expression.
 - Let's learn how to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$!

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\hookrightarrow O(?)} + \underbrace{T(\lfloor n/2 \rfloor)}_{\hookrightarrow O(?)} + \Theta(n)$$

Analyzing Runtime

- First, let's make a few simplifications.

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

$$T(1) = \Theta(1) \rightarrow C \times n$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

Analyzing Runtime

- First, let's make a few simplifications.
 - **Simplification 1** Using the definition of Big-Θ, rewrite $\Theta(1)$ and $\Theta(n)$ terms.

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

$$T(1) \leq c_1$$

$$T(n) \leq T([n/2]) + T([n/2]) + c_2 n$$

Analyzing Runtime

- First, let's make a few simplifications.
 - Simplification 1** Using the definition of Big- Θ , rewrite $\Theta(1)$ and $\Theta(n)$ terms.
 - Simplification 2** $n = 1$ 2의 제곱승만
즉 $2 \times 1^{\text{번}} = 2$.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \leq c_1$$

$$T(n) \leq 2T(n/2) + c_2 n$$

Analyzing Runtime

- First, let's make a few simplifications.
 - **Simplification 1** Using the definition of Big- Θ , rewrite $\Theta(1)$ and $\Theta(n)$ terms.
 - **Simplification 2** n is a power of 2.
 - **Simplification 3** $c = \max\{c_1, c_2\}$.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Analyzing Runtime

- First, let's make a few simplifications.
 - **Simplification 1** Using the definition of Big- Θ , rewrite $\Theta(1)$ and $\Theta(n)$ terms.
 - **Simplification 2** n is a power of 2.
 - **Simplification 3** $c = \max\{c_1, c_2\}$.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Analyzing Runtime

- First, let's make a few simplifications.
 - **Simplification 1** Using the definition of Big- Θ , rewrite $\Theta(1)$ and $\Theta(n)$ terms.
 - **Simplification 2** n is a power of 2.
 - **Simplification 3** $c = \max\{c_1, c_2\}$.

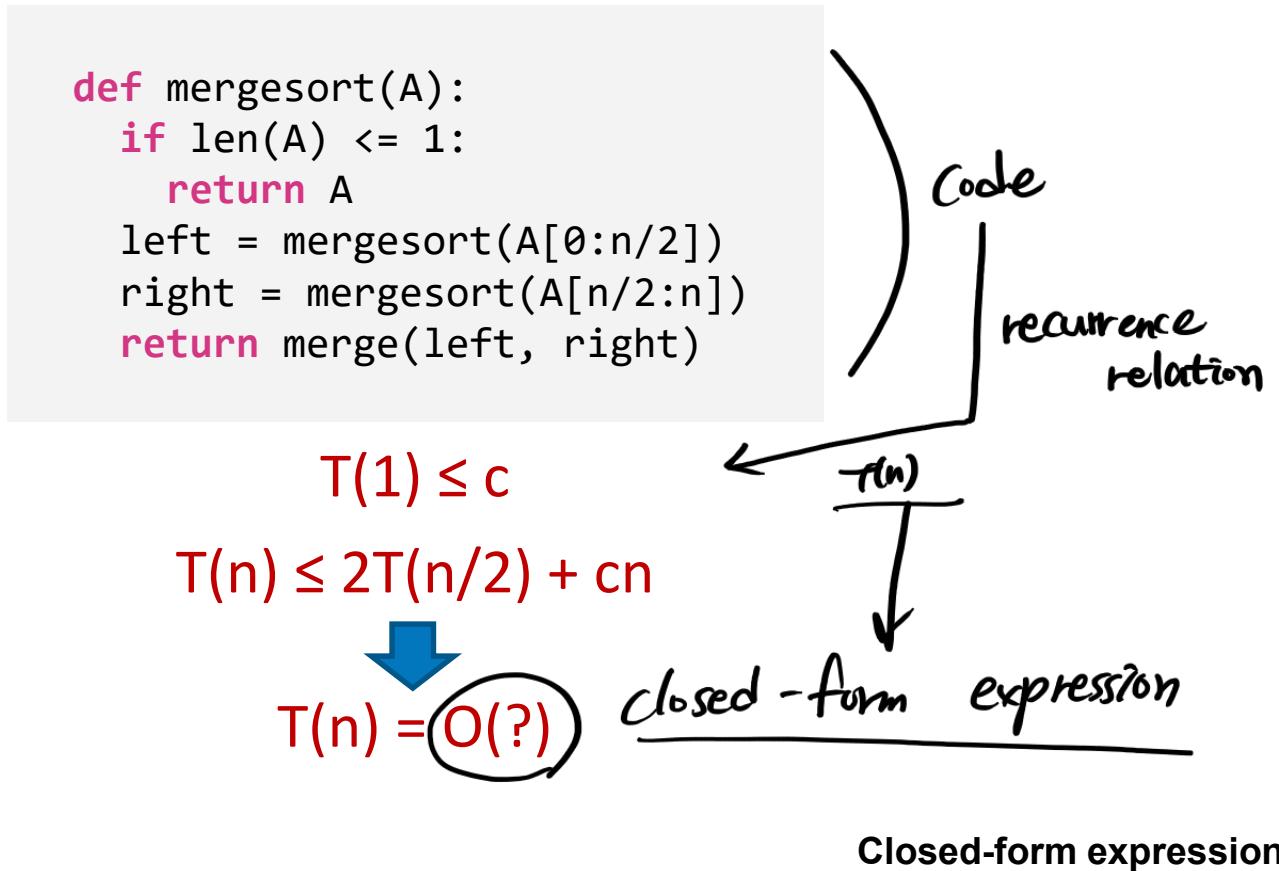
$$\begin{aligned}\cancel{T(0) = \Theta(1)} \\ T(1) \leq c \\ T(n) \leq 2T(n/2) + cn\end{aligned}$$

How do we translate this simplified recurrence relation to a closed-form expression?

Big-O

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.



Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - Recursion tree method
 - Iteration method
 - Master method
 - Substitution Method

4가지 방법

→ 핵심 2가지.

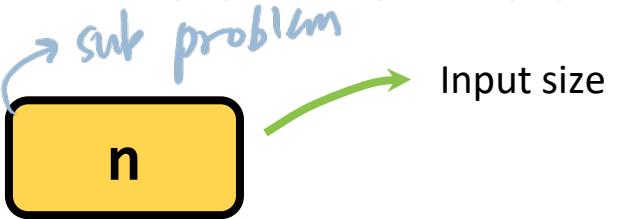
Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method**
 - Iteration method
 - Master method
 - Substitution Method

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

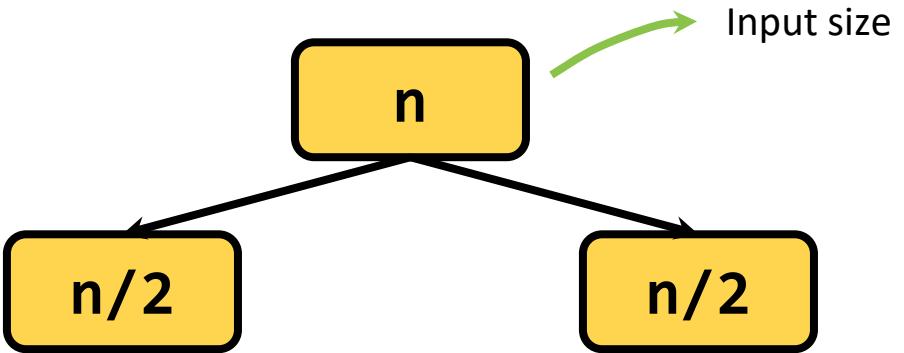
Recursion Tree Method



$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

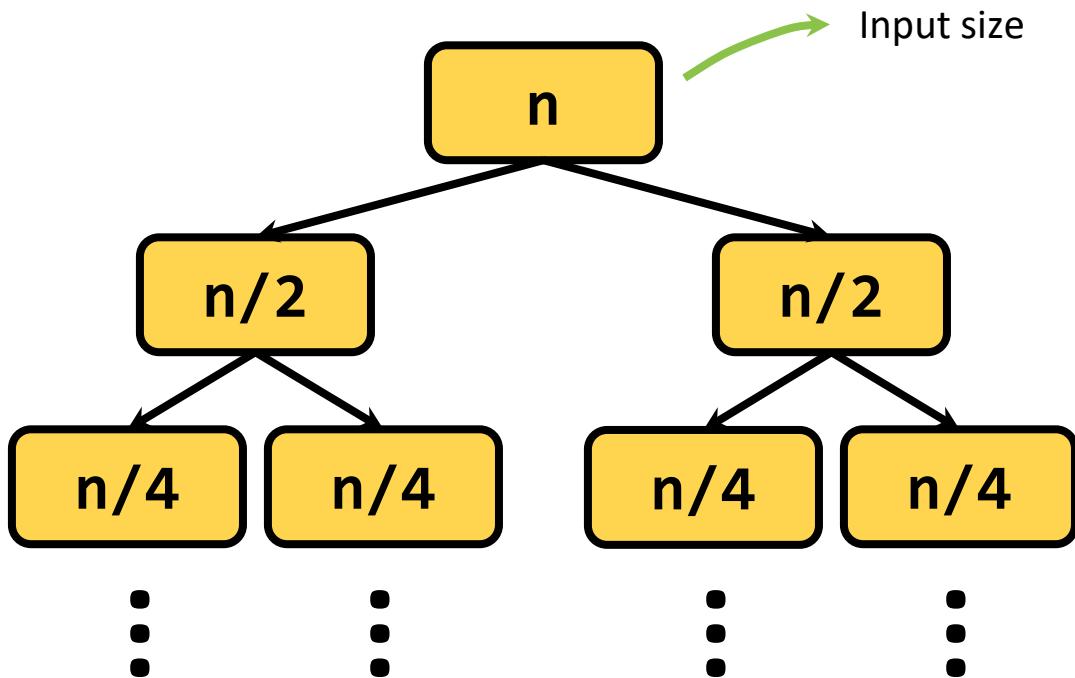
Recursion Tree Method



$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

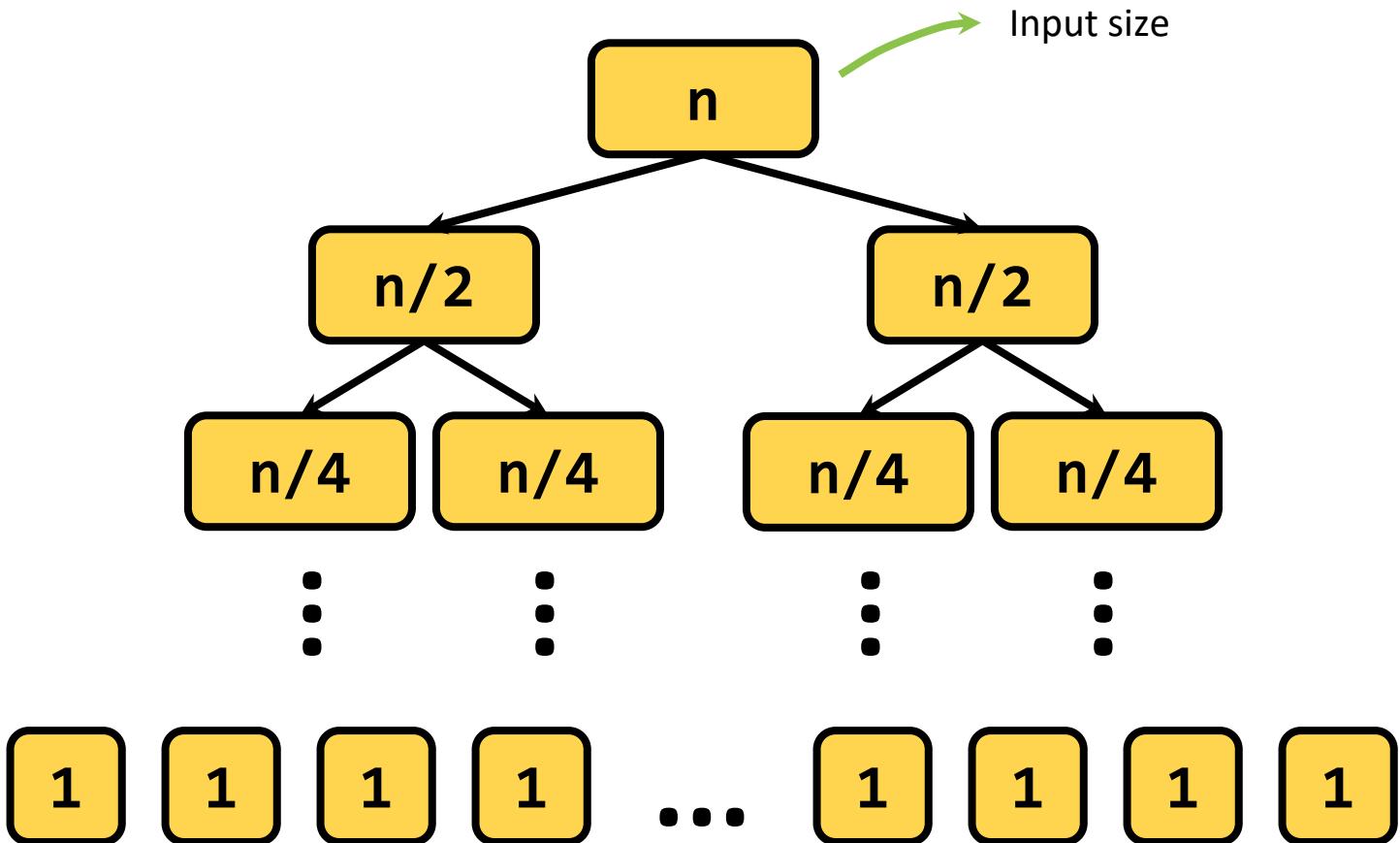
Recursion Tree Method



$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

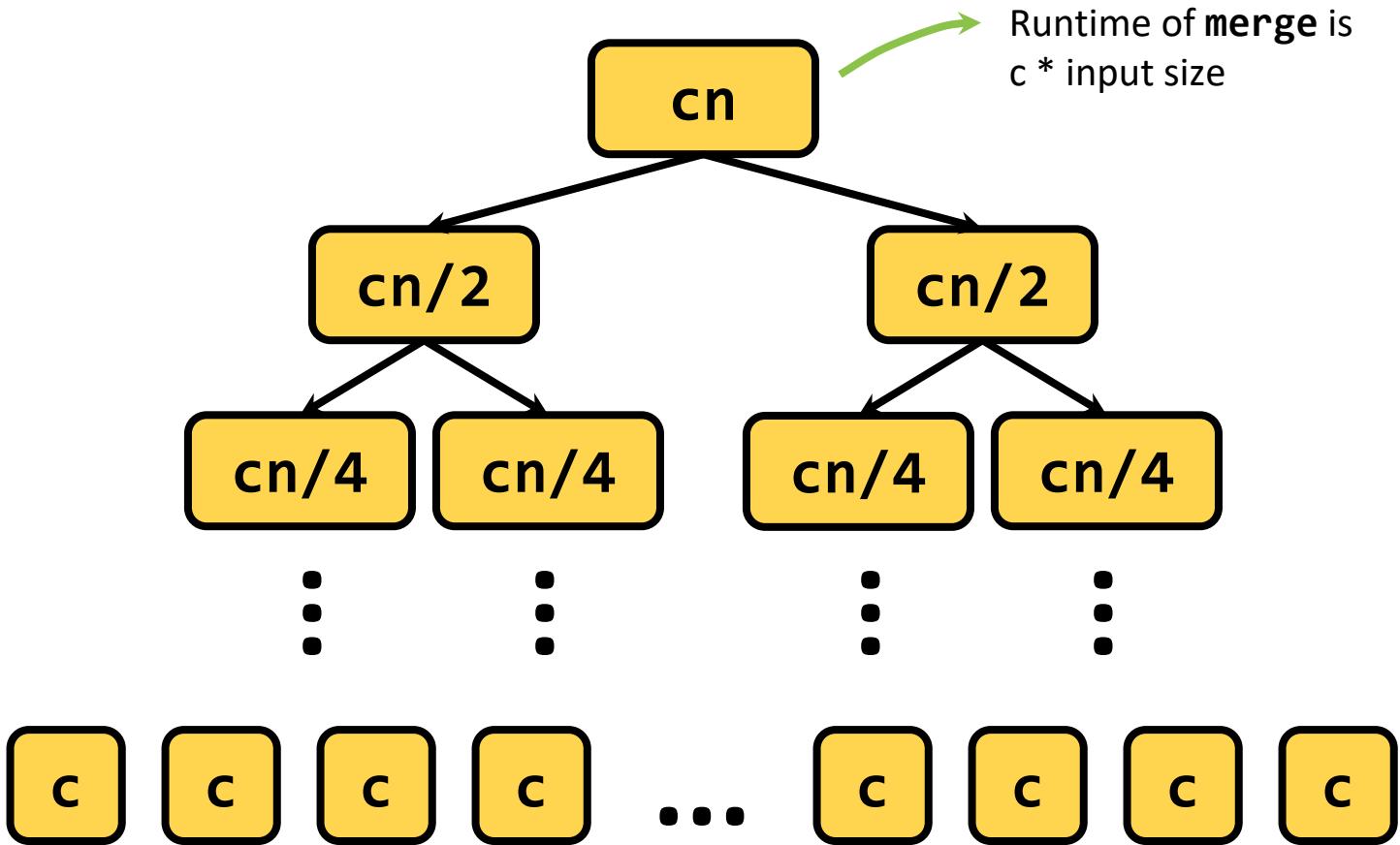
Recursion Tree Method



$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

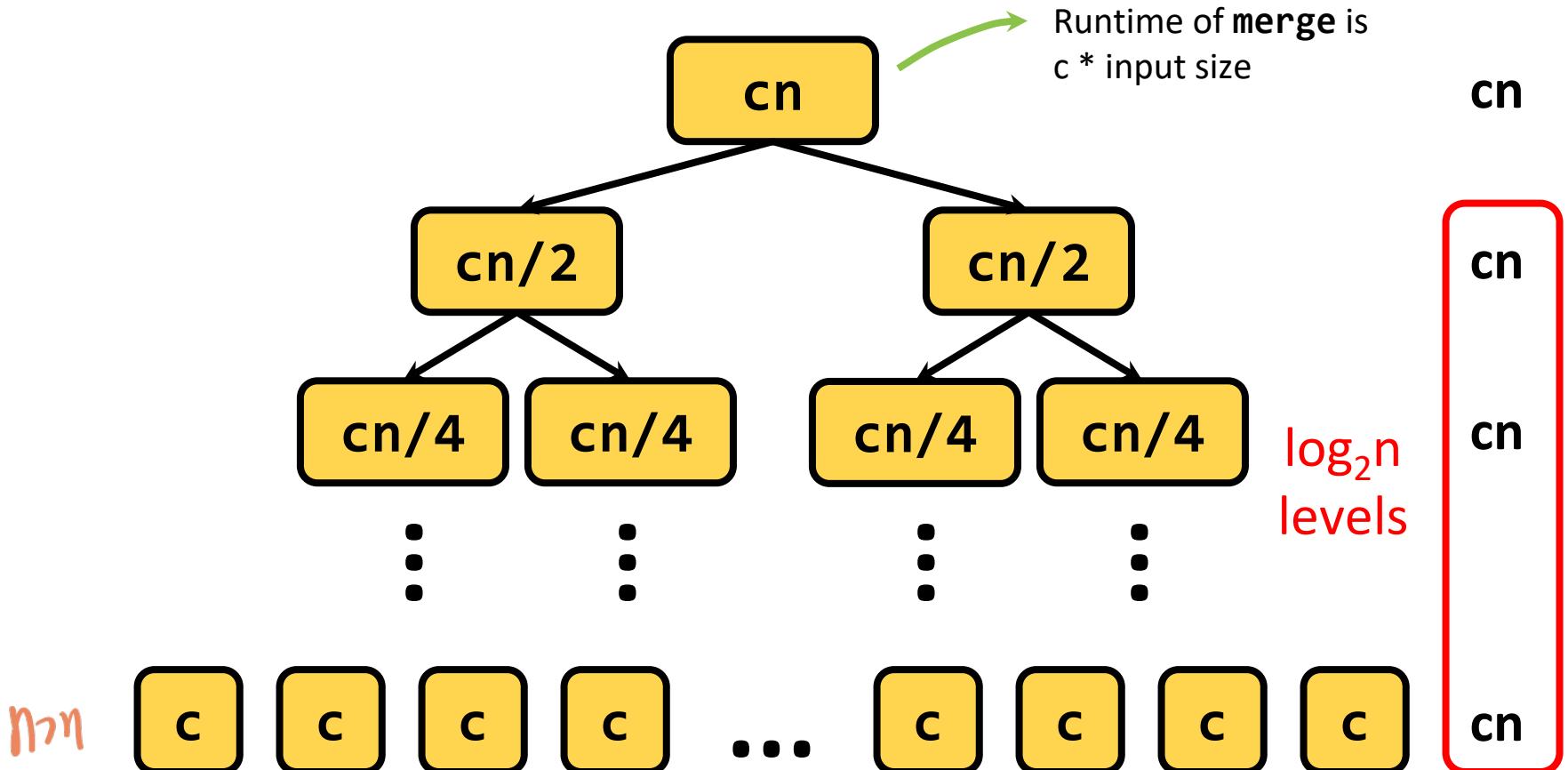
Recursion Tree Method



$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Recursion Tree Method



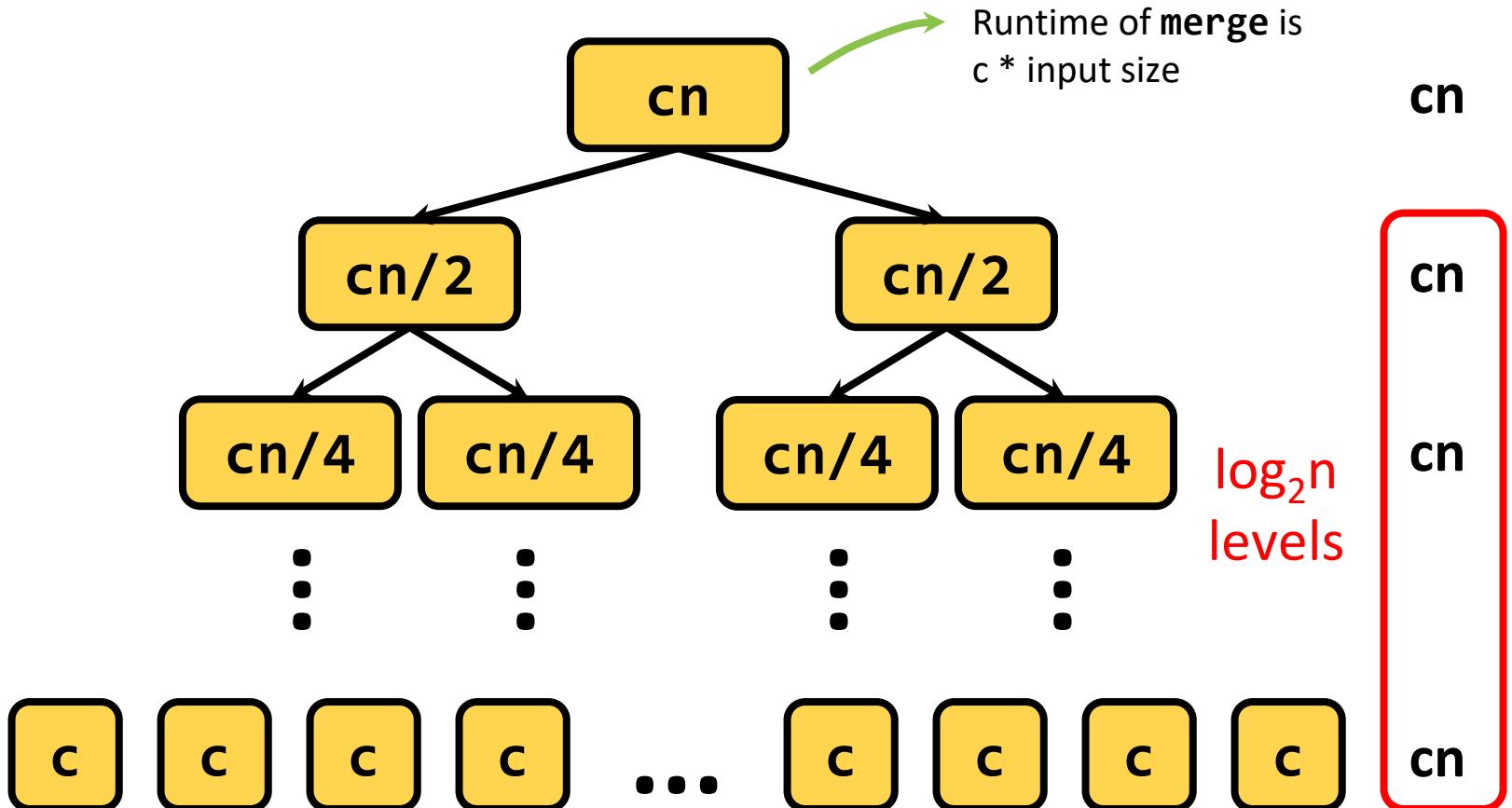
$$2^h = n$$

$$h = \log_2 n$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Recursion Tree Method

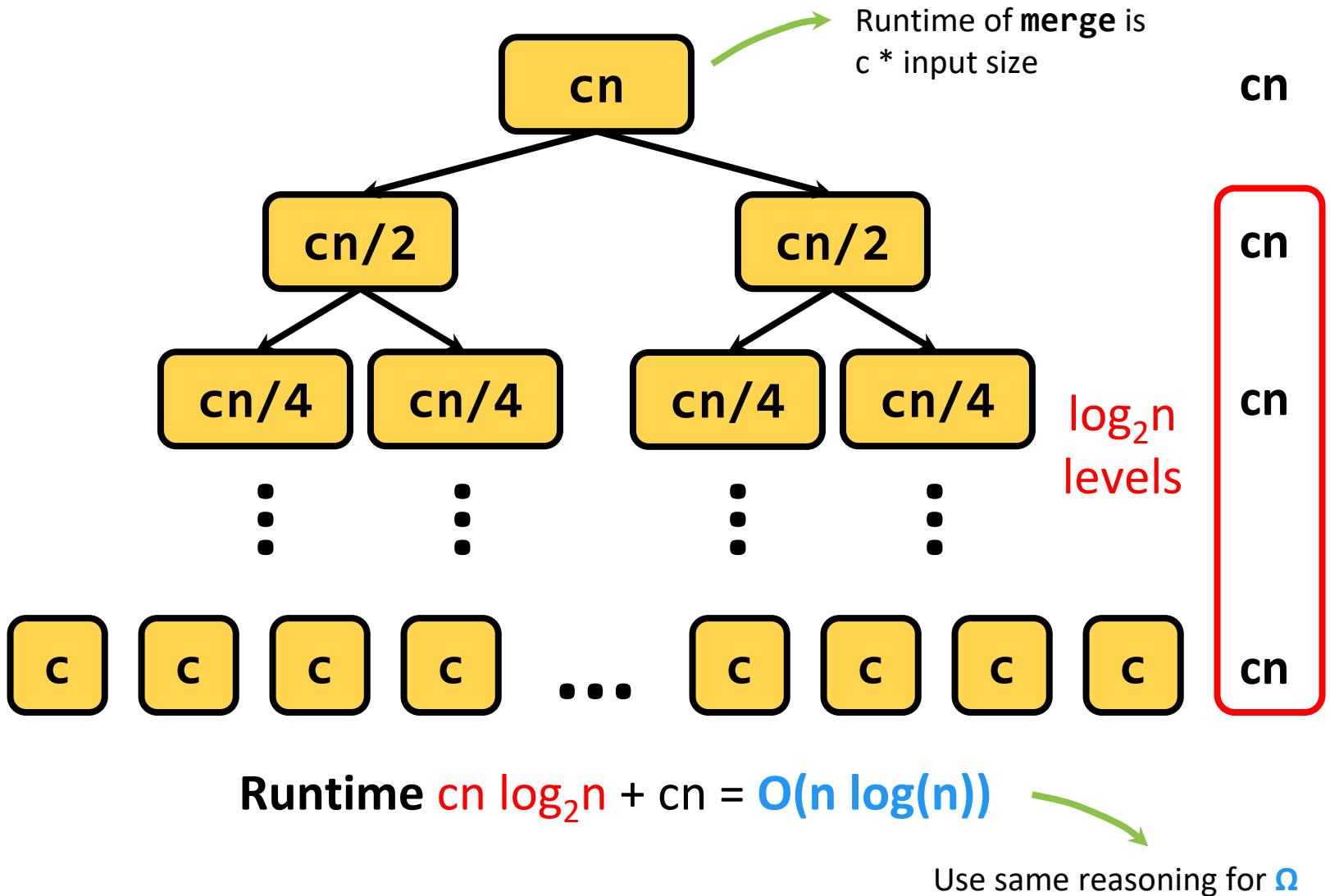


Runtime $\underline{cn \log_2 n} + cn = O(n \log(n))$
closed form expression

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Recursion Tree Method



Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method $\Theta(n \log(n))$.**
 - Iteration method
 - Master method
 - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method** $\Theta(n \log(n))$.
 - **Iteration method**
 - Master method
 - Substitution Method

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

- Apply the relationship until you see a pattern.

$$T(n) \leq 2 \cdot T(n/2) + cn$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{4}\right) + \frac{cn}{2}$$

$$T(n) \leq 2 \cdot \left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn$$

$$T(n) \leq 4T\left(\frac{n}{4}\right) + 2cn$$

⋮

$$\leq 3 \cdot T\left(\frac{n}{8}\right) + 3cn$$

$k = 2^3$ (나눌 수 있는 횟수)

$$\therefore k = \log_2 n$$

$$T(n) \leq 2^k T\left(\frac{n}{2^k}\right) + kn$$

$2^k n$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Iteration Method

- Apply the relationship until you see a pattern.

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + cn \\ &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\ &= 4 \cdot T(n/4) + 2cn \end{aligned}$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Iteration Method

- Apply the relationship until you see a pattern.

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + cn \\ &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\ &= 4 \cdot T(n/4) + 2cn \\ &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\ &= 8 \cdot T(n/8) + 3cn \end{aligned}$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

- Apply the relationship until you see a pattern.

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + cn \\ &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\ &= 4 \cdot T(n/4) + 2cn \\ &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\ &= 8 \cdot T(n/8) + 3cn \\ &\dots \\ &\leq 2^k T(n/2^k) + kcn \end{aligned}$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Iteration Method

- Apply the relationship until you see a pattern.

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + cn \\ &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\ &= 4 \cdot T(n/4) + 2cn \\ &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\ &= 8 \cdot T(n/8) + 3cn \\ &\dots \\ &\leq 2^k T(n/2^k) + kcn \end{aligned}$$

- What is k?

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

- Apply the relationship until you see a pattern.

$$\begin{aligned}
 T(n) &\leq 2 \cdot T(n/2) + cn \\
 &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\
 &= 4 \cdot T(n/4) + 2cn \\
 &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\
 &= 8 \cdot T(n/8) + 3cn \\
 &\dots \\
 &\leq 2^k T(n/2^k) + kcn
 \end{aligned}$$

- What is k ? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$T(n) \leq 2^k T(n/2^k) + kcn$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

- Apply the relationship until you see a pattern.

$$\begin{aligned}
 T(n) &\leq 2 \cdot T(n/2) + cn \\
 &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\
 &= 4 \cdot T(n/4) + 2cn \\
 &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\
 &= 8 \cdot T(n/8) + 3cn \\
 &\dots \\
 &\leq 2^k T(n/2^k) + kcn
 \end{aligned}$$

- What is k ? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$\begin{aligned}
 T(n) &\leq 2^k T(n/2^k) + kcn \\
 &= 2^{\log_2 n} T(n/2^{\log_2 n}) + cn\log_2 n
 \end{aligned}
 \quad \text{Substitute } k = \log_2 n$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

- Apply the relationship until you see a pattern.

$$\begin{aligned}
 T(n) &\leq 2 \cdot T(n/2) + cn \\
 &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\
 &= 4 \cdot T(n/4) + 2cn \\
 &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\
 &= 8 \cdot T(n/8) + 3cn \\
 &\dots \\
 &\leq 2^k T(n/2^k) + kcn
 \end{aligned}$$

- What is k ? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$\begin{aligned}
 T(n) &\leq 2^k T(n/2^k) + kcn \\
 &= 2^{\log_2 n} T(n/2^{\log_2 n}) + cn\log_2 n \\
 &= nT(1) + cn\log_2 n
 \end{aligned}$$

Substitute $k = \log_2 n$
Simplify

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

- Apply the relationship until you see a pattern.

$$\begin{aligned}
 T(n) &\leq 2 \cdot T(n/2) + cn \\
 &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\
 &= 4 \cdot T(n/4) + 2cn \\
 &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\
 &= 8 \cdot T(n/8) + 3cn \\
 &\dots \\
 &\leq 2^k T(n/2^k) + kcn
 \end{aligned}$$

- What is k ? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$\begin{aligned}
 T(n) &\leq 2^k T(n/2^k) + kcn \\
 &= 2^{\log_2 n} T(n/2^{\log_2 n}) + cn\log_2 n && \text{Substitute } k = \log_2 n \\
 &= nT(1) + cn\log_2 n && \text{Simplify} \\
 &\leq cn + cn\log_2 n \\
 &= O(n\log n)
 \end{aligned}$$

Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method** $\Theta(n \log(n))$.
 - **Iteration method** $\Theta(n \log(n))$.
 - Master method
 - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method** $\Theta(n \log(n))$.
 - **Iteration method** $\Theta(n \log(n))$.
 - **Master method**
 - Substitution Method

More examples

- Needlessly recursive integer multiplication

$$T(n) = 4 T(n/2) + O(n)$$

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

- Karatsuba integer multiplication

$$T(n) = 3 T(n/2) + O(n)$$

$$T(n) = O(n^{1.5} \approx n^{1.6})$$

- MergeSort

$$T(n) = \underline{2T(n/2) + O(n)}$$

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

- Another example

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

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

What's the pattern?!?!!?!

Today's Outline

- Divide and Conquer I
 - ~~Proving correctness with induction~~ Done!
 - ~~Proving runtime with recurrence relations~~ Done!
 - Proving the Master method
 - *Problems: Comparison-sorting*
 - *Algorithms: Mergesort*
 - Reading: CLRS 2.3, 4.3-4.6

Solving Recurrences

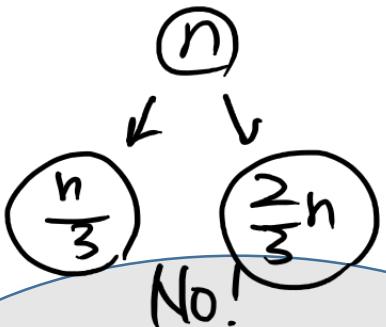
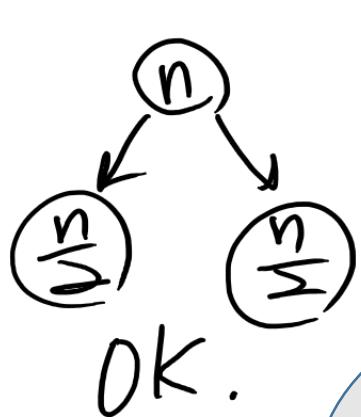
- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method** $\Theta(n \log(n))$.
 - **Iteration method** $\Theta(n \log(n))$.
 - **Master method**
 - Substitution Method

Master Method



- A **formula** that solves recurrences when all of the sub-problems are the same size

- We will see an example later when not all problems are the same size.



A useful
formula it is.
You should know
why it works.



Jedi master Yoda

Master Method

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

The Master method states:

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

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Three parameters:

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

Master Method

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

We can also take n/b to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$ and the theorem is still true.

The Master method states:

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

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

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

Three parameters:

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

Understanding the Master Theorem

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

- What do these three cases mean?

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

The Eternal Struggle



Branching causes the number
of problems to explode!
**The most work is at the
bottom of the tree!**

The problems lower in
the tree are smaller!
**The most work is at
the top of the tree!**

Consider Examples

$$1. \quad T(n) = T\left(\frac{n}{2}\right) + n$$

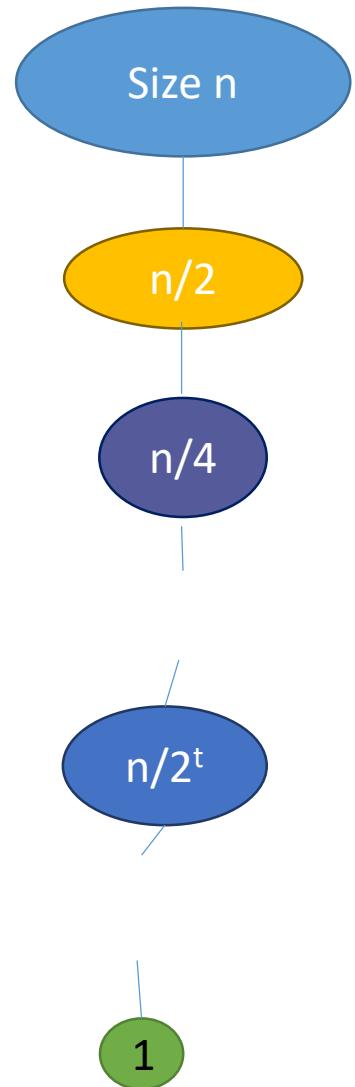
$$2. \quad T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

$$3. \quad T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$$

First example: tall and skinny tree

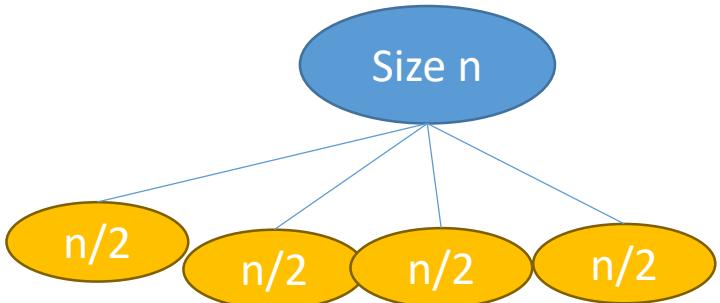
$$1. T(n) = T\left(\frac{n}{2}\right) + n, \quad (a < b^d)$$

- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.
- $T(n) = O(\text{work at top}) = O(n)$



Third example: bushy tree

$$3. \quad T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \quad (a > b^d)$$

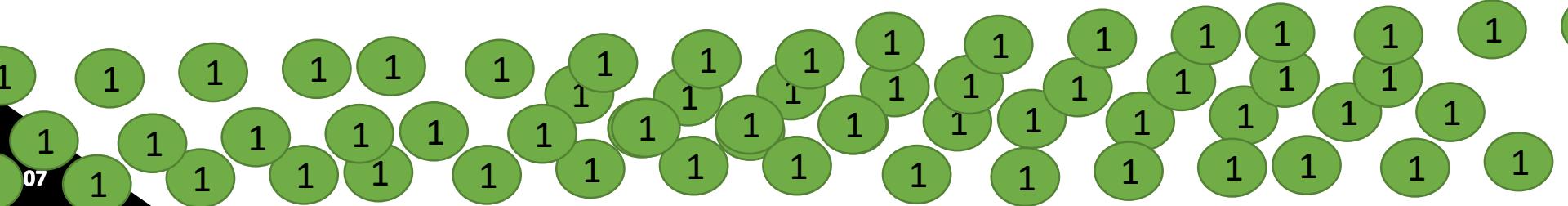


WINNER



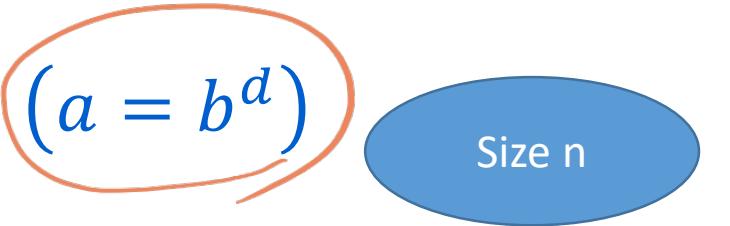
**Most work at
the bottom
of the tree!**

- There are a **HUGE** number of leaves, and the total work is dominated by the time to do work at these leaves.
bottom! dominate!
- $T(n) = O(\text{work at bottom}) = O(4^{\text{depth of tree}}) = O(n^2)$

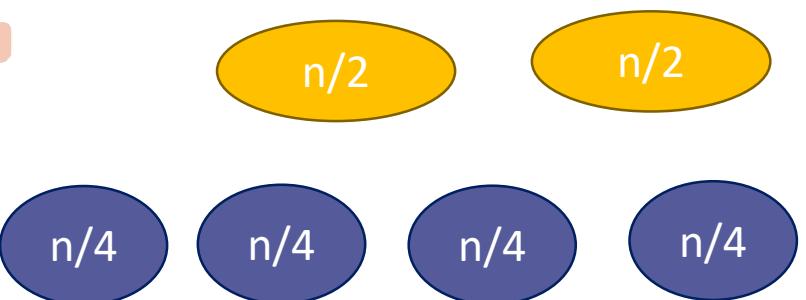


Second example: just right

$$2. \quad T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n,$$



- The branching **just balances** out the amount of work.
- The same amount of work is done at every level.
- $T(n) = (\text{number of levels}) * (\text{work per level})$
- $= \log(n) * O(n) = O(n \log(n))$



Master Method

- We can prove the Master Method by writing out a generic proof using a recursion tree.
 - Draw out the tree.
 - Determine the work per level.
 - Sum across all levels.
- The three cases of the Master Method correspond to whether the recurrence is top heavy, balanced, or bottom heavy.

proof in textbook

Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - **Recursion tree method** $\Theta(n \log(n))$.
 - **Iteration method** $\Theta(n \log(n))$.
 - **Master method** $\Theta(n \log(n))$.
 - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
 - Recursion tree method $\Theta(n \log(n))$.
 - Iteration method $\Theta(n \log(n))$.
 - Master method $\Theta(n \log(n))$.
 - Substitution Method Next time!

Master 를 쓰는 경우 \rightarrow sub problem 102가지 \rightarrow 2¹⁰² 가지.

Mergesort

```
def mergesort(A):
    if len(A) <= 1:
        return A
    L = mergesort(A[0:n/2])
    R = mergesort(A[n/2:n])
    return merge(L, R)
```

Worst-case runtime $\Theta(n \log(n))$

Mergesort

```
def mergesort(A):
    if len(A) <= 1:
        return A
    L = mergesort(A[0:n/2])
    R = mergesort(A[n/2:n])
    return merge(L, R)
```

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



It's the same as the
worst-case runtime!

Today's Outline

- Divide and Conquer I
 - ~~Proving correctness with induction~~ Done!
 - ~~Proving runtime with recurrence relations~~ Done!
 - How do we measure the runtime of a recursive algorithm?
 - ~~Proving the Master method~~ Done!
 - A useful theorem so we do not have to answer this question from scratch each time
 - *Problems: Comparison-sorting*
 - *Algorithms: Mergesort*
 - Reading: CLRS 2.3, 4.3-4.6

So far...

	Proving correctness	Proving runtime
Iterative	Induction on the iteration, leveraging a loop invariant (e.g. insertion sort)	<i>Iteration invariant</i> Intuition
Recursive	<i>recursion invariant</i> Induction on the input size (e.g. mergesort)	Defining and solving recurrence relations

So far...

	Proving correctness	Proving runtime
Iterative	Induction on the iteration, leveraging a loop invariant (e.g. insertion sort)	Intuition
Recursive	Induction on the input size (e.g. mergesort)	Defining and solving recurrence relations

So far...

- **Divide-and-conquer algorithms via defining and solving recurrence relations**
 - After deriving the recurrence relation, we learned several methods to find the closed-form runtime expression: recursion-tree method, iteration method, Master method.
 - Now, we will learn another method: **substitution method!**

Master Method

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

The Master method states:

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

Three parameters:

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

A powerful theorem it is...

The master theorem only works when all sub-problems are the same size.

Subproblem Aolz Zlotof



Jedi master Yoda

Substitution Method

Substitution Method

1. Guess what the answer is. *답을 짜기*
 2. Formally prove that's what the answer is. *증명하기*
- Let's try it out with an example recurrence from last time:
 - $T(1) \leq 1$
 - $T(n) \leq 2T(n/2) + n$

$$T(1) \leq 1$$

$$T(n) \leq 2T(n/2) + n$$



Substitution Method

1. Guess what the answer is.

- Try solving it...

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

$$T(n) = 2(2T(n/4) + n/2) + n$$

$$T(n) = 4T(n/4) + 2n$$

$$T(n) = 4(2T(n/8) + n/4) + 2n$$

$$T(n) = 8T(n/8) + 3n$$

...

- Following the pattern...

$$T(n) = nT(1) + n \log(n) = n (\log(n) + 1)$$

$$T(1) \leq 1$$

$$T(n) \leq 2T(n/2) + n$$



Substitution Method

2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \leq k(\log(k) + 1)$ for all $1 \leq k < n$.
- **Base case** $T(1) = 1 = 1(\log(1) + 1)$.
- **Inductive step**
 - $$\begin{aligned} T(n) &= 2T(n/2) + n && \text{Substitute } n/2 \text{ into inductive hyp.} \\ &\leq 2((n/2)(\log(n/2) + 1) + n \\ &= 2((n/2)(\log(n) - 1 + 1) + n \\ &= 2((n/2) \log(n)) + n \\ &= n(\log(n) + 1) \end{aligned}$$
- **Conclusion** By induction, $T(n) = n(\log(n) + 1)$ for all $n > 0$.

Substitution Method

- So far, just seems like a different way of doing the same thing.
- But, let's try it out with a new recurrence:
 - $T(n) = 10n$, when $1 \leq n \leq 10$
 - $T(n) = 3n + T(n/5) + T(n/2)$, otherwise

Substitution Method

$$T(n) = 10n, \text{ when } 1 \leq n \leq 10$$

$$T(n) = 3n + T(n/5) + T(n/2), \text{ otherwise}$$

1. Guess what the answer is.

- Try solving it

[Whiteboard] – Gets gross fast

- Try plotting it → Guess O(n)

- What else do we know?

- $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$

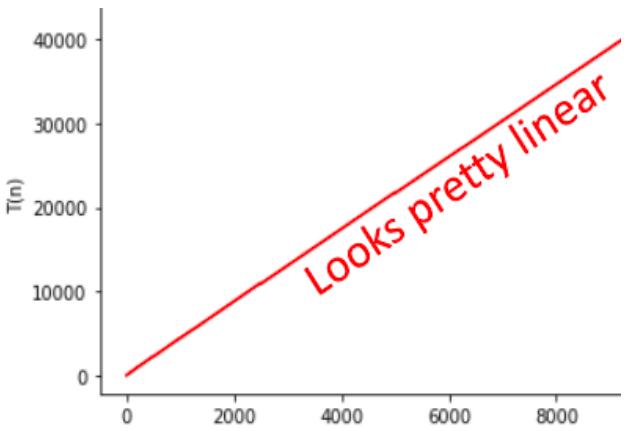
$$\leq 3n + 2 \cdot T\left(\frac{n}{2}\right)$$

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

- $T(n) \geq 3n$

- So the right answer is somewhere between $O(n)$ and $O(n \log(n))$.

- Let's guess $O(n)$



Substitution Method

$$T(n) = 10n \text{ when } 1 \leq n \leq 10$$

$$T(n) = 3n + T(n/5) + T(n/2) \text{ otherwise}$$

Guess $O(n)$

2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \leq Ck$ for all $1 \leq k < n$.

- **Base case** $T(k) \leq Ck$ for all $k \leq 10$.

- **Inductive step**

$$\begin{aligned} \circ \quad T(n) &= 3n + T(n/5) + T(n/2) \\ &\leq 3n + C(n/5) + C(n/2) \\ &= 3n + (C/5)n + (C/2)n \\ &\leq Cn \end{aligned}$$

C is some constant we'll have to fill in later!

C must be ≥ 10 since the recurrence states $T(k) = 10k$ when $1 \leq k \leq 10$

Solve for C to satisfy the inequality. $C = 10$ works.

- **Conclusion** There exists some C such that for all $n > 1$, $T(n) \leq Cn$. Therefore, $T(n) = O(n)$.

Substitution Method

$$T(n) = 10n \text{ when } 1 \leq n \leq 10$$

$$T(n) = 3n + T(n/5) + T(n/2) \text{ otherwise}$$

2. Formally prove that's what the answer is. Pretend like we knew it

- Inductive hypothesis $T(k) \leq 10k$ for all $1 \leq k < n$.

- Base case $T(k) \leq 10k$ for all $k \leq 10$.

- Inductive step

$$\begin{aligned} \circ \quad T(n) &= 3n + T(n/5) + T(n/2) \\ &\leq 3n + 10(n/5) + 10(n/2) \\ &= 3n + (10/5)n + (10/2)n \\ &\leq 10n \end{aligned}$$

Pretend we knew $C = 10$ all along.

- Conclusion For all $n > 1$, $T(n) \leq 10n$. Therefore, $T(n) = O(n)$.

Substitution Method

- What have we learned?
 - The substitution method can work when the master theorem doesn't.
 - For example with different-sized sub-problems
 - Step 1: generate a guess 예측
 - Step 2: try to prove that your guess is correct 증명 \rightarrow constant in parameter로 쓰다.
 - Might need to leave some constants unspecified until the end
 - Then see what they need to be for the proof to work
 - Step 3: profit
 - Pretend you didn't do Steps 1 and 2 and write down a nice proof

Today's Outline

- Divide and Conquer I
 - ~~Proving correctness with induction~~ Done!
 - ~~Proving runtime with recurrence relations~~ Done!
 - How do we measure the runtime of a recursive algorithm?
 - ~~Proving the Master method~~ Done!
 - A useful theorem (do not have to answer the runtime from scratch)
 - ~~Learn the Substitution method~~ Done!
 - It can be used when the master method doesn't work
 - *Problems: Comparison-sorting*
 - *Algorithms: Mergesort*
 - *Reading: CLRS 2.3, 4.3-4.6*

CSE301 Introduction to Algorithms

Divide and Conquer II

Fall 2022



Instructor : Hoon Sung Chwa

Course Overview

- Algorithmic Analysis
- **Divide and Conquer**
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Divide and Conquer II
 - Linear-time selection $O(n)$
 - Proving correctness
 - Proving runtime with recurrence relations
 - *Problems: selection*
 - *Algorithms: Select*
 - Reading: CLRS 9

Linear-Time Selection

Today's Outline

- Divide and Conquer II
 - Linear-time selection
 - Proving correctness
 - Proving runtime with recurrence relations
 - *Problems: selection*
 - *Algorithms: Select*
 - Reading: CLRS 9

Linear-Time Selection

- Task Find the k^{th} smallest element in an unsorted list in $O(n)$ -time.

1	64	9	49	16	4	0	25	36	81
---	----	---	----	----	---	---	----	----	----

SELECT(A,k): return the $k^{\text{'th}}$ smallest element in A

$$\text{SELECT}(A,0)=0$$

$$\text{SELECT}(A,1)=1$$

$$\text{SELECT}(A,2)=4$$

$$\text{SELECT}(A,4)=16$$

$$\text{SELECT}(A,9)=81$$

$$\text{SELECT}(A,0)=\text{MIN}(A) \quad \rightarrow \text{최소값}$$

$$\text{SELECT}(A,n/2-1)=\text{MEDIAN}(A)$$

$$\text{SELECT}(A,n-1)=\text{MAX}(A)$$

- Such an algorithm could find the **min** in $O(n)$ -time if $k=0$ or the **max** if $k=n-1$.
- Such an algorithm could find the **median** in $O(n)$ -time if $k=[n/2]-1$ (this definition allows the median of lists of even-length to always be elements of the list, as opposed to the average of two elements).

Linear-Time Selection

- **Finding the min and max**

Iterate through the list and keep track of the smallest and largest elements.

Runtime $O(n)$.

- **Finding the k^{th} smallest element (naive)**

Sort the list and return the element in index k of the sorted list.

1	64	9	49	16	4	0	25	36	81
---	----	---	----	----	---	---	----	----	----

$k=3$



0	1	4	9	16	25	36	49	64	81
---	---	---	---	----	----	----	----	----	----

Not Quite Linear-Time Selection

```
def naive_select(A, k):  
    A = mergesort(A)  
    return A[k]
```

Worst-case runtime
 $\Theta(n \log(n))$

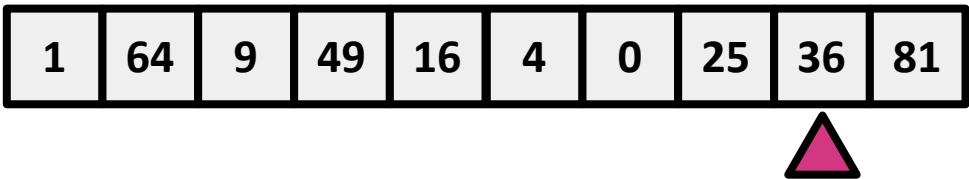
Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.

1	64	9	49	16	4	0	25	36	81
---	----	---	----	----	---	---	----	----	----

Linear-Time Selection

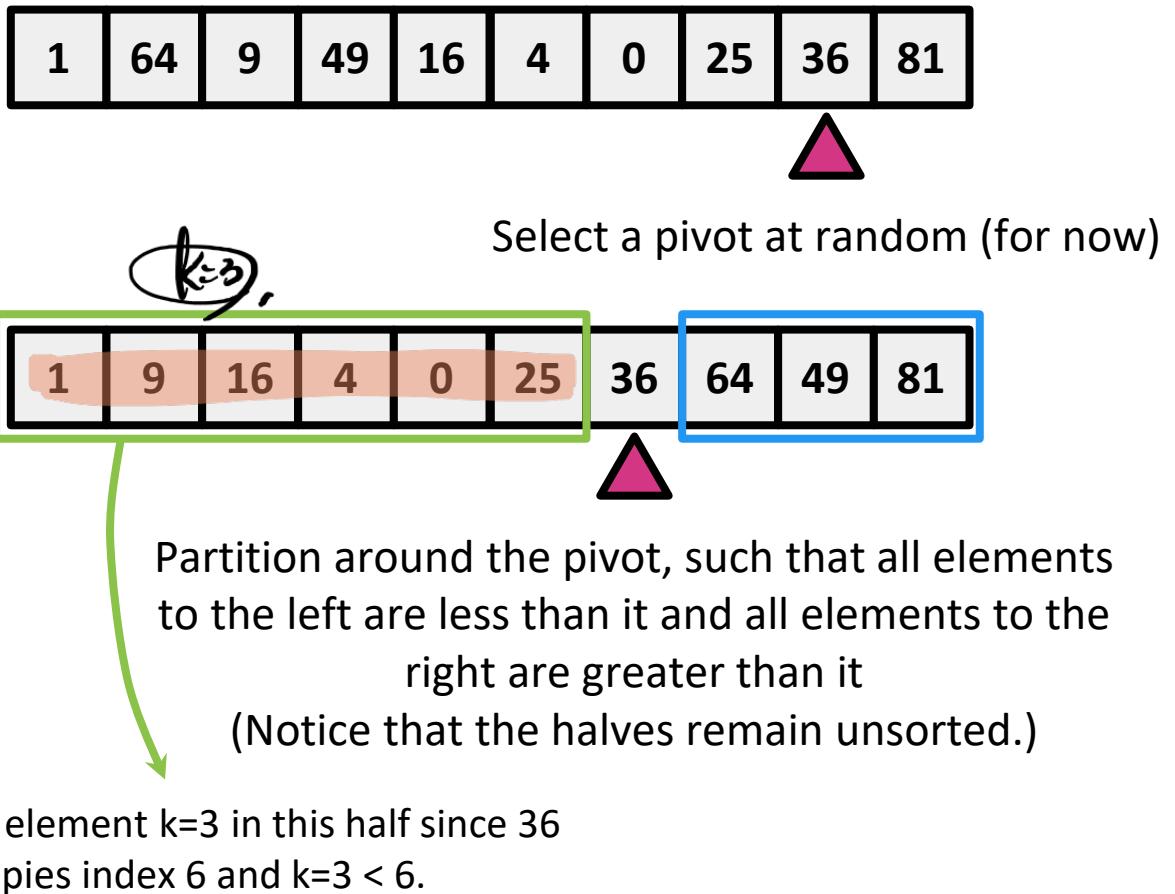
- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



Select a pivot at random (for now)

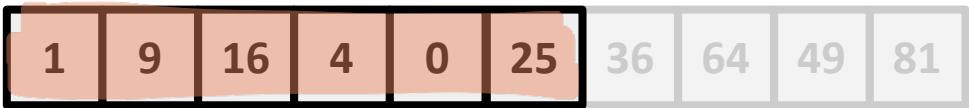
Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



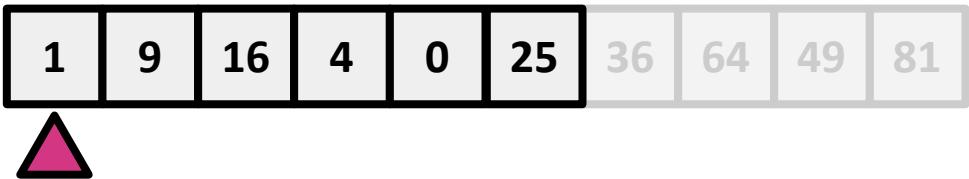
Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



Linear-Time Selection

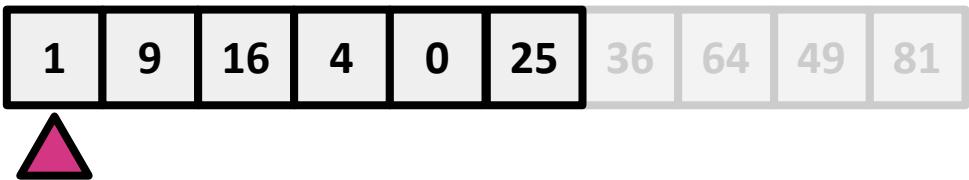
- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



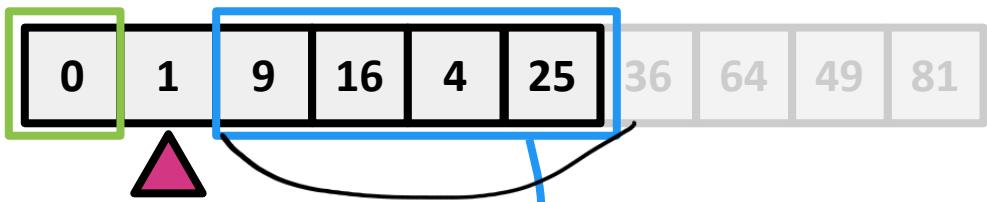
Select another pivot at random (for now)

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



Select another pivot at random (for now)



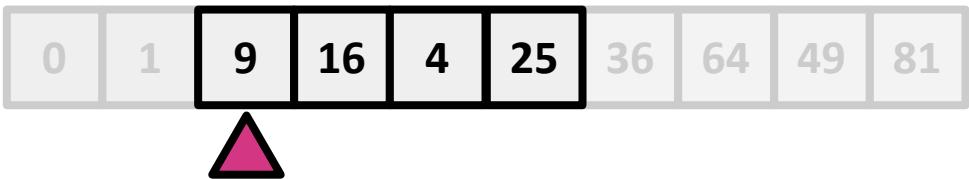
Partition around the pivot

Find element $k=3-(1+1)$ in this half since 1 occupies index 1 and $k=3 > 1$.



Linear-Time Selection

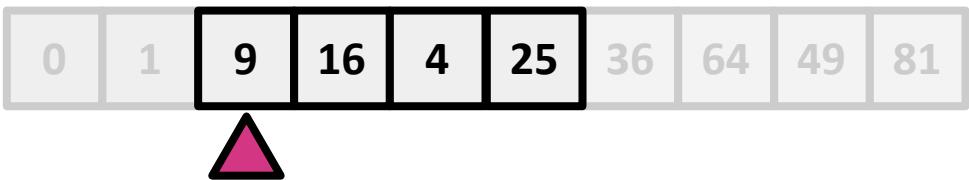
- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



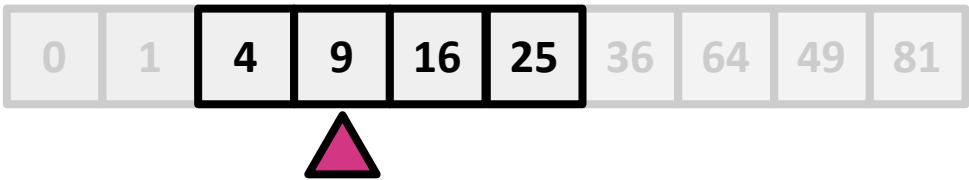
Select another pivot at random (for now)

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



Select another pivot at random (for now)



Partition around the pivot
We found the element!

Linear-Time Selection

```
def select(A, k, c=100):
```

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
```

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
```

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
```

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
```

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

“Worst-case” runtime
 $\Theta(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

“Worst-case” runtime

$\Theta(n^2)$



We'll discuss this
runtime later...

Linear-Time Selection

```

def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
  
```

Note: this is different from the “worst-case” we saw for insertion sort (we’ll revisit during Randomized Algs).

“Worst-case” runtime

$\Theta(n^2)$

We'll discuss this runtime later...



Linear-Time Selection

```
def partition_about_pivot(A, pivot):
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
    left, right = [], []
    for i in range(len(A)):
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
    left, right = [], []
    for i in range(len(A)):
        if A[i] == pivot: continue
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
    left, right = [], []
    for i in range(len(A)):
        if A[i] == pivot: continue
        elif A[i] < pivot:
            left.append(A[i])
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
    left, right = [], []
    for i in range(len(A)):
        if A[i] == pivot: continue
        elif A[i] < pivot:
            left.append(A[i])
        else:
            right.append(A[i])
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
    left, right = [], []
    for i in range(len(A)):
        if A[i] == pivot: continue
        elif A[i] < pivot:
            left.append(A[i])
        else:
            right.append(A[i])
    return left, right
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
    left, right = [], []
    for i in range(len(A)):
        if A[i] == pivot: continue
        elif A[i] < pivot:
            left.append(A[i])
        else:
            right.append(A[i])
    return left, right
```

Worst-case runtime
 $\Theta(n)$

Linear-Time Selection

- **Intuition** Partition the list about a pivot selected at random, either return the pivot itself or recurse on the left or right sublists (but not both).
- You might have two questions at this point...
 1. **Does this actually work?**
 2. **Is it fast?**

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Linear-Time Selection

- **Intuition** Partition the list about a pivot selected at random, either return the pivot itself or recurse on the left or right sublists (but not both).
- You might have two questions at this point...
 1. **Does this actually work?**
 2. **Is it fast?**

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Linear-Time Selection

1. Does this actually work? We've already seen an example!

- Formally, similar to last time, we proceed by induction, inducting on the length of the input list.

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Proving Correctness

```
def proof_of_correctness_helper(algorithm):  
    if algorithm.type == "iterative":  
        # 1) Find the loop invariant  
        # 2) Define the inductive hypothesis  
        #     (internal state at iteration i)  
        # 3) Prove the base case (i=0)  
        # 4) Prove the inductive step (i => i+1)  
        # 5) Prove the conclusion (i=n => correct)  
  
    elif algorithm.type == "recursive":  
        # 1) Define the inductive hypothesis  
        #     (correct for inputs of sizes 1 to i)  
        # 2) Prove the base case (i < small constant)  
        # 3) Prove the inductive step (i => i+1 OR  
        #     {1,2,...,i} => i+1)  
        # 4) Prove the conclusion (i=n => correct)  
  
    # TODO
```

Proving Correctness

- Recall, there are four components in a proof by induction.
 - **Inductive Hypothesis** The algorithm works on input lists of length 1 to i.
 - **Base case** The algorithm works on input lists of length 1.
 - **Inductive step** If the algorithm works on input lists of length 1 to i, then it works on input lists of length i+1.
 - **Conclusion** If the algorithm works on input lists of length n, then it works on the entire list.

Proving Correctness

- Formally, for **select**...

Proving Correctness

- Formally, for **select**...
 - **Inductive Hypothesis** **select(A, k)** correctly finds the k^{th} -smallest element for inputs of length 1 to i .

Proving Correctness

- Formally, for **select**...
 - **Inductive Hypothesis** **select(A, k)** correctly finds the k^{th} -smallest element for inputs of length 1 to i .
 - **Base case** **select(A, k)** correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.

Proving Correctness

- Formally, for **select**...
 - **Inductive Hypothesis** **select(A, k)** correctly finds the k^{th} -smallest element for inputs of length 1 to i .
 - **Base case** **select(A, k)** correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.
 - **Inductive step**
Suppose the algorithm works on input lists of length 1 to i .
Calling **select(A, k)** on an input list of length $i+1$ selects a pivot, partitions around it, and compares the length of the left list to k .
There are three cases:

Proving Correctness

- Formally, for **select**...
 - **Inductive Hypothesis** **select(A, k)** correctly finds the k^{th} -smallest element for inputs of length 1 to i .
 - **Base case** **select(A, k)** correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.
 - **Inductive step**
Suppose the algorithm works on input lists of length 1 to i .
Calling **select(A, k)** on an input list of length $i+1$ selects a pivot, partitions around it, and compares the length of the left list to k .
There are three cases:
 - **len(left) == k**: exactly k items less than the pivot, so return the pivot.
 - **len(left) > k**: More than k items less than the pivot, so return the k^{th} -smallest element of the left half of the list.
 - **len(left) < k**: There are fewer than k items \leq to the pivot, so return the $(k - \text{len(left)} - 1)^{\text{st}}$ -smallest element of the right half of the list.

Proving Correctness

- Formally, for **select**...
 - **Inductive Hypothesis** **select(A, k)** correctly finds the k^{th} -smallest element for inputs of length 1 to i .
 - **Base case** **select(A, k)** correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.
 - **Inductive step**
Suppose the algorithm works on input lists of length 1 to i .
Calling **select(A, k)** on an input list of length $i+1$ selects a pivot, partitions around it, and compares the length of the left list to k .
There are three cases:
 - **len(left) == k**: exactly k items less than the pivot, so return the pivot.
 - **len(left) > k**: More than k items less than the pivot, so return the k^{th} -smallest element of the left half of the list.
 - **len(left) < k**: There are fewer than k items \leq to the pivot, so return the $(k - \text{len(left)} - 1)^{\text{st}}$ -smallest element of the right half of the list.
 - **Conclusion** The inductive hypothesis holds for all i . In particular, given an input list of any length n , **select(A, k)** correctly finds the k^{th} -smallest element!

Today's Outline

- Divide and Conquer II
 - Linear-time selection
 - ~~Proving correctness~~ Done!
 - Proving runtime with recurrence relations
 - *Problems: selection*
 - *Algorithms: Select*
 - Reading: CLRS 9

Analyzing Runtime

- Writing a recurrence relation for `select` gives:

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- Writing a recurrence relation for `select` gives:

$$T(n) = \begin{cases} O(n) & \text{len(left)} == k \\ T(\text{len(left)}) + O(n) & \text{len(left)} > k \\ T(\text{len(right)}) + O(n) & \text{len(left)} < k \end{cases}$$

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- Writing a recurrence relation for `select` gives:

$$T(n) = \begin{cases} O(n) & \text{len(left)} == k \\ T(\text{len(left)}) + O(n) & \text{len(left)} > k \\ T(\text{len(right)}) + O(n) & \text{len(left)} < k \end{cases}$$

↑
The runtime for the recursive call to `select`

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

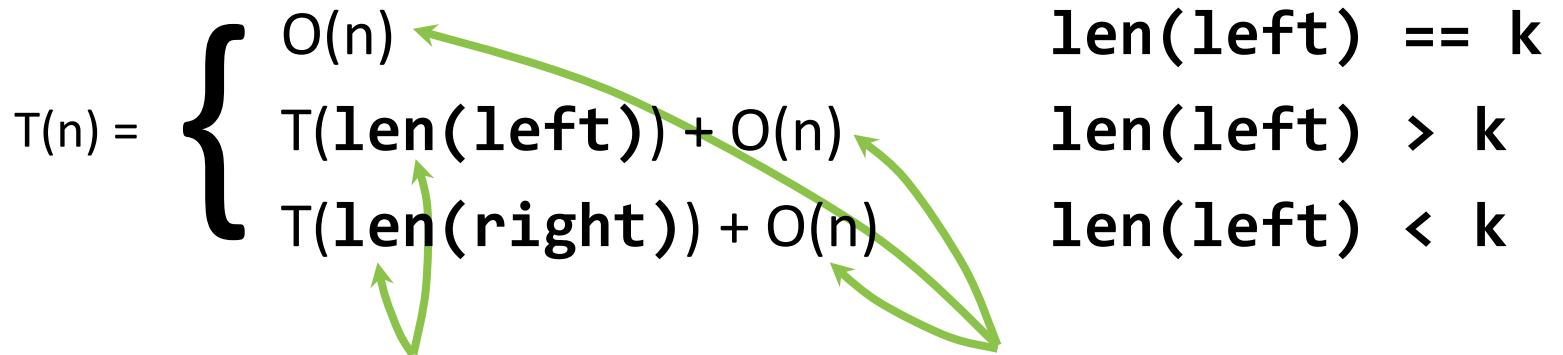
- Writing a recurrence relation for `select` gives:

$$T(n) = \begin{cases} O(n) \\ T(\text{len(left)}) + O(n) \\ T(\text{len(right)}) + O(n) \end{cases}$$

The runtime for the recursive call to `select`

The runtime to partition about the chosen pivot

len(left) == k
 len(left) > k
 len(left) < k

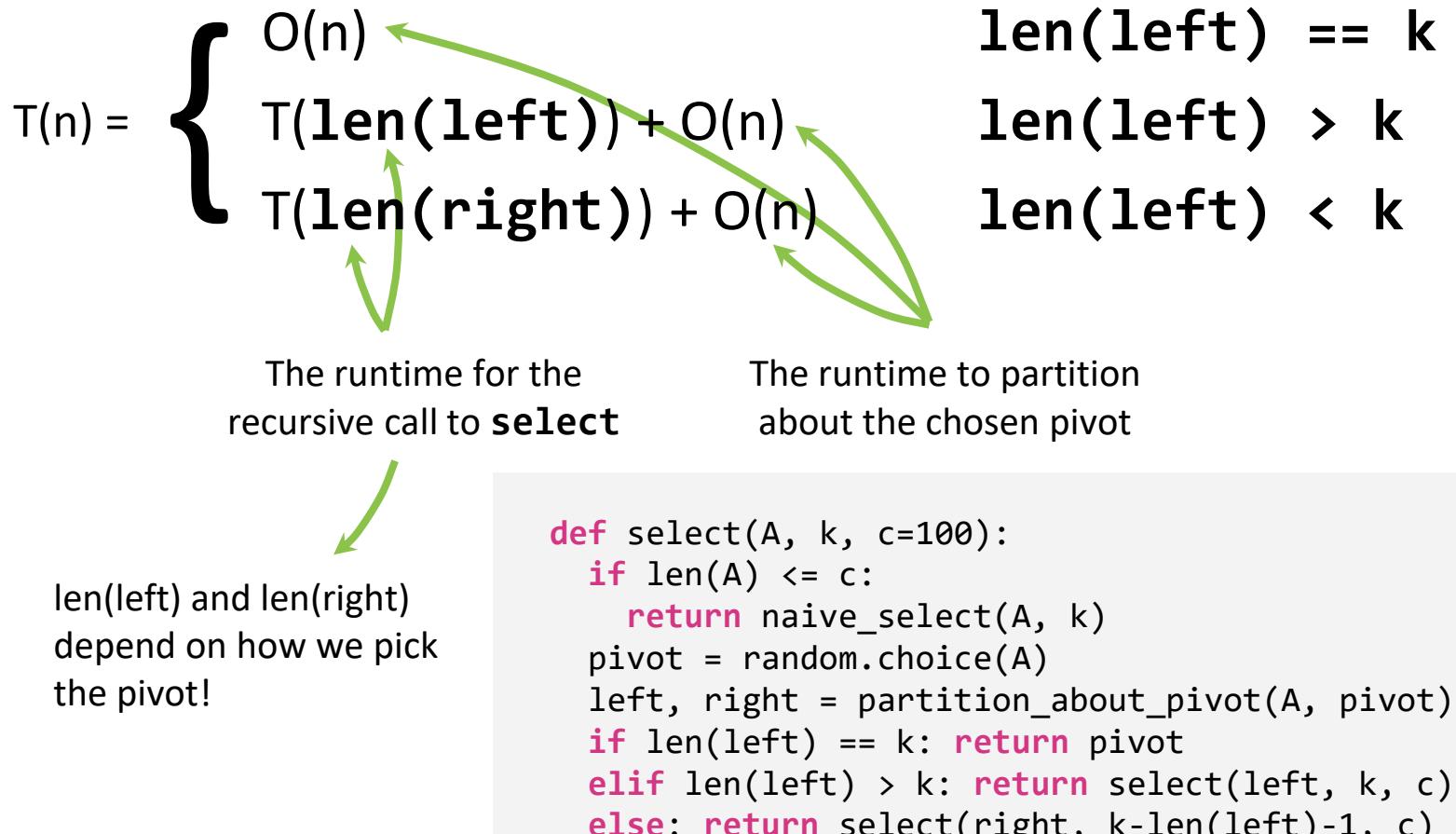


```

def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
  
```

Analyzing Runtime

- Writing a recurrence relation for `select` gives:



Analyzing Runtime in an Ideal World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In an ideal world, we split the input exactly in half, such that:
 $\text{len(left)} = \text{len(right)} = (n-1)/2$.
 - Then we could use **Master Theorem!**
 - What's the recurrence?

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Analyzing Runtime in an Ideal World

- **len(left)** and **len(right)** determine the runtime of the recursive calls to **select**.
 - In an ideal world, we split the input exactly in half, such that:
len(left) = len(right) = (n-1)/2.
 - Then we could use **Master Theorem!**
 - What's the recurrence? $T(n) \leq T(n/2) + O(n)$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Analyzing Runtime in an Ideal World

- **len(left)** and **len(right)** determine the runtime of the recursive calls to **select**.
 - In an ideal world, we split the input exactly in half, such that:
len(left) = len(right) = (n-1)/2.
 - Then we could use **Master Theorem!**
 - What's the recurrence? $T(n) \leq T(n/2) + O(n)$
 - Then, $a = 1$, $b = 2$, $d = 1$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Analyzing Runtime in an Ideal World

- **len(left)** and **len(right)** determine the runtime of the recursive calls to **select**.
 - In an ideal world, we split the input exactly in half, such that:
len(left) = len(right) = (n-1)/2.
 - Then we could use **Master Theorem!**
 - What's the recurrence? $T(n) \leq T(n/2) + O(n)$
 - Then, $a = 1$, $b = 2$, $d = 1$ (Case 2: $a < b^d$)

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Analyzing Runtime in an Ideal World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In an ideal world, we split the input exactly in half, such that:
 $\text{len(left)} = \text{len(right)} = (n-1)/2$.
 - Then we could use **Master Theorem!**
 - What's the recurrence? $T(n) \leq T(n/2) + O(n)$
 - Then, $a = 1$, $b = 2$, $d = 1$ (Case 2: $a < b^d$)
 - $T(n) \leq O(n^d) = O(n)$

pivotol best → 짧간값 어떤?

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

pivot이 짜고있을까? Analyzing Runtime

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - If we get super unlucky, we split the input, such that: `len(left) = n - 1` and `len(right) = 1` or vice versa.
 - Then it would be a lot slower.
 - $T(n) \leq T(n-1) + O(n)$
 - Then, $O(n)$ levels of $O(n)$
 - $T(n) \leq O(n^2)$

$$T(n) \leq Cn + C(n-1) + C(n-2) \dots + C$$

$$= C \frac{n(n+1)}{2} \leq O(n^2)$$

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

“Worst-case” runtime $\Theta(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

“Worst-case” runtime $\Theta(n^2)$

We discussed this
runtime from earlier!

Analyzing Runtime

- Recall `pivot = random.choice(A)` i.e. we randomly chose the pivot.
 - It's *possible* to get unlucky, thus leading to runtime of $\Theta(n^2)$. *worst*
 - We'll formalize this unluckiness when we study Randomized Algs.
- How might we pick a better pivot?
 - After all, it's called **linear-time** selection, which implies $\Theta(n)$ -time.

Analyzing Runtime

- Recall in an ideal world, we split the input exactly in half, such that:
 $\text{len(left)} = \text{len(right)} = (n-1)/2$. → 2분할이면 linear 가능
- **Key Insight** The ideal world requires us to pick the pivot that divides the input list in half

Analyzing Runtime

- Recall in an ideal world, we split the input exactly in half, such that:
 $\text{len(left)} = \text{len(right)} = (n-1)/2$.
- **Key Insight** The ideal world requires us to pick the pivot that divides the input list in half aka **the median**

Analyzing Runtime

- Recall in an ideal world, we split the input exactly in half, such that:
 $\text{len(left)} = \text{len(right)} = (n-1)/2$.
- **Key Insight** The ideal world requires us to pick the pivot that divides the input list in half aka **the median** aka **`select(A, k=[n/2]-1)`**.
- To approximate the ideal world, the linear-time select algorithm picks the pivot that divides the input list **approximately** in half aka **close to the median**.

Reasonable Analyzing Runtime in an ~~Ideal~~ World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.
 - Once again, we could use **Master Theorem!**
 - What's the recurrence?

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Reasonable Analyzing Runtime in an ~~Ideal~~ World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.
 - Once again, we could use **Master Theorem**!
 - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Reasonable Analyzing Runtime in an ~~Ideal~~ World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.
 - Once again, we could use **Master Theorem**!
 - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$
 - Then, $a = 1$, $b = 10/7$, $d = 1$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Reasonable Analyzing Runtime in an ~~Ideal~~ World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.
 - Once again, we could use **Master Theorem!**
 - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$
 - Then, $a = 1$, $b = 10/7$, $d = 1$ (Case 2: $a < b^d$)

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Reasonable Analyzing Runtime in an ~~Ideal~~ World

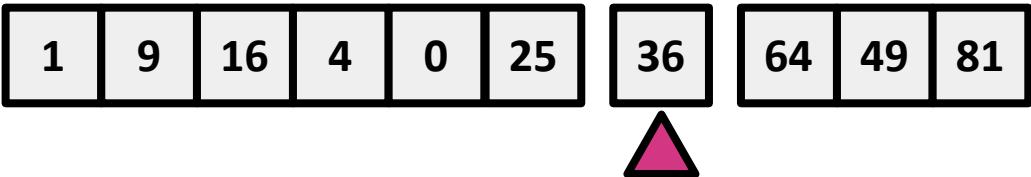
- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.
 - Once again, we could use **Master Theorem**!
 - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$
 - Then, $a = 1$, $b = 10/7$, $d = 1$ (Case 2: $a < b^d$)
 - $T(n) \leq O(n^d) = O(n)$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

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

Reasonable Analyzing Runtime in an ~~Ideal~~ World

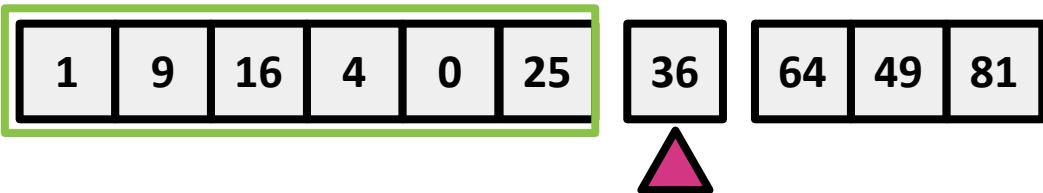
- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.



The **goal** is to pick a pivot such that

Reasonable Analyzing Runtime in an ~~Ideal~~ World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.

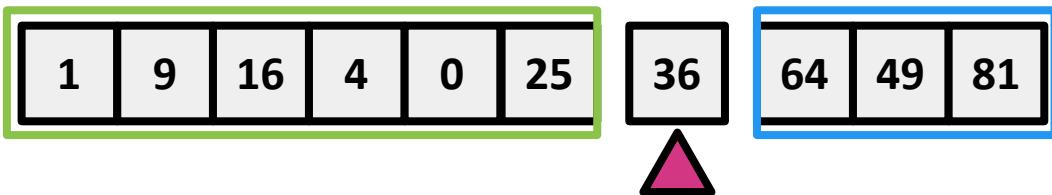


The **goal** is to pick a pivot such that

$$3n/10 < \text{len(left)} < 7n/10$$

Reasonable Analyzing Runtime in an ~~Ideal~~ World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
 - In a reasonable world, we split the input roughly in half, such that:
 $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.



The **goal** is to pick a pivot such that

$$3n/10 < \text{len(left)} < 7n/10 \text{ and } 3n/10 < \text{len(right)} < 7n/10$$

Another Divide and Conquer Algorithm

- We can't solve **select(A, n/2)** (yet).
- But we can solve **select(B, m/2)** for **len(B) = m < n**.
- How does having an algorithm that can find the median of smaller lists help us?

Another Divide and Conquer Algorithm

- We can't solve **select(A, n/2)** (yet).
- But we can solve **select(B, m/2)** for **len(B) = m < n**.
- How does having an algorithm that can find the median of smaller lists help us?



Pro tip: making the inductive hypothesis i.e. assuming correctness of the algorithm on smaller inputs is a helpful technique for designing divide and conquer algorithms.

Another Divide and Conquer Algorithm

- We can't solve **select(A, n/2)** (yet).
- But we can solve **select(B, m/2)** for **len(B) = m < n**.
- How does having an algorithm that can find the median of smaller lists help us? **It can help us pick a pivot that's close to the median.**



Pro tip: making the inductive hypothesis i.e. assuming correctness of the algorithm on smaller inputs is a helpful technique for designing divide and conquer algorithms.

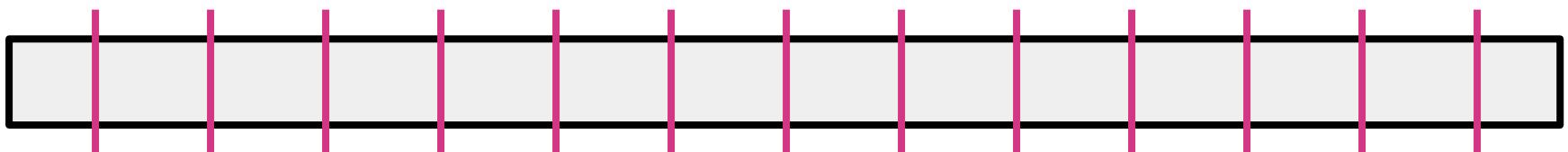
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.



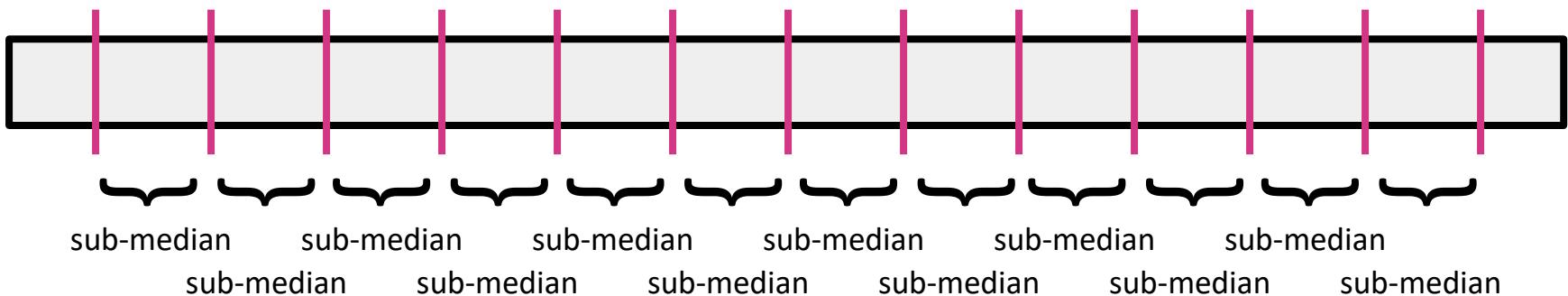
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
 - Divide the original list into small groups.



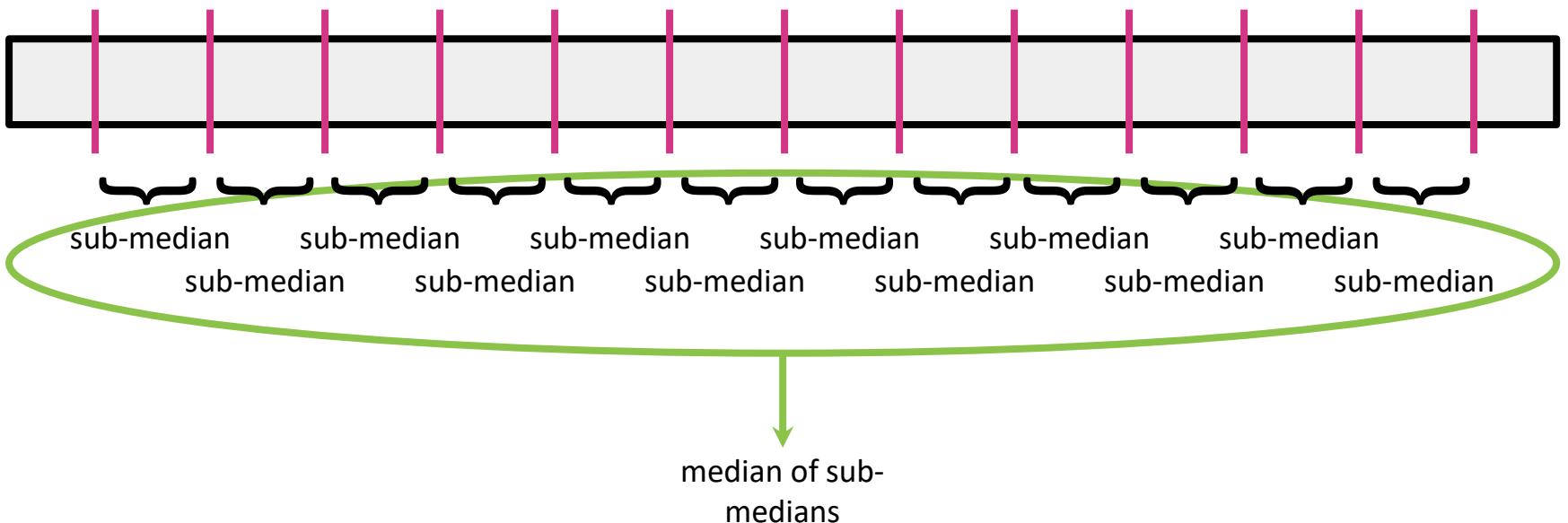
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
 - Divide the original list into small groups.
 - Find the sub-median of each small group.



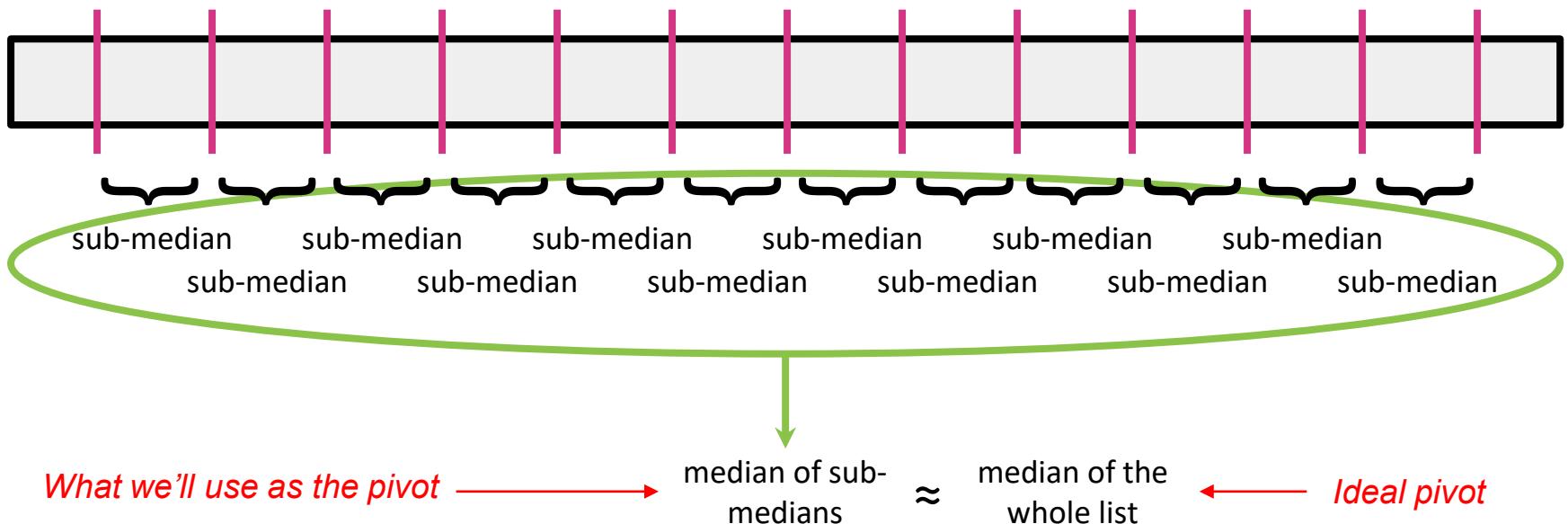
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
 - Divide the original list into small groups.
 - Find the sub-median of each small group.
 - Find the median of all of the sub-medians.



Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
 - Divide the original list into small groups.
 - Find the sub-median of each small group.
 - Find the median of all of the sub-medians.

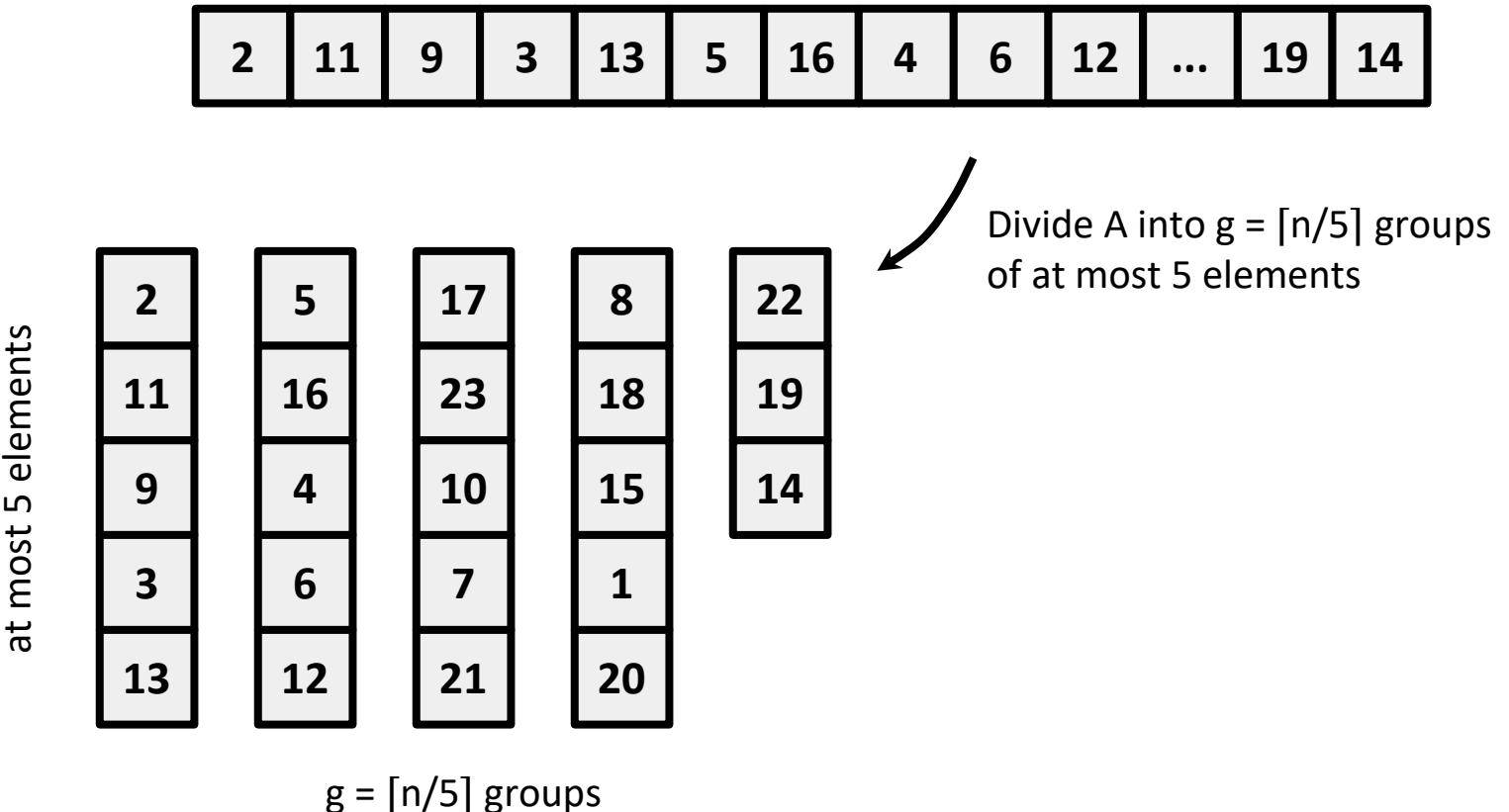


Lemma: The median of sub-medians is close to the median.

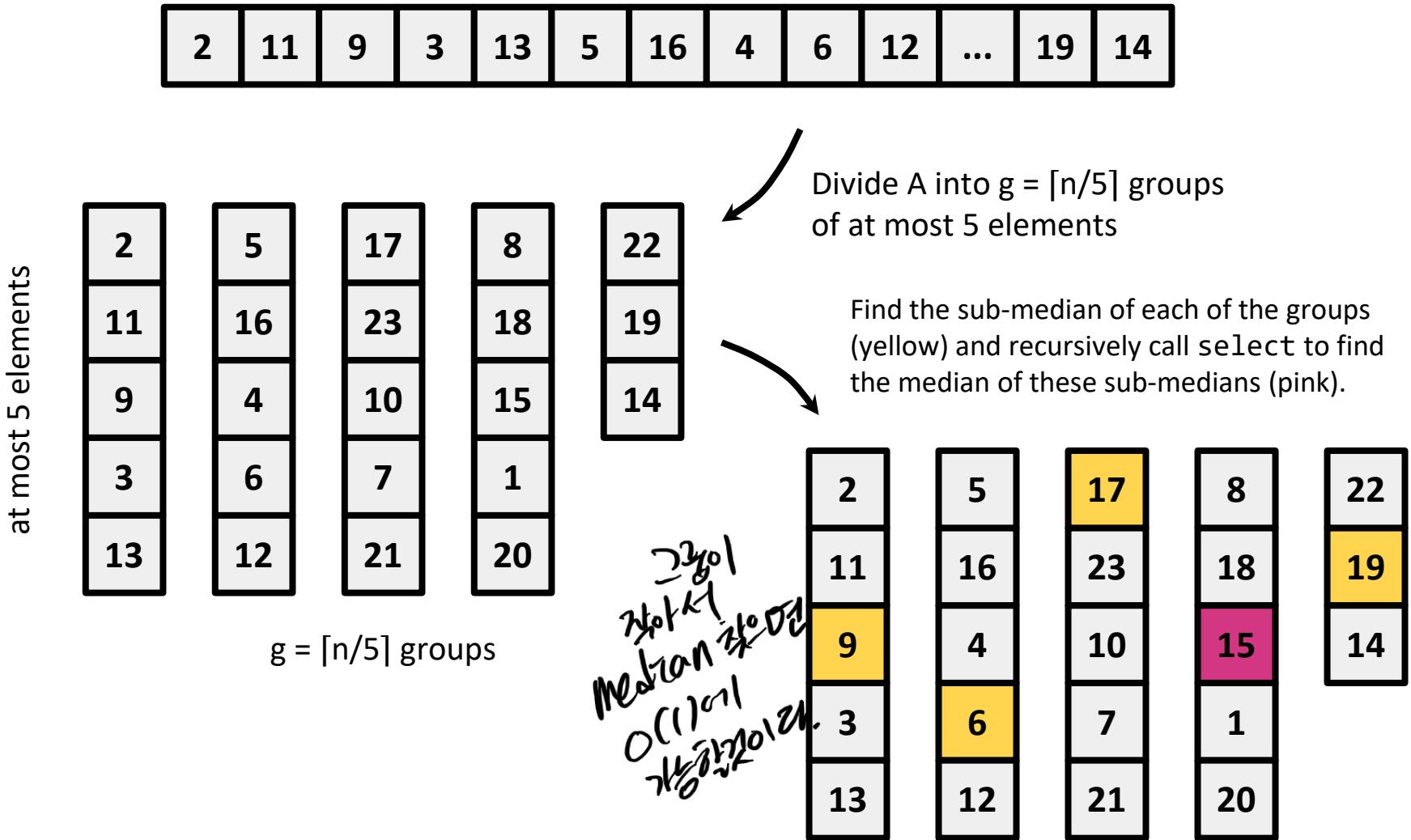
Another Divide and Conquer Algorithm

2	11	9	3	13	5	16	4	6	12	...	19	14
---	----	---	---	----	---	----	---	---	----	-----	----	----

Another Divide and Conquer Algorithm



Another Divide and Conquer Algorithm



How to pick the pivot

- median_of_medians(A):

- Split A into $m = \lceil \frac{n}{5} \rceil$ groups, of size ≤ 5 each.

- For $i=1, \dots, m$:

- Find the median within the i 'th group, call it p_i

- $p = \text{SELECT}([p_1, p_2, p_3, \dots, p_m], m/2)$

- return p

 $O(n)$
 $O(n)$
 $m = \lceil \frac{n}{5} \rceil$

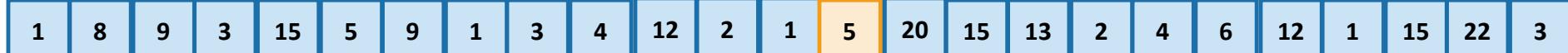
$$T(m) = T\left(\frac{n}{10}\right)$$

8

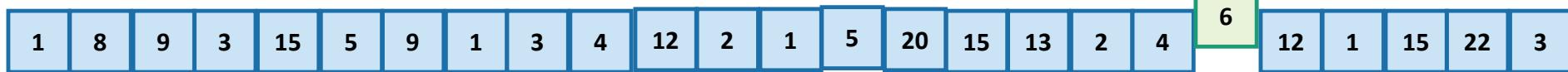
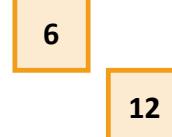
This takes time $O(1)$, for each group, since each group has size 5. So that's $O(m)$ total in the for loop.

4

$\lceil \frac{n}{5} \rceil$



Pivot is $\text{SELECT}([8, 4, 5, 6, 12], 3) = 6$:



PARTITION around that 6:

6

This part is R: it's almost the same size as L.

This part is L

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

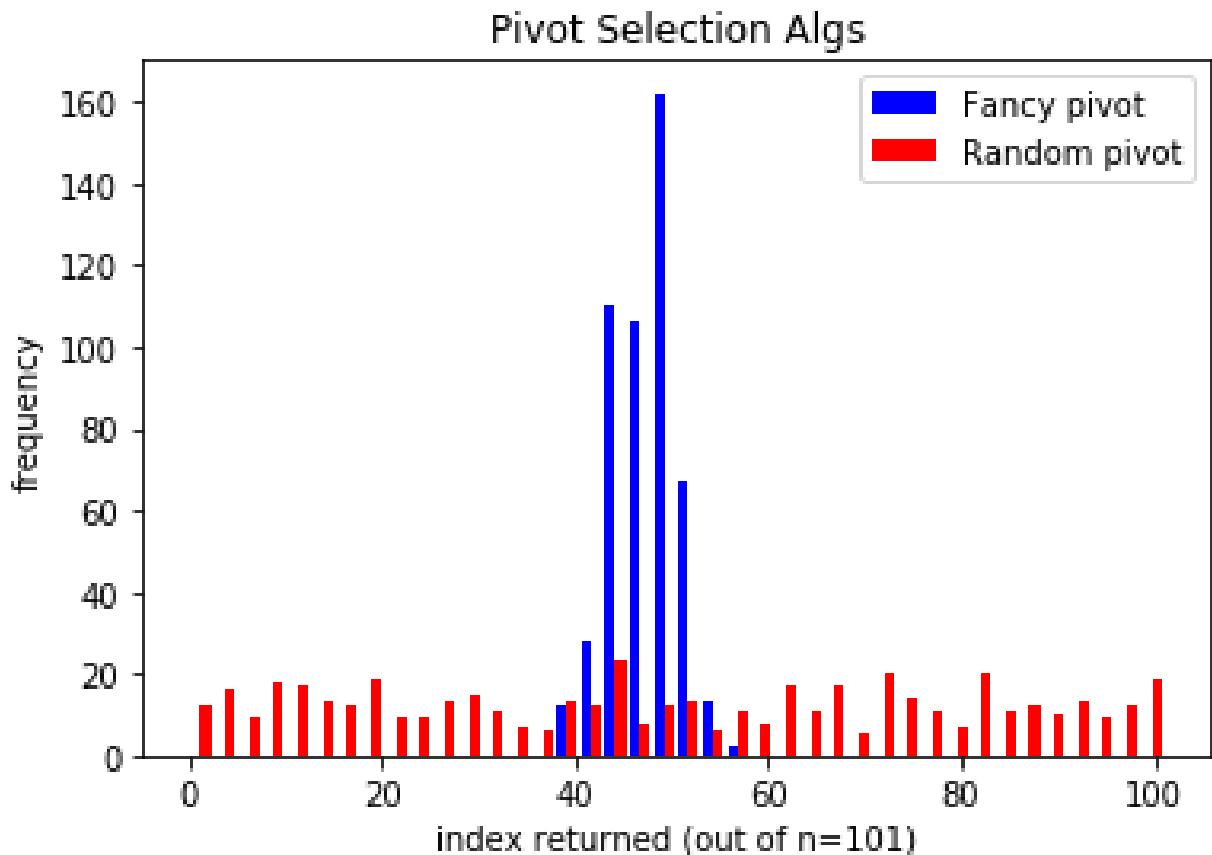
“Worst-case” runtime $\Theta(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A) median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

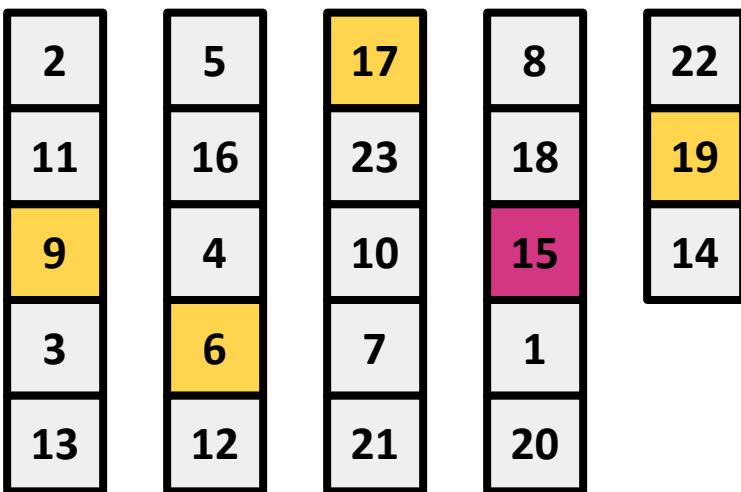
Analyzing Runtime

- Emprically,



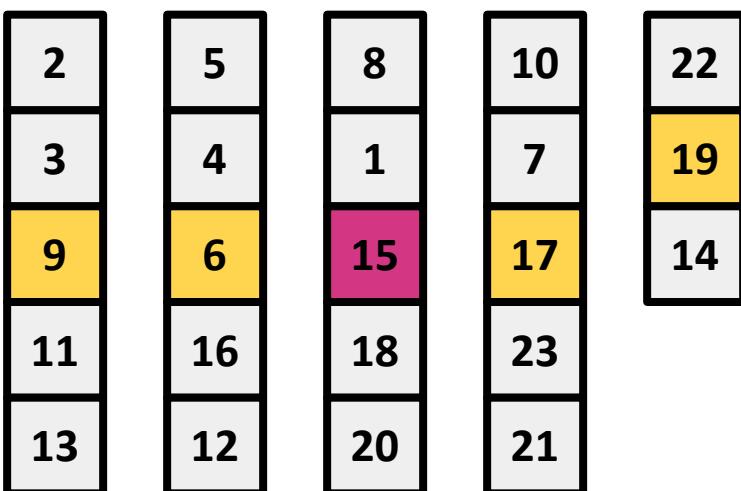
Analyzing Runtime

- Clearly, the median of medians (15) is not necessarily the actual median (12), but we claim that it's guaranteed to be pretty close.



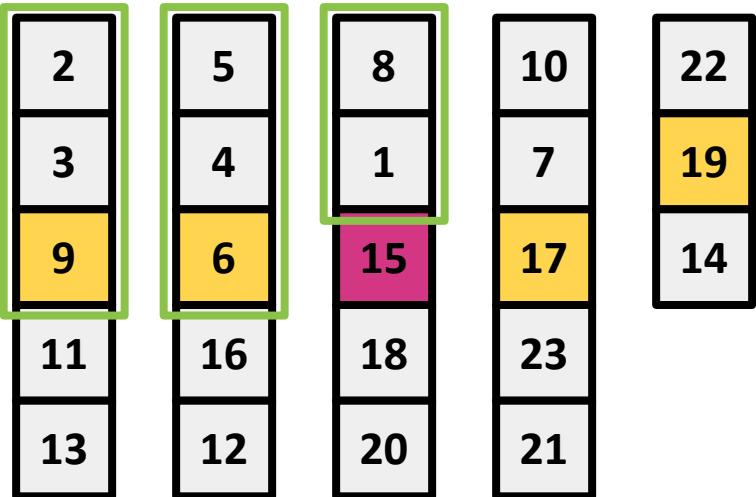
Analyzing Runtime

- To see why, partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.
 - **At least** how many elements are guaranteed to be **smaller** than the median of medians?



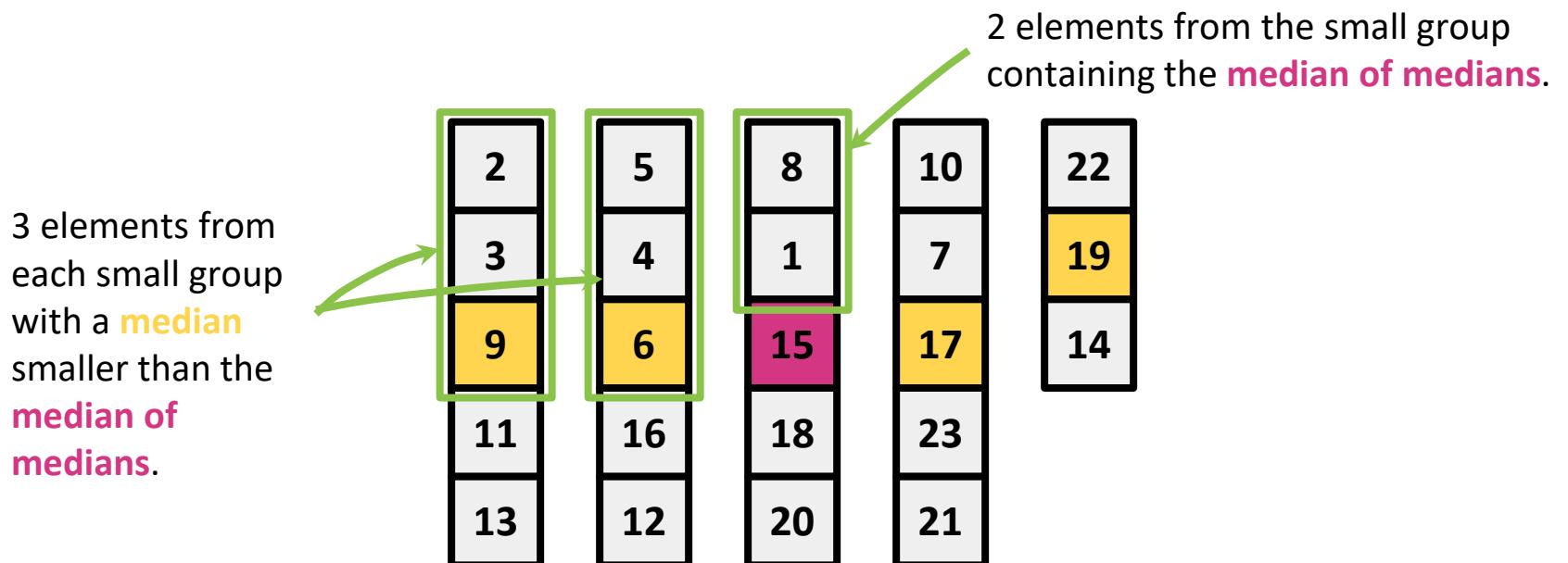
Analyzing Runtime

- To see why, partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.
 - At least** how many elements are guaranteed to be smaller than the median of medians? **At least** **these (1, 2, 3, 4, 5, 6, 8, 9)**. There might be more (7, 11, 12, 13, 14), but we are *guaranteed* that at least these will be smaller.



Analyzing Runtime

- To see why, partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.
 - At least** how many elements are guaranteed to be smaller than the median of medians? **At least** these (1, 2, 3, 4, 5, 6, 8, 9). There might be more (7, 11, 12, 13, 14), but we are *guaranteed* that at least these will be smaller.

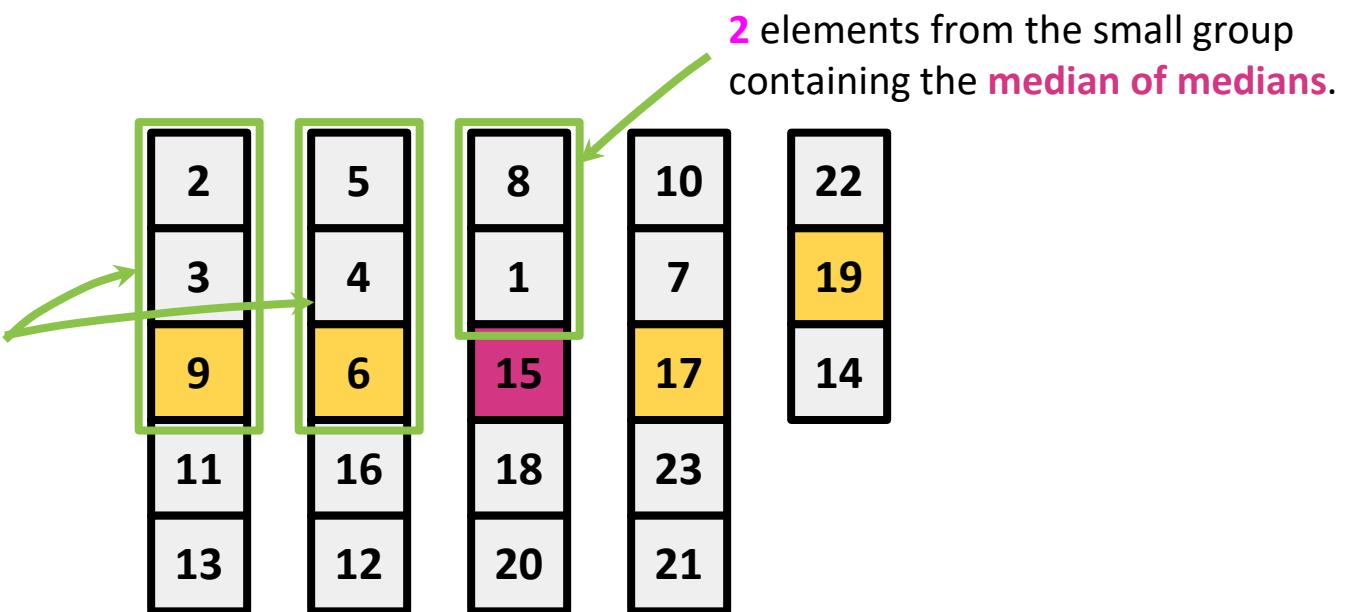


Analyzing Runtime

- As a function of n (the size of the original list), how many elements are guaranteed to be **smaller** than the median of medians?
 - Let $g = \lceil n/5 \rceil$ represent the number of groups.
 - At least** $3 \cdot (\lceil g/2 \rceil - 1) + 2$ elements.

To exclude the list
with the **median
of medians**.

3 elements from
each small group
with a **median**
smaller than the
**median of
medians**.



Analyzing Runtime

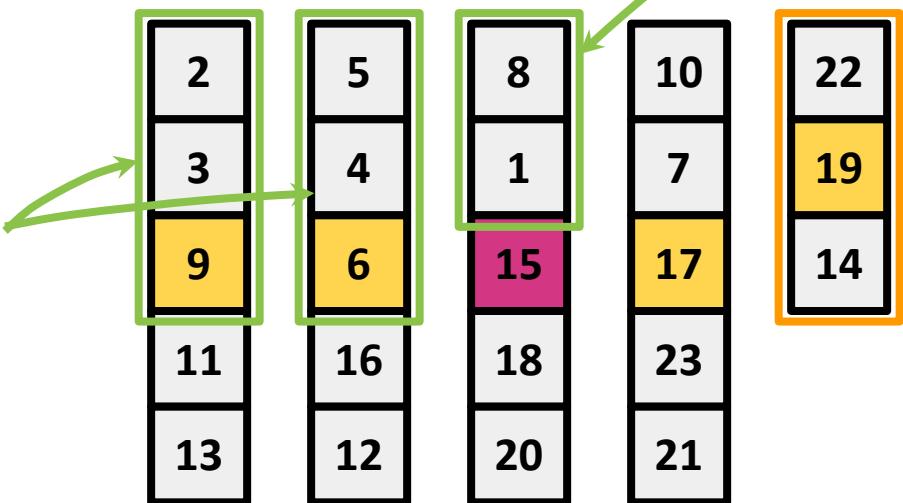
- As a function of n (the size of the original list), how many elements are guaranteed to be **smaller** than the median of medians?
 - Let $g = \lceil n/5 \rceil$ represent the number of groups.
 - At least** $3 \cdot (\lceil g/2 \rceil - 1) + 2$ elements.

To exclude the list with the **median of medians**.

To exclude the list with the **leftovers**.

3 elements from each small group with a **median** smaller than the **median of medians**.

2 elements from the small group containing the **median of medians**.

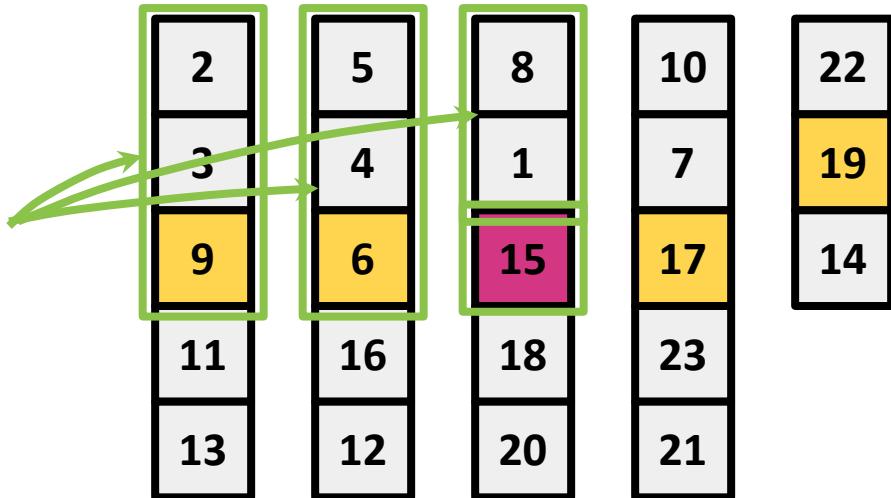


Analyzing Runtime

- If at least $3 \cdot (\lceil g/2 \rceil - 2) + 2$ elements are guaranteed to be smaller than the median of medians, at most how many elements are larger than the median of medians?
 - At most $n - 1 - (3 \cdot (\lceil g/2 \rceil - 2) + 2)$

n-1 is for all of the elements except for the median of medians.

At most everything besides these elements

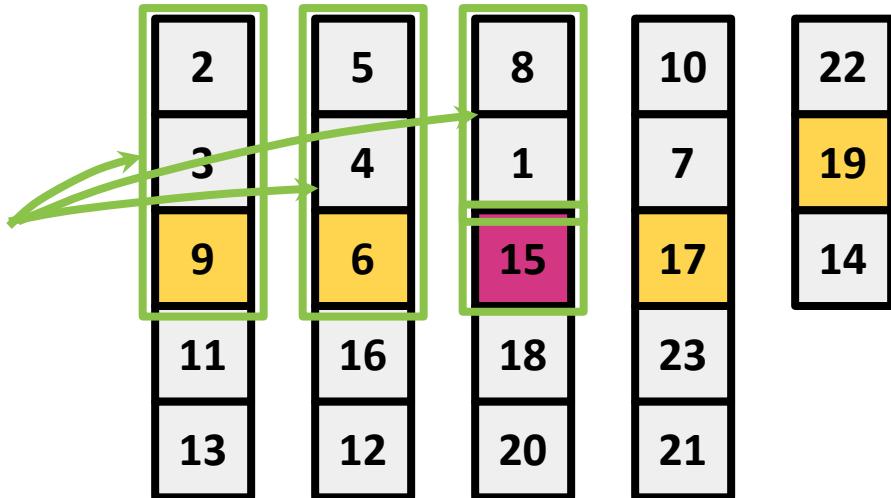


Analyzing Runtime

- If at least $3 \cdot (\lceil g/2 \rceil - 2) + 2$ elements are guaranteed to be smaller than the median of medians, at most how many elements are larger than the median of medians?
 - At most $n - 1 - (3 \cdot (\lceil g/2 \rceil - 2) + 2) \leq 7n/10 + 3$ elements.

$n-1$ is for all of the elements except for the median of medians.

At most everything besides these elements



Analyzing Runtime

- We just showed that ...

$$3n/10 - 4 \leq \text{len(left)}$$

$$\text{len(right)} \leq 7n/10 + 3$$

median_of_medians will choose a pivot greater than at least $3 \cdot (\lceil g/2 \rceil - 2) + 2 \geq 3n/10 - 4$ elements.

median_of_medians will choose a pivot less than at most $7n/10 + 3$ elements.

Analyzing Runtime

- We just showed that ...

$$3n/10 - 4 \leq \text{len(left)} \leq 7n/10 + 3$$

$$3n/10 - 4 \leq \text{len(right)} \leq 7n/10 + 3$$

median_of_medians will choose a pivot greater than at least $3 \cdot (\lceil g/2 \rceil - 2) + 2 \geq 3n/10 - 4$ elements.

- We can just as easily show the inverse.

median_of_medians will choose a pivot less than at most $7n/10 + 3$ elements.

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A) median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
 - $T(n) = n \log(n)$ when $n \leq 100$
 - $T(n) \leq T(n/5) +$

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

$T\left(\frac{n}{5}\right) + O(n)$

Analyzing Runtime

- What's the recurrence relation?
 - $T(n) = n \log(n)$ when $n \leq 100$
 - $T(n) \leq T(n/5) + T(7n/10) +$

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

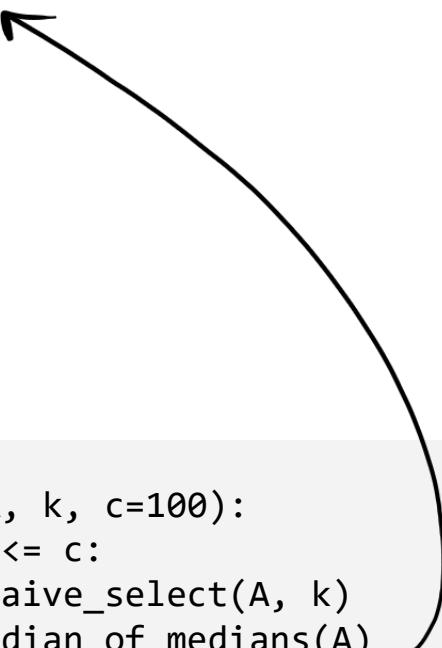
$$\frac{3}{7}n / \frac{1}{10}n$$

Analyzing Runtime

- What's the recurrence relation?

- $T(n) = n \log(n)$ when $n \leq 100$
- $T(n) \leq T(n/5) + T(7n/10) + O(n)$

$\frac{\text{pivot}}{\text{分割}}$ $\frac{\text{select(left)}}{\text{선택}}$



```

def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
  
```

Analyzing Runtime

- What's the recurrence relation?
 - $T(n) = n \log(n)$ when $n \leq 100$
 - $T(n) \leq T(n/5) + T(7n/10) + O(n)$

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
 - $T(n) = n \log(n)$ when $n \leq 100$
 - $T(n) \leq T(n/5) + T(7n/10) + O(n)$
 - We can't use **Master Theorem!**

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?

- $T(n) = n \log(n)$ when $n \leq 100$
- $T(n) \leq T(n/5) + T(7n/10) + O(n)$
- We can't use **Master Theorem!**
- We use **substitution method!**

subproblem \exists 가 대상

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

Substitution Method

$$T(n) = n \log(n) \text{ when } n \leq 100$$

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

1. Guess what the answer is.

- Linear-time select
- Comparing to mergesort recurrence, less than $n \log(n)$



Substitution Method

$$T(n) = n \log(n) \text{ when } n \leq 100$$

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \leq Ck$ for all $1 \leq k < n$.
- **Base case** $T(k) \leq Ck$ for all $k \leq 100$. C is some constant we'll have to fill in later!
- **Inductive step**
 - $$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + dn \\ &\leq C(n/5) + C(7n/10) + dn \\ &= (C/5)n + (7C/10)n + dn \\ &\leq Cn \end{aligned}$$
 C must be $\geq \log(n)$ for $n \leq 100$, so $C \geq 7$.
- **Conclusion** There exists some $C = \max\{7, 10d\}$ such that for all $n > 1$, $T(n) \leq Cn$. Therefore, $T(n) = O(n)$. Solve for C to satisfy the inequality. $C \geq 10d$ works.

Substitution Method

$$T(n) = n \log(n) \text{ when } n \leq 100$$

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \leq \max\{7, 10d\}k$ for all $1 \leq k < n$.
- **Base case** $T(k) \leq \max\{7, 10d\}k$ for all $k \leq 100$.
- **Inductive step**
 - $$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + dn \\ &\leq \max\{7, 10d\}(n/5) + \max\{7, 10d\}(7n/10) + dn \\ &= (\max\{7, 10d\}/5)n + (7\max\{7, 10d\}/10)n + dn \\ &\leq \max\{7, 10d\}n \end{aligned}$$
- **Conclusion** There exists some $C = \max\{7, 10d\}$ such that for all $n > 1$, $T(n) \leq \max\{7, 10d\}n$. Therefore, $T(n) = O(n)$.

Substitution Method

1. Guess what the answer is.
1. Formally prove that's what the answer is.
 - Might need to leave some constants unspecified until the end and see what they need to be for the proof to work.

Today's Outline

- Divide and Conquer II
 - Linear-time selection
 - ~~Proving correctness~~ Done!
 - ~~Proving runtime with recurrence relations~~ Done!
 - *Problems: selection*
 - *Algorithms: Select*
 - Reading: CLRS 9

Linear-Time Selection

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A) median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

Worst-case runtime $\Theta(n)$

Linear-Time Selection

```

def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A) median_of_medians(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
  
```

Note: back to talking about the same worst-case we saw for insertion sort (we'll revisit during Randomized Algs).

Worst-case runtime $\Theta(n)$



Graduate School

The entrance to graduate school marks
a critical phase of transition
for most graduate students

from

knowledge to

knowledge

– Lui Sha, UIUC CS Professor



Graduate School

The entrance to graduate school marks
a critical phase of transition
for most graduate students

from absorbing knowledge to creating knowledge

– Lui Sha, UIUC CS Professor



Linear-Time Selection

- **Task** Find the k^{th} smallest element in an unsorted list in $O(n)$ -time.

1	64	9	49	16	4	0	25	36	81
---	----	---	----	----	---	---	----	----	----

SELECT(A,k): return the $k^{\text{'th}}$ smallest element in A

SELECT(A,0)=MIN(A)

SELECT(A,n/2-1)=MEDIAN(A)

SELECT(A,n-1)=MAX(A)

Linear-Time Selection

- **Finding the min and max**

Iterate through the list and keep track of the smallest and largest elements.

Runtime $O(n)$.

- **Finding the k^{th} smallest element (naive)**

Sort the list and return the element in index k of the sorted list.

Runtime $O(n \log(n))$.

$O(n)$ 은 가능한가? → divide & conquer
이해!

1	64	9	49	16	4	0	25	36	81
---	----	---	----	----	---	---	----	----	----

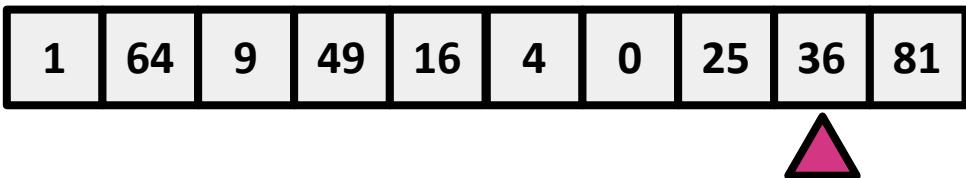
$k=3$



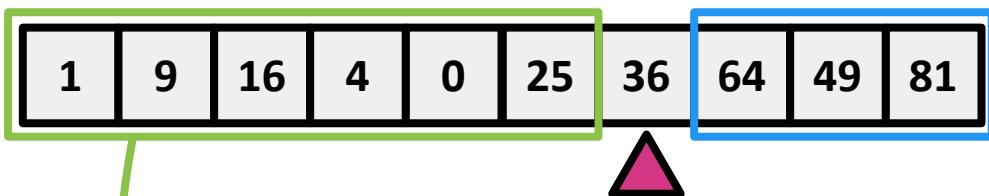
0	1	4	9	16	25	36	49	64	81
---	---	---	---	----	----	----	----	----	----

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
 - Suppose we want to find element $k=3$.



Select a pivot at random (for now)



Partition around the pivot, such that all elements to the left are less than it and all elements to the right are greater than it
 (Notice that the halves remain unsorted.)

Find element $k=3$ in this half since 36 occupies index 6 and $k=3 < 6$.

Analyzing Runtime

- Writing a recurrence relation for `select` gives:

$$T(n) = \begin{cases} O(n) & \text{len(left)} == k \\ T(\text{len(left)}) + O(n) & \text{len(left)} > k \\ T(\text{len(right)}) + O(n) & \text{len(left)} < k \end{cases}$$

*len(left) and len(right)
depend on how we
pick the pivot*

running time
pivot이 어떤 위치인가

```
def select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k: return pivot
    elif len(left) > k: return select(left, k, c)
    else: return select(right, k-len(left)-1, c)
```

easiest way

How to pick the pivot?

- Idea #1: choose a random pivot

- Unlucky case: $\text{len(left)} = n - 1$ and $\text{len(right)} = 1$ or vice versa
- $T(n) \leq T(n-1) + O(n)$
- Worst-case runtime $\Theta(n^2)$

- Idea #2: choose a pivot that divides the input list in half (the median)

- $\text{len(left)} = \text{len(right)} = (n-1)/2$
- $T(n) \leq T(n/2) + O(n)$
- Worst-case runtime $\Theta(n)$

○ We do not know how to find the median in linear time



- Idea #3: find a pivot “close enough” to median

- $3n/10 < \text{len(left)}, \text{len(right)} < 7n/10$.
- $T(n) \leq T(7n/10) + O(n)$
- Worst-case runtime $\Theta(n)$

CSE301 Introduction to Algorithms

Linear-Time Sorting

Fall 2022



Instructor : Hoon Sung Chwa

Today's Outline

- Linear-Time Sorting
 - Comparison-based sorting lower bounds
 - Algorithms: Counting sort, bucket sort, and radix sort
 - Reading: CLRS 8.1-8.2

Linear-Time Sorting

Sorting

- We've seen a few sorting algorithms
 - Insertion sort is worst-case $\Theta(n^2)$ -time.
 - Mergesort is worst-case $\Theta(n \log(n))$ -time.
- Can we do better?

Comparison-Based Sorting

- Comparison-based algorithms use “comparisons” to achieve their output.
 - insertion_sort** and **merge_sort** are comparison-based sorting algorithms.
 - Linear-time **select** is a comparison-based algorithm.
 - Later, we’ll see a randomized comparison-based sorting algorithm called **quick_sort**.

lower bound

4	3	1	5	2
---	---	---	---	---

• :

3	4	1	5	2
---	---	---	---	---

• :

Is 3 < 4?

A few comparisons that
insertion_sort makes.

Is 1 < 4? Is 1 < 3?

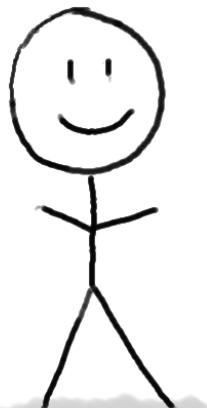
Comparison-Based Sorting

- Suppose we want to sort three items



Sort these three things.

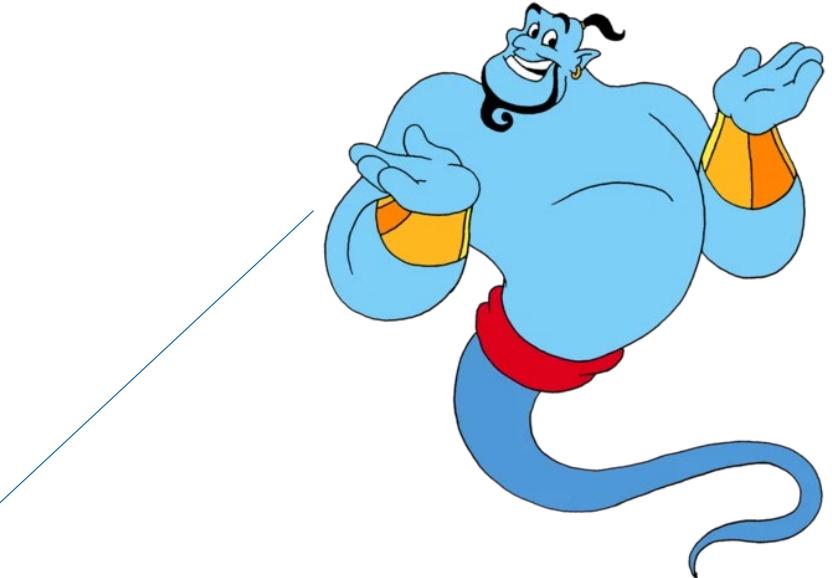
Is bigger than ?



Algorithm

YES

The algorithm's job is to output a correctly sorted list of all the objects.



There is a **genie** who knows what the right order is.

The genie can answer YES/NO questions of the form:
is [this] bigger than [that]?

Comparison-Based Sorting

- **Theorem** [Lower bound of $\Omega(n \log(n))$]:

Any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$ -time

- How to prove this?

1. Consider all comparison-based algorithms, one-by-one, and analyze them.
2. Don't do that.

Instead, argue that all comparison-based sorting algorithms produce a decision tree.
Then analyze decision trees.

Decision Tree

- Represent all comparisons as a **decision tree**



Sort these three things.



$$\text{😊} \leq \text{🚒} ?$$

YES

NO

etc...

$$\text{☕} \leq \text{😊} ?$$

YES

NO



$$\text{☕} < \text{🚒} ?$$

YES

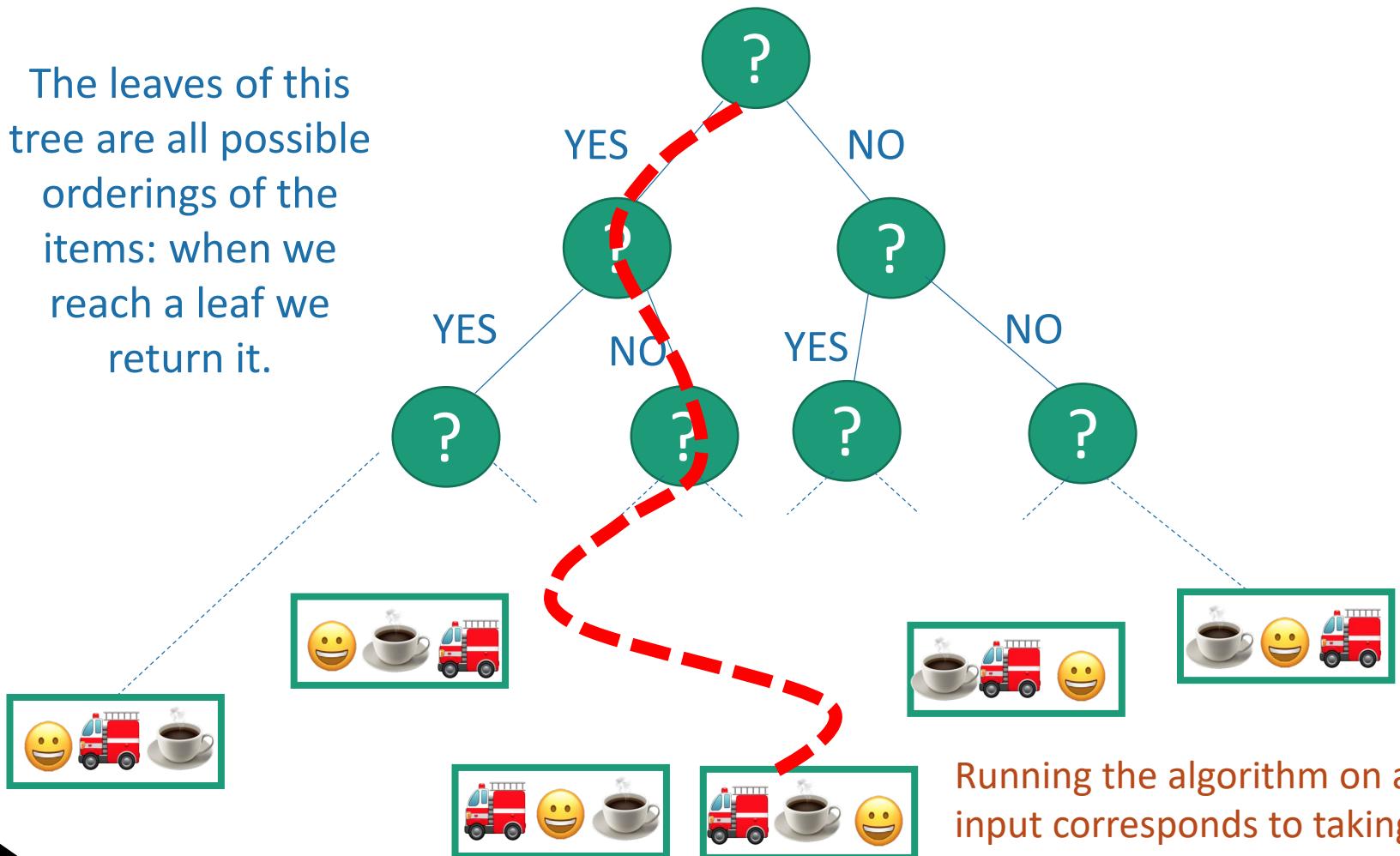
NO



Decision Tree

- All comparison-based algorithms have an associated decision tree

The leaves of this tree are all possible orderings of the items: when we reach a leaf we return it.

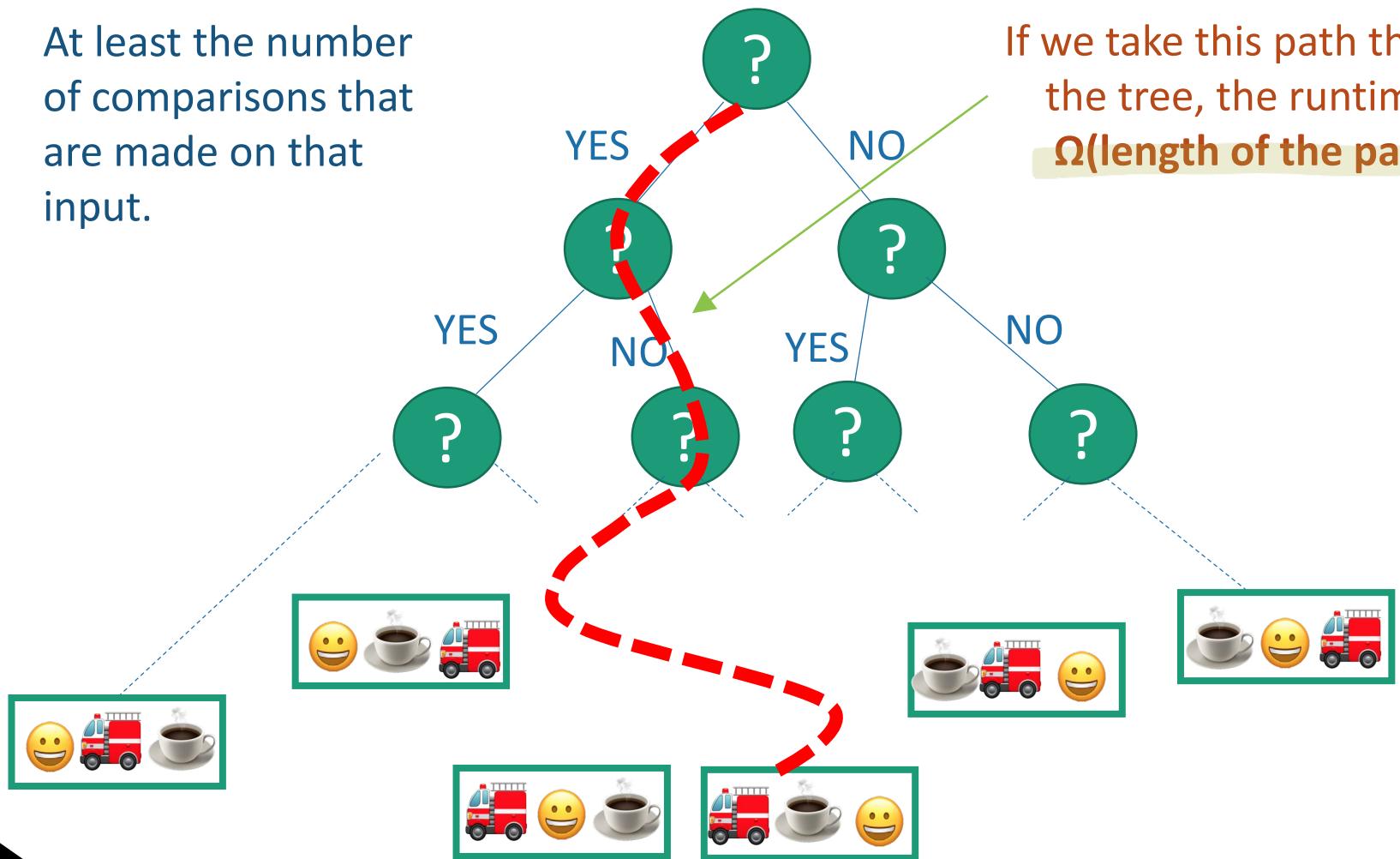


Decision Tree

- Q. What is the runtime on a **particular input?**

At least the number of comparisons that are made on that input.

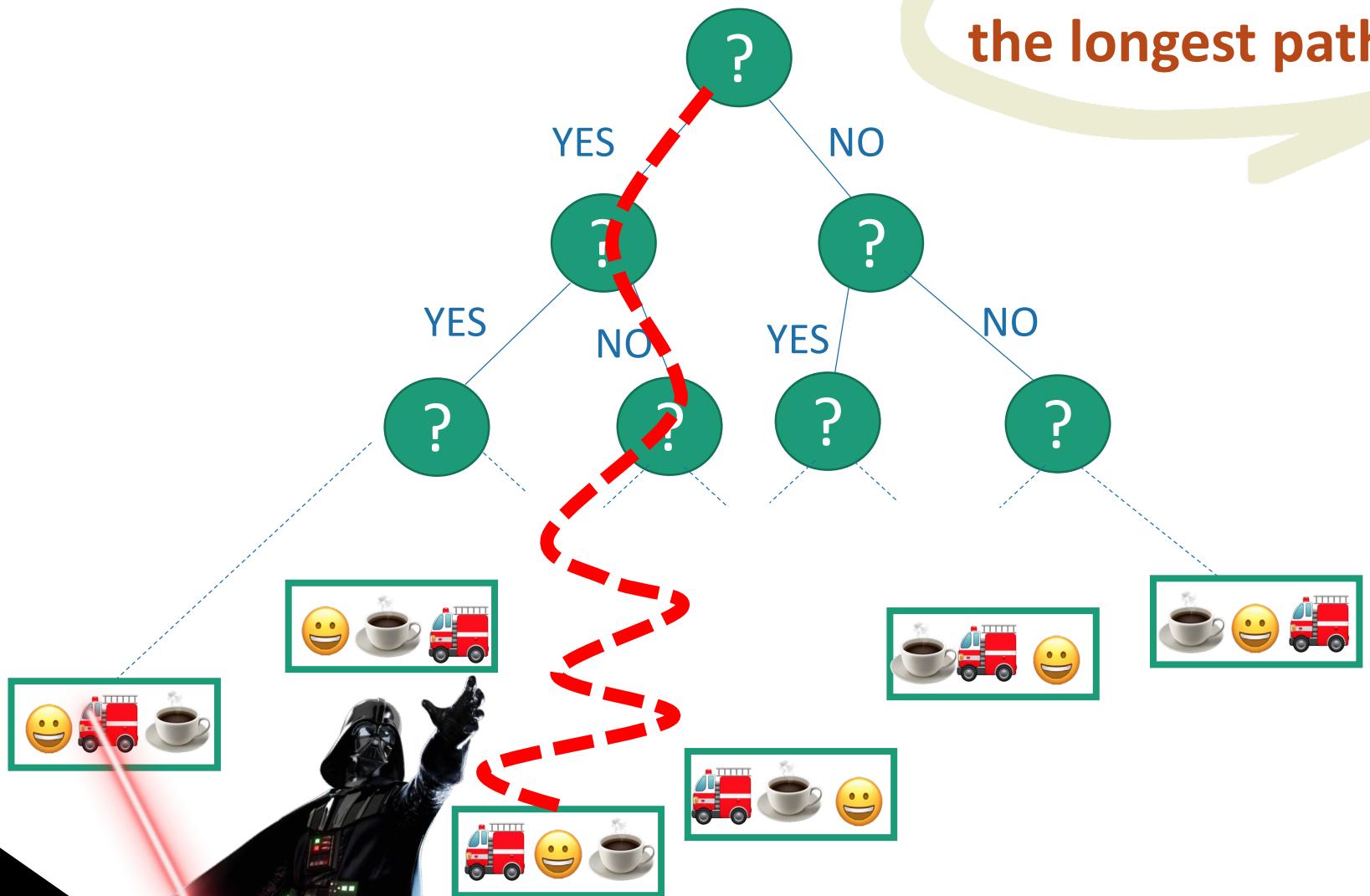
If we take this path through the tree, the runtime is **$\Omega(\text{length of the path})$** .



Decision Tree

- Q. What is the **worst-case runtime**?

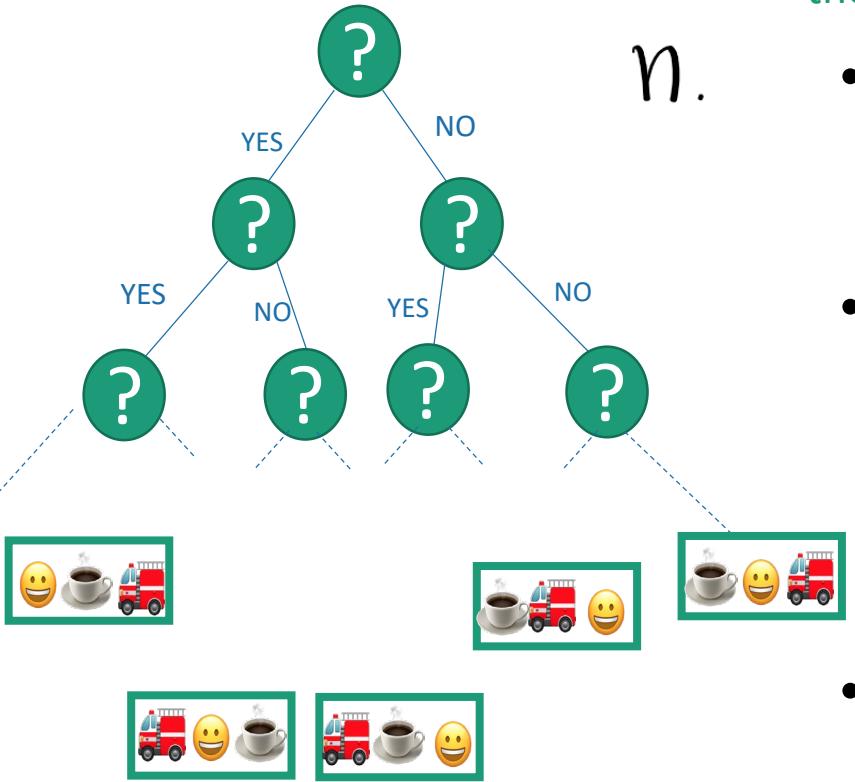
At least $\Omega(\text{length of the longest path})$



Decision Tree

- Q. How long is the longest path?

$n.$



We want a statement: in all such trees, the longest path is at least _____

- This is a **binary** tree with at least $n!$ leaves.
minimized sorting method
- The shallowest tree with $n!$ leaves is the completely balanced one, which has depth $\log(n!)$.
- So in all such trees, the longest path is at least **$\log(n!)$** .

- $\log(n!)$ is about $n \log(n/e) = \Omega(n \log(n))$.

Conclusion: the longest path has length at least $\Omega(n \log(n))$.

Comparison-Based Sorting

- **Theorem** [Lower bound of $\Omega(n \log(n))$]:

Any deterministic comparison-based sorting algorithm requires $\underline{\Omega(n \log(n))}$ -time
최악의 시간

- Proof:

- Any deterministic comparison-based algorithm can be represented as a decision tree with $n!$ leaves
- The worst-case running time is the depth of the decision tree
- All decision trees with $n!$ leaves have depth at least $\underline{\Omega(n \log(n))}$
- So any comparison-based sorting algorithm must have worst-case running time at least $\Omega(n \log(n))$

Is Linear-Time Sorting Nonsense?

- If any deterministic comparison-based sorting algorithm requires $\Omega(n \log(n))$ -time, then what's this nonsense about linear-time sorting algorithms?
 - We can achieve $O(n)$ worst-case runtime if we make assumptions about the input.
*(Input에 대한 가정이)
예전 가능하다.*
 - e.g. They are integers ranging from 0 to k-1.
- Beyond comparison-based sorting algorithms
 - **Counting sort, Bucket sort, Radix sort**

Counting Sort

```
def counting_sort(A, k):
    # A consists of n integers ranging from 0 to k-1
```

Counting Sort

```
def counting_sort(A, k):
    # A consists of n integers ranging from 0 to k-1
    counts = [0] * k
    for i in range(len(A)):
        counts[A[i]] += 1
```

Counting Sort

```
def counting_sort(A, k):
    # A consists of n integers ranging from 0 to k-1
    counts = [0] * k
    for i in range(len(A)):
        counts[A[i]] += 1
    result = []
    for i in range(k):
        # Extends result by counts[i] i's
        result.extend([i] * counts[i])
```

Counting Sort

```
def counting_sort(A, k):
    # A consists of n integers ranging from 0 to k-1
    counts = [0] * k
    for i in range(len(A)):
        counts[A[i]] += 1
    result = []
    for i in range(k):
        # Extends result by counts[i] i's
        result.extend([i] * counts[i])
    return result
```

Counting Sort

```
def counting_sort(A, k):
    # A consists of n integers ranging from 0 to k-1
    counts = [0] * k
    for i in range(len(A)):
        counts[A[i]] += 1
    result = []
    for i in range(k):
        # Extends result by counts[i] i's
        result.extend([i] * counts[i])
    return result
```

in
k

Titled algorithm
수열 정렬 알고리즘

Worst-case runtime $\Theta(n+k)$

Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3

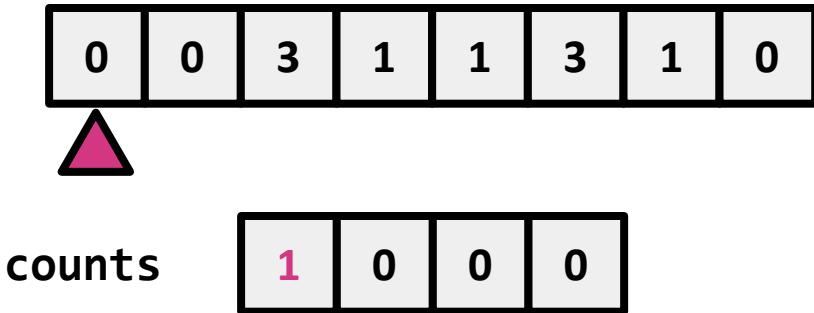
0	0	3	1	1	3	1	0
---	---	---	---	---	---	---	---

counts

0	0	0	0
---	---	---	---

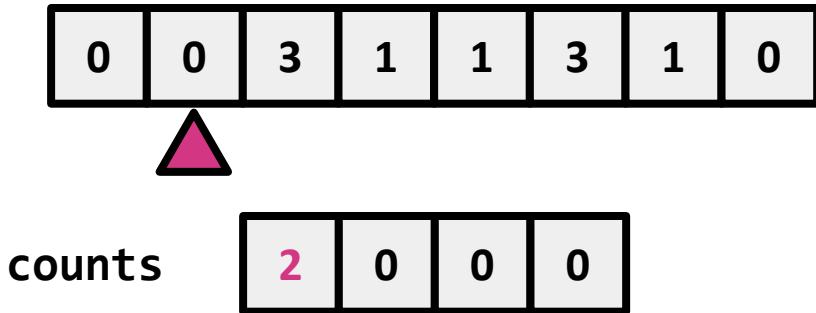
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



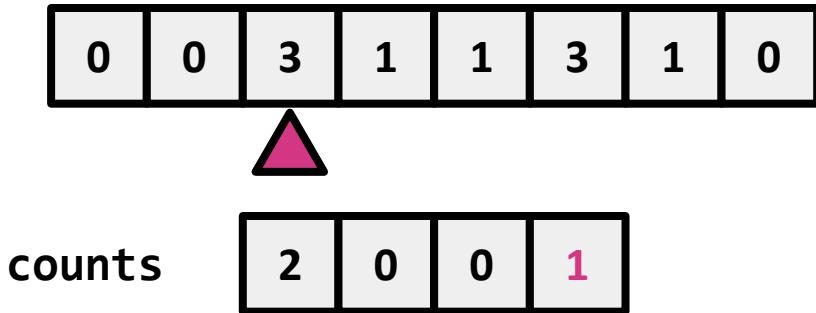
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



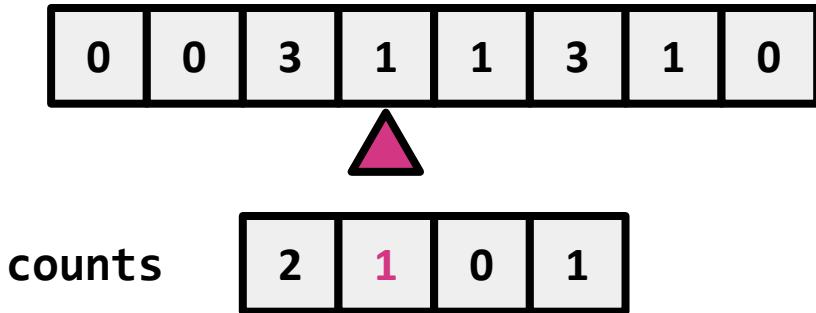
Counting Sort

- Suppose A consists of 8 integers ranging from 0 to 3



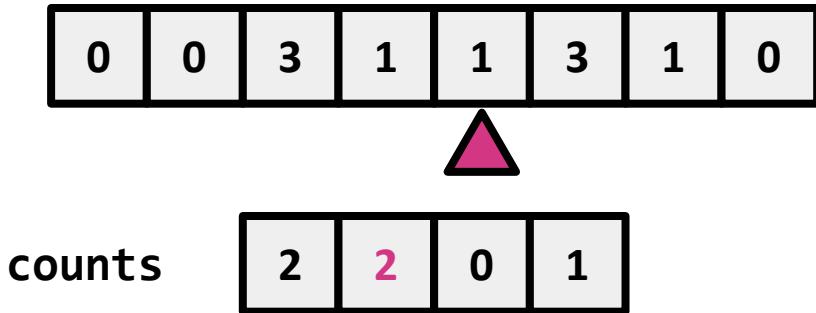
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



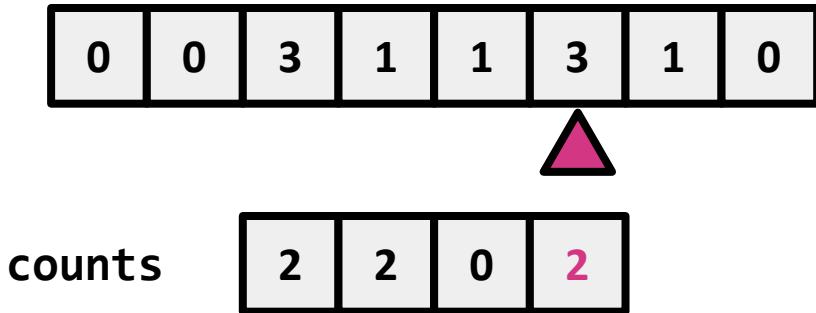
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



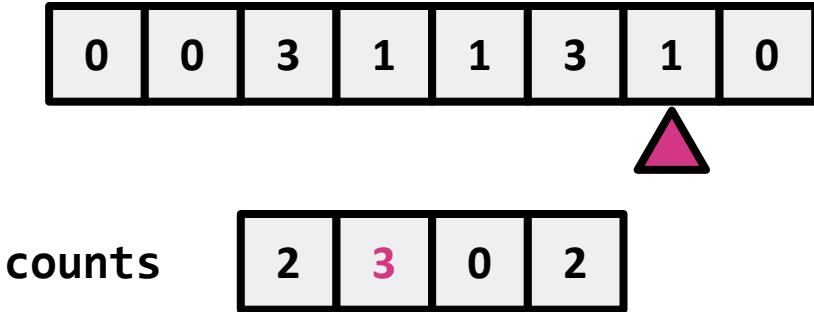
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



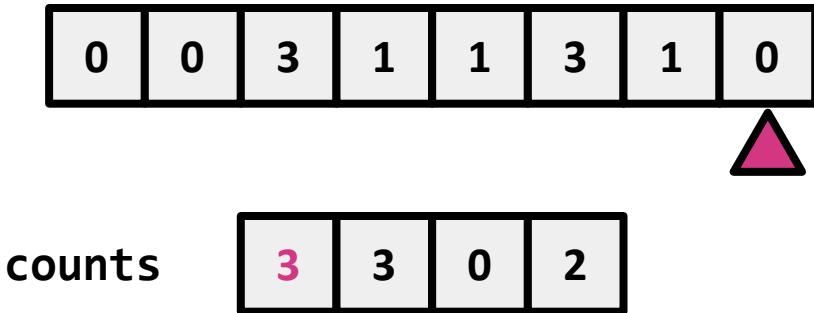
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



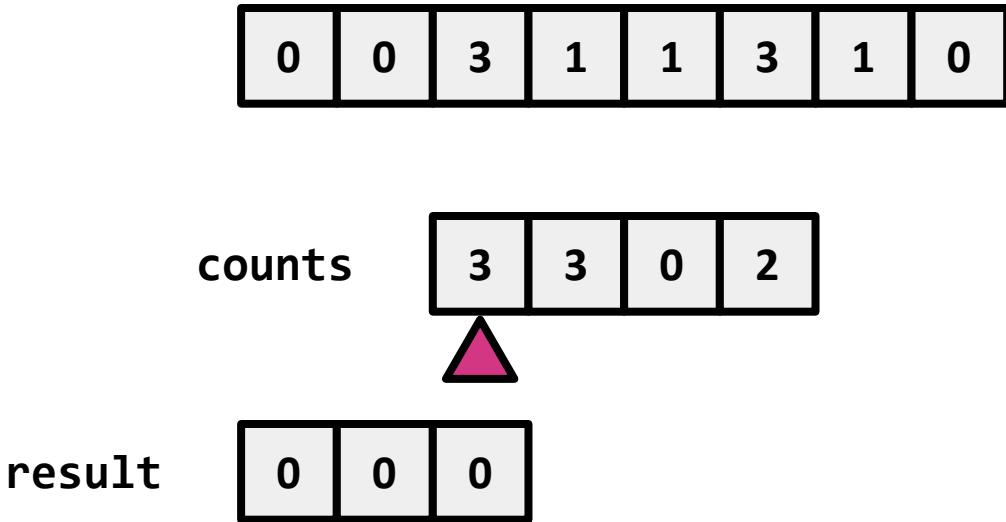
Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



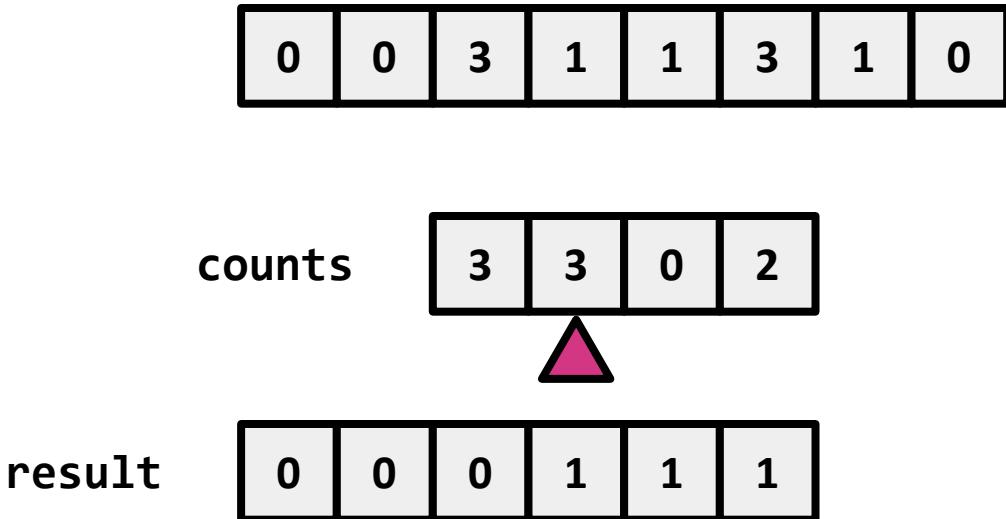
Counting Sort

- Suppose A consists of 8 integers ranging from 0 to 3



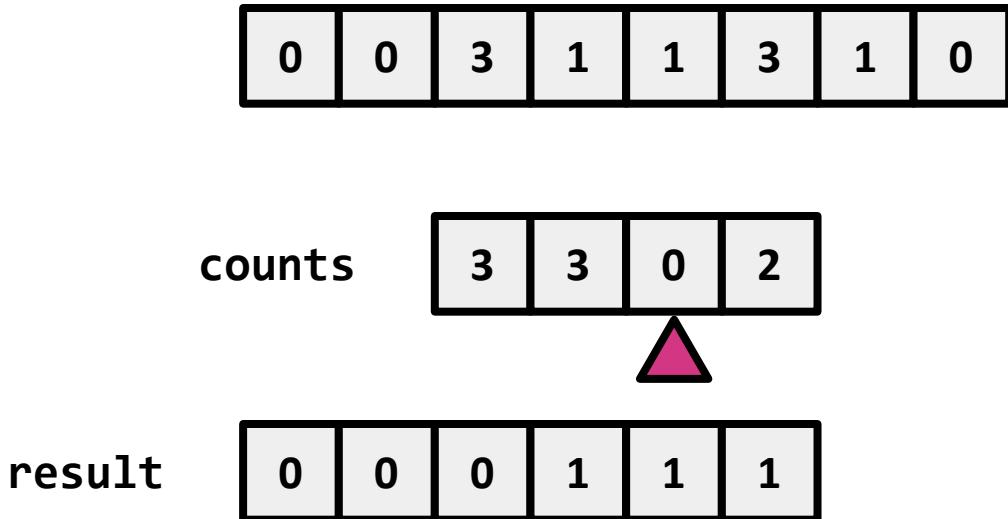
Counting Sort

- Suppose A consists of 8 integers ranging from 0 to 3



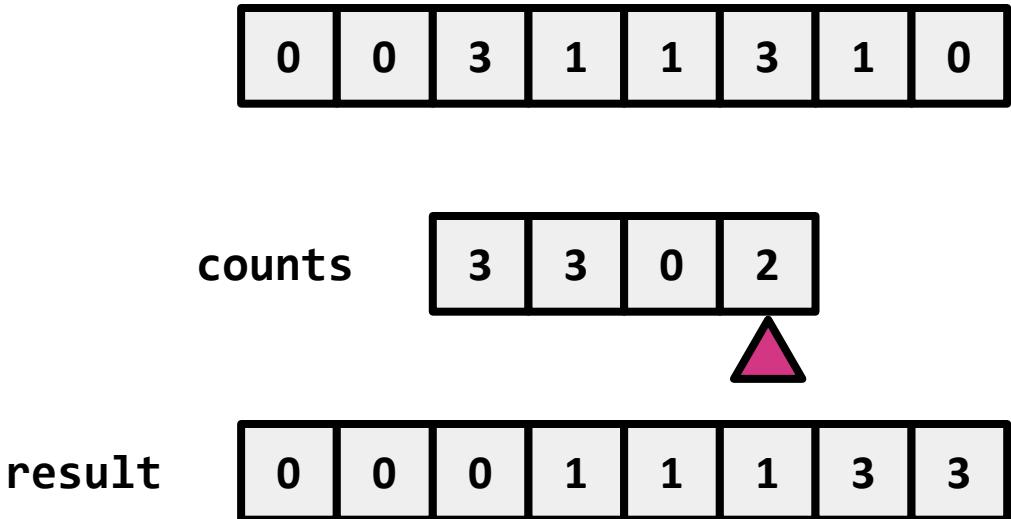
Counting Sort

- Suppose A consists of 8 integers ranging from 0 to 3



Counting Sort

- Suppose **A** consists of 8 integers ranging from 0 to 3



Bucket Sort

```

def bucket_sort(A, k, num_buckets):
    # A consists of n integers ranging from 0 to k-1.
    # Pointless to have more buckets than integers.
    num_buckets = min(num_buckets, k)
    buckets = [[] for i in range(num_buckets)]
    for i in range(len(A)):
        b = get_bucket(A[i], k, num_buckets)
        buckets[b].append(A[i])
    result = []
    if num_buckets < k:
        for j in range(num_buckets):
            result.extend(stable_sort(buckets[j]))
    else:
        for j in range(num_buckets):
            result.extend(buckets[j])
    return result
  
```

n
 k

Worst-case runtime $\Theta(\max\{n \log(n), n+k\})$

Bucket Sort

```

def bucket_sort(A, k, num_buckets):
    # A consists of n integers ranging from 0 to k-1.
    # Pointless to have more buckets than integers.
    num_buckets = min(num_buckets, k)
    buckets = [[] for i in range(num_buckets)]
    for i in range(len(A)):
        b = get_bucket(A[i], k, num_buckets)
        buckets[b].append(A[i])
    result = []
    if num_buckets < k:
        for j in range(num_buckets):
            result.extend(stable_sort(buckets[j]))
    else:
        for j in range(num_buckets):
            result.extend(buckets[j])
    return result
  
```

A stable sort keeps equal elements in the same order:
 $[1^{(a)}, 3, 2, 1^{(b)}] \Rightarrow [1^{(a)}, 1^{(b)}, 2, 3]$

Worst-case runtime $\Theta(\max\{n \log(n), n+k\})$

Happens if $k > \text{num_buckets}$ and all of the values end up in the same bucket.

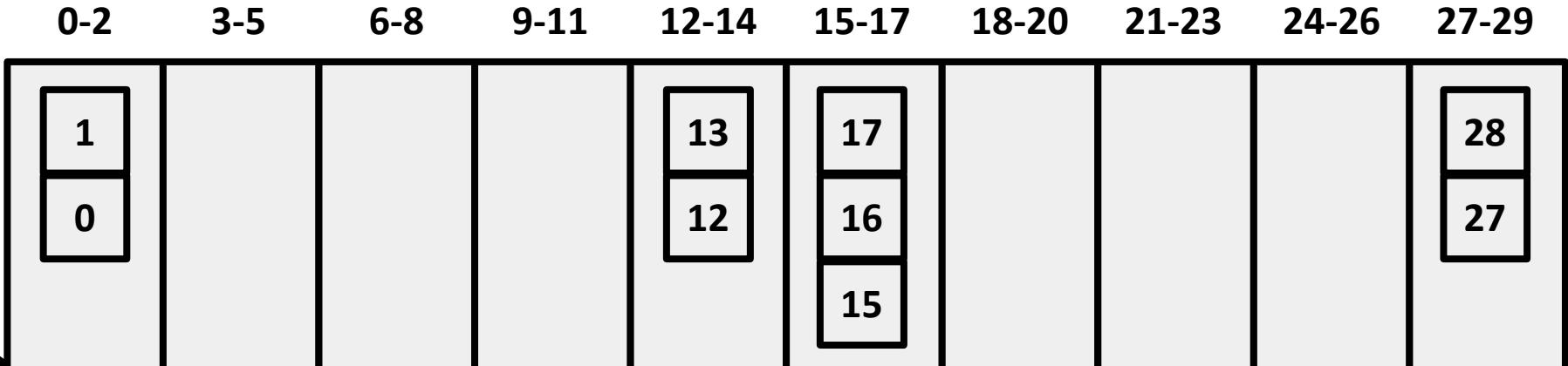
Bucket Sort

```
def get_bucket(value, k, num_buckets):
    # The implementation of this function varies
    # depending on the predefined bucketing scheme;
    # the following examples rely on use of int
    # division.
    return value / math.ceil(k / num_buckets)
```

Worst-case runtime $\Theta(1)$

Bucket Sort

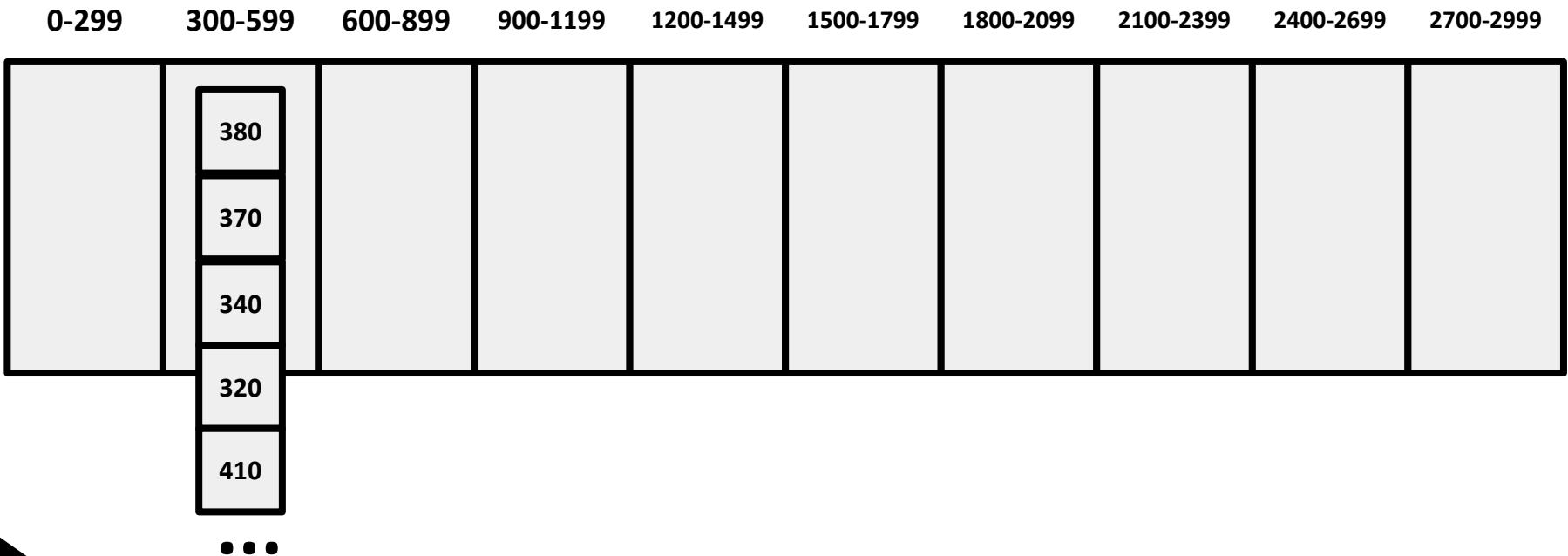
- Two cases for num_buckets and k:
 - $k \leq \text{num_buckets}$** At most one key per bucket, so buckets don't need another stable_sort to be sorted (similar to counting_sort).
 - $k > \text{num_buckets}$** Might be multiple keys per bucket, so buckets need another stable_sort to be sorted.
- Suppose $k = 30$ and $\text{num_buckets} = 10$. Then we group keys 0 to 2 in the same bucket, 3 to 5 in the same bucket, etc.
 - $A = [17, 13, 16, 12, 15, 1, 28, 0, 27]$ produces:



Bucket Sort

element \rightarrow ~~good~~
 \rightarrow bad

- In an extreme case, a bucket might receive all of the inserted keys.
- Suppose $k = 3000$ and $\text{num_buckets} = 10$.
 - $A = [380, 370, 340, 320, 410, \dots]$ would need to `stable_sort` all of the elements in the original list since they all fall in the same bucket.



Radix Sort

Digit by Digit

```

def radix_sort(A, d, k):
    # A consists of n d-digit integers
    # with digits ranging from 0 to k-1.
    for i in range(d):
        # Creates list of (value's digit i, value) pairs
        # For i = 0: [23, 4, 51, 76, 8] =>
        # [(3, 23), (4, 4), (1, 51), (6, 76), (8, 8)]
        A_pairs = make_pairs(A, k, i)
        # Bucket sorts according to first element of
        # pair and returns a list of values
        A_new = bucket_sort_with_pairs(A_pairs, k, k)
        A = A_new
    return result
  
```

stable

Radix Sort

```

def radix_sort(A, d, k):
    # A consists of n d-digit integers
    # with digits ranging from 0 to k-1.
    for i in range(d):
        # Creates list of (value's digit i, value) pairs
        # For i = 0: [23, 4, 51, 76, 8] =>
        # [(3, 23), (4, 4), (1, 51), (6, 76), (8, 8)]
        A_pairs = make_pairs(A, k, i) → stable sort이므로
                                         가능.
        # Bucket sorts according to first element of
        # pair and returns a list of values
        A_new = bucket_sort_with_pairs(A_pairs, k, k)
        A = A_new
    return result
  
```

digit의 ↗

Worst-case runtime $\Theta(d(n+k))$

Radix Sort

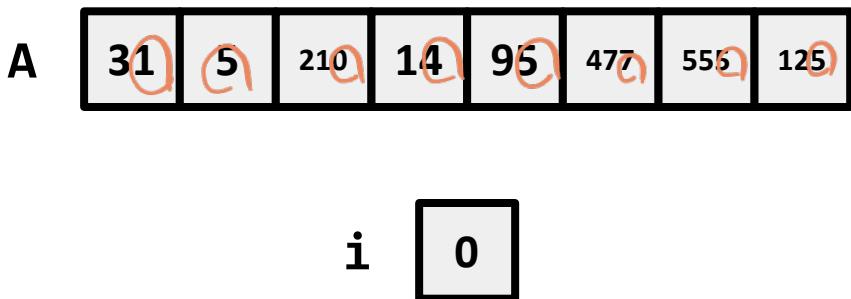
```
def make_pairs(A, k, i):
    result = []
    for a in A:
        # e.g. a=1023, k=10, i=1: (1023/(10**1))%10 = 2
        key = (a / (k ** i)) % k
        result.append((key, a))
    return result
```

10^1 polynomial

Worst-case runtime $\Theta(n)$

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **radix_sort(A, 3, 10)**:



Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **radix_sort(A, 3, 10)**:

A	31	5	210	14	95	477	555	125
----------	----	---	-----	----	----	-----	-----	-----

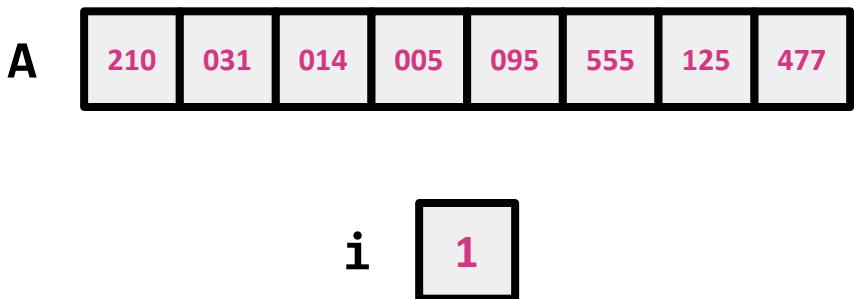
i 0

A_pairs	(1, 031)	(5, 005)	(0, 210)	(4, 014)	...	(5, 125)
----------------	----------	----------	----------	----------	-----	----------

A_new	210	031	014	005	095	555	125	477
--------------	-----	-----	-----	-----	-----	-----	-----	-----

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **radix_sort(A, 3, 10)**:



Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **radix_sort(A, 3, 10)**:

A	210	031	014	005	095	555	125	477
----------	-----	-----	-----	-----	-----	-----	-----	-----

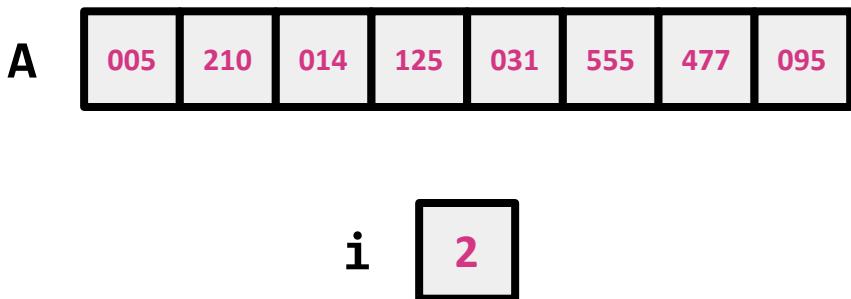
i 1

A_pairs	(1, 210)	(3, 031)	(1, 014)	(0, 005)	...	(7, 477)
----------------	----------	----------	----------	----------	-----	----------

A_new	005	210	014	125	031	555	477	095
--------------	-----	-----	-----	-----	-----	-----	-----	-----

Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **radix_sort(A, 3, 10)**:



Radix Sort

- Suppose **A** consists of eight 3-digit integers, with digits ranging from 0 to 9. Calling **radix_sort(A, 3, 10)**:

A	005	210	014	125	031	555	477	095
----------	-----	-----	-----	-----	-----	-----	-----	-----

i 2

A_pairs	(0, 005)	(2, 210)	(0, 014)	(1, 125)	...	(0, 095)
----------------	----------	----------	----------	----------	-----	----------

A_new	005	014	031	095	125	210	477	555
--------------	-----	-----	-----	-----	-----	-----	-----	-----

→ should be stable sort

Radix Sort



Proof of correctness

- **Inductive hypothesis:**

After the i 'th iteration, \mathbf{A} is sorted by the first i least-significant digits.

- **Base case: ($i = 0$)**

At the start of the first iteration, \mathbf{A} is trivially sorted by its 0 least-significant digits.

- **Inductive step:**

Suppose that after the i 'th iteration, \mathbf{A} is sorted by the i least-significant digits.

After we run **bucket_sort** (sorts \mathbf{A} by the $i+1$ digit of the elements), the elements are sorted by their $i+1$ least-significant digits.

Since **bucket_sort** is **stable**, the elements in each bucket keep their original order, and by the inductive hypothesis, they are ordered by their i least-significant digits.

- **Conclusion:**

The loop terminates at the start of iteration d . The collection of d -digit integers in \mathbf{A} are sorted by their d least-significant digits, which implies that \mathbf{A} is sorted.

The story so far

- If we use a **comparison-based** sorting algorithm, it **MUST** run in time $\Omega(n \log(n))$
- If we assume a bit of structure on the values (small integers or other reasonable data), we have an **$O(n)$** -time sorting algorithm

Why would we ever use a comparison-based sorting algorithm??

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
 - It has lots of precision...

π	$\frac{1234}{9876}$	e	$37!$	6.0221409	n^n	42
-------	---------------------	---	-------	-----------	-------	----

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
 - It has lots of precision...

π	$\frac{1234}{9876}$	e	$37!$	6.0221409	n^n	42
-------	---------------------	---	-------	-----------	-------	----



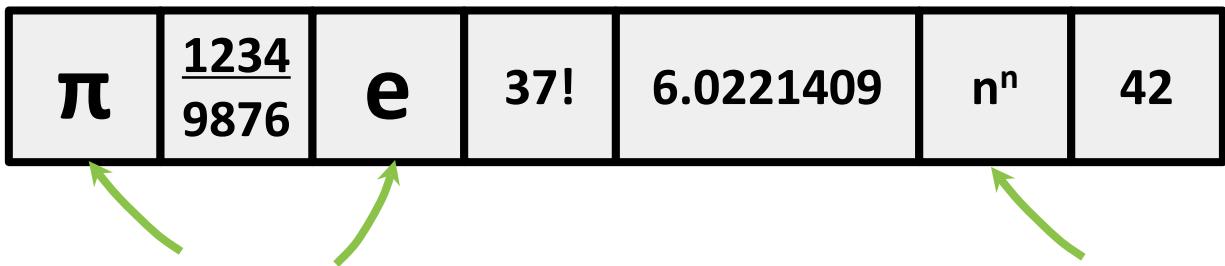
We can compare these pretty quickly (just look at their most significant digit):

- $\pi = 3.14159\dots$
- $e = 2.71818\dots$

But `radix_sort` requires us to look at all digits, which is problematic—both have infinitely many!

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
 - It has lots of precision...



We can compare these pretty quickly (just look at their most significant digit):

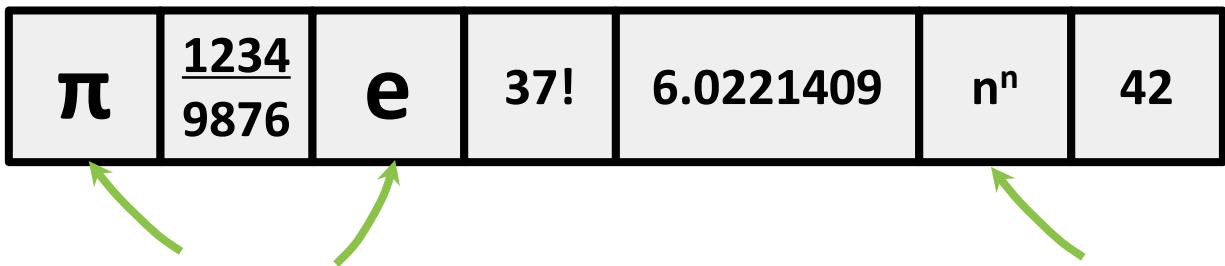
- $\pi = 3.14159\dots$
- $e = 2.71818\dots$

But `radix_sort` requires us to look at all digits, which is problematic—both have infinitely many!

Even with integers, if it's really big, `radix_sort` is slow.

Comparison-Based vs. Linear-Time Sorting

- Why would we ever use a comparison-based sorting algorithm?
 - It has lots of precision...



We can compare these pretty quickly (just look at their most significant digit):

- $\pi = 3.14159\dots$
- $e = 2.71818\dots$

But **radix_sort** requires us to look at all digits, which is problematic—both have infinitely many!

Even with integers, if it's really big, **radix_sort** is slow.

- **radix_sort** needs extra memory for the buckets (not in-place).
- Need to know ordering and buckets ahead of time for linear-time sorting.

Today's Outline

- Linear-Time Sorting
 - ~~Comparison-based sorting lower bounds~~ Done!
 - ~~Algorithms: Counting sort, bucket sort, and radix sort~~ Done!
 - Reading: CLRS 8.1-8.2

CSE301 Introduction to Algorithms

Randomized Algorithms I

Fall 2022



Instructor : Hoon Sung Chwa

Course Overview

- Algorithmic Analysis
- Divide and Conquer
- **Randomized Algorithms**
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Randomized Algorithms I
 - How do we measure the runtime of a randomized algorithm?
 - **Expected runtime & worst-case runtime**
 - Algorithms
 - Bogosort
 - Randomized quicksort
 - Randomized select
 - Majority element
 - Reading: CLRS 5, 7, 9.2

Randomized Algorithms

- A randomized algorithm
 - 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'll focus on these algorithms today.
 - Monte Carlo algorithms guarantee runtime, but not correctness. We'll revisit this when we see Karger's algorithm.

Pivot 선택 \rightarrow 랜덤

정확한 결과 \uparrow
확률

결과는 같

정확도 \uparrow
running time \downarrow

running time \uparrow
정확도 \downarrow

How do we measure the runtime of a randomized algorithm?

- Scenario 1

1. Bad guy picks the input.

2. You run your randomized algorithm.



- Scenario 2

1. Bad guy picks the input.

2. Bad guy chooses the randomness (fixes the dice)



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

Bogosort

```
def bogosort(A):
    # Randomly permutes A until it's sorted
    while True:
        random.shuffle(A)
        sorted = True
        for i in range(len(A)-1):
            if A[i] > A[i+1]:
                sorted = False
        if sorted:
            return A
```

Worst-case runtime
 $O(\infty)$

Bogosort

- Unlike the deterministic algorithms that we've studied so far, when analyzing Las Vegas randomized algorithms, we're interested in:
 - What's the expected (average-case) runtime of the algorithm?
 - How does this compare to the worst-case runtime of the algorithm?

Bogosort

```
def bogosort(A):
    # Randomly permutes A until it's sorted
    while True:
        random.shuffle(A)
        sorted = True
        for i in range(len(A)-1):
            if A[i] > A[i+1]:
                sorted = False
        if sorted:
            return A
```

Worst-case

$O(\infty)$



Think of this as the adversary chooses the randomness.

Expected

$O(n \cdot n!)$



$\Pr[\text{randomly list sorted}] = 1/n!$

We expect to permute ~~A~~ $n!$ times before it's sorted.
 Each permutation requires $O(n)$ -time.

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.



Quicksort

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

Choose a pivot.



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

Partition around it.



Quicksort



Choose a pivot.



Partition around it.



Recurse on both subarrays.

Quicksort



Choose a pivot.



Partition around it.



Recurse on both subarrays.

Choose a pivot and partition around it.



Choose a pivot and partition around it.



Quicksort



Choose a pivot.



Partition around it.



Recurse on both subarrays.

Choose a pivot and partition around it.



Choose a pivot and partition around it.

Recurse on both subarrays.



Recurse on both subarrays.

Quicksort

```
def quicksort(A):
    if len(A) <= 1:
        return
    pivot = A[0] → pivot fixed (시작점)
    left, right = partition_about_pivot(A, pivot)
    quicksort(left)
    quicksort(right)
```

recurrive

Worst-case runtime

$$\Theta(n^2)$$

Randomized Quicksort

```
def randomized_quicksort(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    quicksort(left)
    quicksort(right)
```

이진 quick-sort 를
다시보자.

Worst-case

$\Theta(n^2)$



Think of this as the adversary chooses the randomness.

Expected

$\Theta(n \log(n))$

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? 🤔

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

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

Runtime of partition.

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

Master method $a = 1$, $b = 2$, $d = 1$.

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? 🤔

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? 🤔

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

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

$$= O(n^2)$$

Iteration method

Expected Runtime of Randomized Quicksort

How do we know the expected runtime of quicksort is $O(n \log n)$?

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 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.

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

Expected Runtime of Randomized 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 Randomized Quicksort

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

Expected Runtime of Randomized 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:

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

Expected Runtime of Randomized 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:

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

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 Randomized 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:

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

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 Randomized 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:

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

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?

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

↳ expectation

By linearity of expectation

We need to figure out this value!

Expected Runtime of Randomized 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

기대값

Expected Runtime of Randomized 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. \rightarrow n. 8. 9가 pivot이 될 때.

Expected Runtime of Randomized 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 ...



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



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



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

Expected Runtime of Randomized Quicksort

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

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

Expected Runtime of Randomized Quicksort

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

$$\begin{aligned}
 \sum_{\substack{n-2 \\ a=0}}^{\sum} E[X_{a,b}] &= \sum_{\substack{n-2 \\ a=0}}^{\sum} 2 / (b - a + 1) \\
 &= \sum_{\substack{n-a-1 \\ a=0}}^{\sum} 2 / (c + 1)
 \end{aligned}$$

Expected Runtime of Randomized Quicksort

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

$$\begin{aligned}
 \sum_{\substack{n-2 \\ a=0}}^{\sum} E[X_{a,b}] &= \sum_{\substack{n-2 \\ a=0}}^{\sum} 2 / (b - a + 1) \\
 &= \sum_{\substack{n-2 \\ a=0}}^{\sum} 2 / (c + 1) \\
 &\leq \sum_{\substack{n-2 \\ a=0}}^{\sum} 2 / (c + 1)
 \end{aligned}$$

Expected Runtime of Randomized Quicksort

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

$$\begin{aligned}
 \sum_{\substack{n-2 \\ a=0}}^{\substack{n-1 \\ b=a+1}} E[X_{a,b}] &= \sum_{\substack{n-2 \\ a=0}}^{\substack{n-1 \\ b=a+1}} 2 / (b - a + 1) \\
 &= \sum_{\substack{n-2 \\ a=0}}^{\substack{n-1 \\ c=1}} 2 / (c + 1) \\
 &\leq \sum_{\substack{n-1 \\ a=0}}^{\substack{n-1 \\ c=1}} 2 / (c + 1) \\
 &\leq 2n \sum_{c=1}^n 1 / (c+1)
 \end{aligned}$$

Expected Runtime of Randomized Quicksort

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

$$\begin{aligned}
 \sum_{\substack{n-2 \\ a=0}}^{\sum} E[X_{a,b}] &= \sum_{\substack{n-2 \\ a=0}}^{\sum} 2 / (b - a + 1) \\
 &= \sum_{\substack{n-2 \\ a=0}}^{\sum} 2 / (c + 1) \\
 &\leq \sum_{\substack{n-1 \\ a=0}}^{\sum} 2 / (c + 1) \quad \ln(n) + O(n) \\
 &\leq 2n \sum_{c=1}^{n-1} 1 / (c+1) \leq 2n \sum_{c=1}^{n-1} 1/c
 \end{aligned}$$

Harmonic series

Expected Runtime of Randomized Quicksort

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

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

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

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

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

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

Harmonic series



Randomized Quicksort

```
def randomized_quicksort(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    quicksort(left)
    quicksort(right)
```

Worst-case $\Theta(n^2)$

Think of this as the adversary chooses the randomness.



Expected $\Theta(n \log(n))$

We can lower-bound it with the sorting lower bound!



A note on implementation

- This pseudocode is easy to understand and analyze, but is not a good way to implement this algorithm.

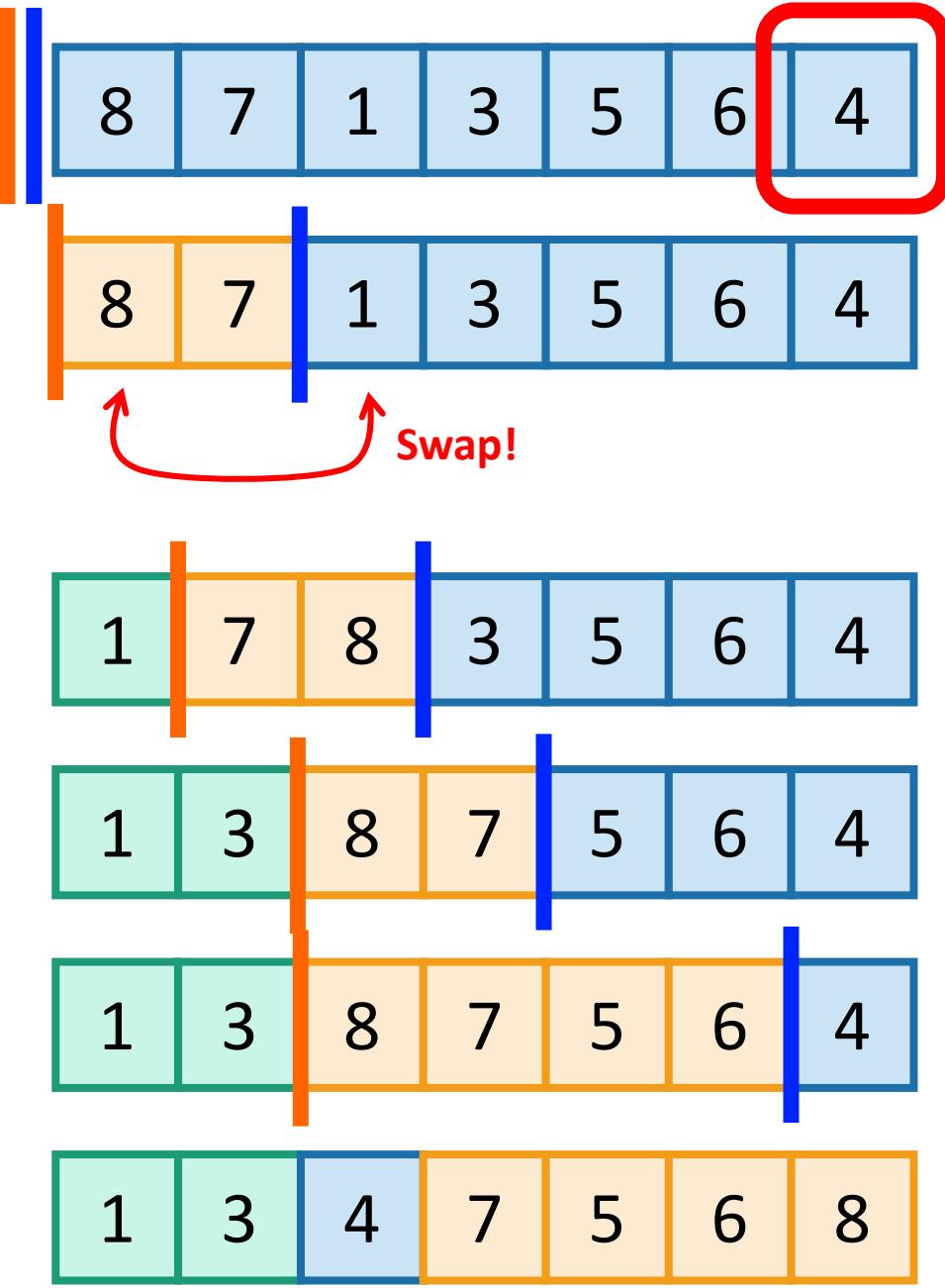
```
• QuickSort(A):

- If len(A) <= 1:
  - return
- Pick some x = A[i] at random. Call this the pivot.
- PARTITION the rest of A into:
  - L (less than x) and
  - R (greater than x)
- Replace A with [L, x, R] (that is, rearrange A in this order)
- QuickSort(L)
- QuickSort(R)

```

- Instead, implement it **in-place** (without separate L and R)
 - See CLRS 7.1
 - Here are some Hungarian Folk Dancers showing you how it's done:
<https://www.youtube.com/watch?v=ywWBy6J5gz8>

A better way to do Partition



Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize | and |

Step | forward.

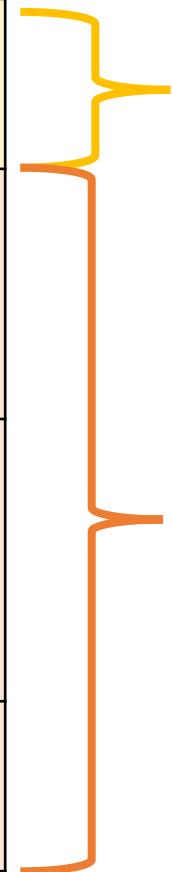
When | sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

See CLRS

QuickSort vs MergeSort

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	<ul style="list-style-type: none"> Worst-case: $O(n^2)$ Expected: $O(n \log(n))$ 	Worst-case: $O(n \log(n))$
Used by	<ul style="list-style-type: none"> Java for primitive types C qsort Unix g++ 	<ul style="list-style-type: none"> Java for objects Perl
In-Place? (With $O(\log(n))$ extra memory)	Yes, pretty easily	Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice runtime).
Stable?	No	Yes



Understand this

These are just for fun.
(Not on exam).

*In fact, I don't know how to do this if you want $O(n\log(n))$ worst-case runtime and stability.

Better Quicksort?

Any ideas to make randomized_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 randomized_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 and instead partition around the median of medians.

Randomized Selection

Randomized Selection

Our next example of a randomized algorithm is randomized_select.

You've actually seen it before.

Randomized Selection

```
def select randomized_select(A, k, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = random.choice(A)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, c)
```

Worst-case runtime

$\Theta(n^2)$

Expected runtime

$\Theta(n)$

Expected Runtime of Randomized Selection

How do we know the expected runtime of
randomized_select is $O(n)$?

Please see CLRS 9.2

Majority Element

Majority Element

반복 이상

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? 🤔



`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).

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). 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

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

Recurrence: $T(n) = 2T(n/2) + O(n)$
Runtime
 $O(n \log n)$

Proof of Correctness

Inductive Hypothesis:

majority_element correctly finds the majority element, provided one exists, for inputs of length 2^i .

Proof of Correctness

Inductive Hypothesis:

majority_element correctly finds the majority element, provided one exists, for inputs of length 2^i .

Base case:

majority_element correctly finds the majority element when $i = 0$ for inputs of length $2^0 = 1$ since **majority_element** returns **A[0]**.

Proof of Correctness

Inductive Hypothesis:

majority_element correctly finds the majority element, provided one exists, for inputs of length 2^i .

Base case:

majority_element correctly finds the majority element when $i = 0$ for inputs of length $2^0 = 1$ since **majority_element** returns $A[0]$.

Inductive step:

Suppose the algorithm works on input lists of length $n/2 = 2^{i-1}$.

Now consider an input of length $n = 2^i$.

Proof of Correctness

Inductive Hypothesis:

majority_element correctly finds the majority element, provided one exists, for inputs of length 2^i .

Base case:

majority_element correctly finds the majority element when $i = 0$ for inputs of length $2^0 = 1$ since **majority_element** returns **A[0]**.

Inductive step:

Suppose the algorithm works on input lists 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[:mid]** or **A[mid+1:]**;
otherwise it would occur at most $\lfloor n/2 \rfloor$ times.

Proof of Correctness

Inductive Hypothesis:

majority_element correctly finds the majority element, provided one exists, for inputs of length 2^i .

Base case:

majority_element correctly finds the majority element when $i = 0$ for inputs of length $2^0 = 1$ since **majority_element** returns **A[0]**.

Inductive step:

Suppose the algorithm works on input lists 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[:mid]** or **A[mid+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.

Proof of Correctness

Inductive Hypothesis:

majority_element correctly finds the majority element, provided one exists, for inputs of length 2^i .

Base case:

majority_element correctly finds the majority element when $i = 0$ for inputs of length $2^0 = 1$ since **majority_element** returns **A[0]**.

Inductive step:

Suppose the algorithm works on input lists 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[:mid]** or **A[mid+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.

Conclusion:

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

Majority Element

Random Algorithm으로는 안될까?

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?

한마디로 하시지, 개별수 이상인지 확인

Majority Element

```
def majority_element(A):
    # randomized
    while True:
        guess = random.choice(A)
        count = 0
        for a in A:
            if equals(guess, a): count += 1
        if count > n/2+1: return guess
```

Runtime

Expected: 🤔

Worst-case: 🤔

Majority Element

```
def majority_element(A):
    # randomized
    while True:
        guess = random.choice(A)
        count = 0
        for a in A:
            if equals(guess, a): count += 1
        if count > n/2+1: return guess
```

Runtime

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

과반수 이상인 원소 있을 때.

Expected Runtime of Majority Element

- Provided there exists a majority element, this element must occur at least $\lfloor n/2 \rfloor + 1$ times.
 - Each iteration (while loop) requires n equals checks

$$P(\text{find the majority element}) \geq \frac{1}{2}$$

$$\begin{aligned} E[\# \text{ iterations}] &\leq \sum_{i=0}^{\infty} \frac{1}{2^i} && \text{Geometric series} \\ &= 2 \end{aligned}$$

$$E[\# \text{ check}] = 2n$$

$$X_i \mid \begin{cases} 1 & : E_i \\ 0 & : \text{else} \end{cases}$$

$$X = \sum_{i=0}^{\infty} X_i$$

$$E[X] = \sum_{i=0}^{\infty} E[X_i] = \sum_{i=0}^{\infty} P(E_i)$$

$$P(E_i) < \frac{1}{2^{i-1}}$$

$$E[X] < \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} \leq \sum_{i=0}^{\infty} \frac{1}{2^i} = 2$$

E_i : the event that find the majority element at i th iteration

the previous $i-1$ attempts did not find the majority element

Majority Element

Divide and Conquer Runtime

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

Randomized Runtime

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

Today's Outline

- Randomized Algorithms I
 - How do we measure the runtime of a randomized algorithm?
 - Expected runtime & worst-case runtime
 - Algorithms
 - Bogosort
 - ✓ Worst-case: $O(\infty)$ Expected: $O(n \cdot n!)$
 - Randomized quicksort
 - ✓ Worst-case: $\Theta(n^2)$ Expected: $\Theta(n \log(n))$
 - Randomized select
 - ✓ Worst-case: $\Theta(n^2)$ Expected: $\Theta(n)$
 - Randomized majority-element
 - ✓ Worst-case: $O(\infty)$ Expected: $O(n)$
 - Reading: CLRS 5, 7, 9.2

CSE301 Introduction to Algorithms

Randomized Algorithms II

Fall 2022



Instructor : Hoon Sung Chwa

Course Overview

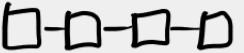
- Algorithmic Analysis
- Divide and Conquer
- **Randomized Algorithms**
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Randomized Algorithms II
 - Direct-address tables, hash tables, hash functions, universal hash families, open-addressing
 - Reading: CLRS: 11

Data Structures

Dictionary operations

		Sorted linked lists	Sorted arrays
Search	 $O(n)$ expected & worst-case	<i>binary search</i> $O(\log n)$ expected & worst-case	
Insert/ Delete	$O(n)$ expected & worst-case <small>without a pointer to the element</small>	$O(n)$ expected & worst-case	

Data Structures

	Sorted linked lists	Sorted arrays	Hash tables
Search	$O(n)$ expected & worst-case	$O(\log n)$ expected & worst-case	$O(1)$ expected $O(n)$ worst-case
Insert/ Delete	$O(n)$ expected & worst-case without a pointer to the element	$O(n)$ expected & worst-case	$O(1)$ expected $O(n)$ worst-case without a pointer to the element

Hashing Basics

Outline

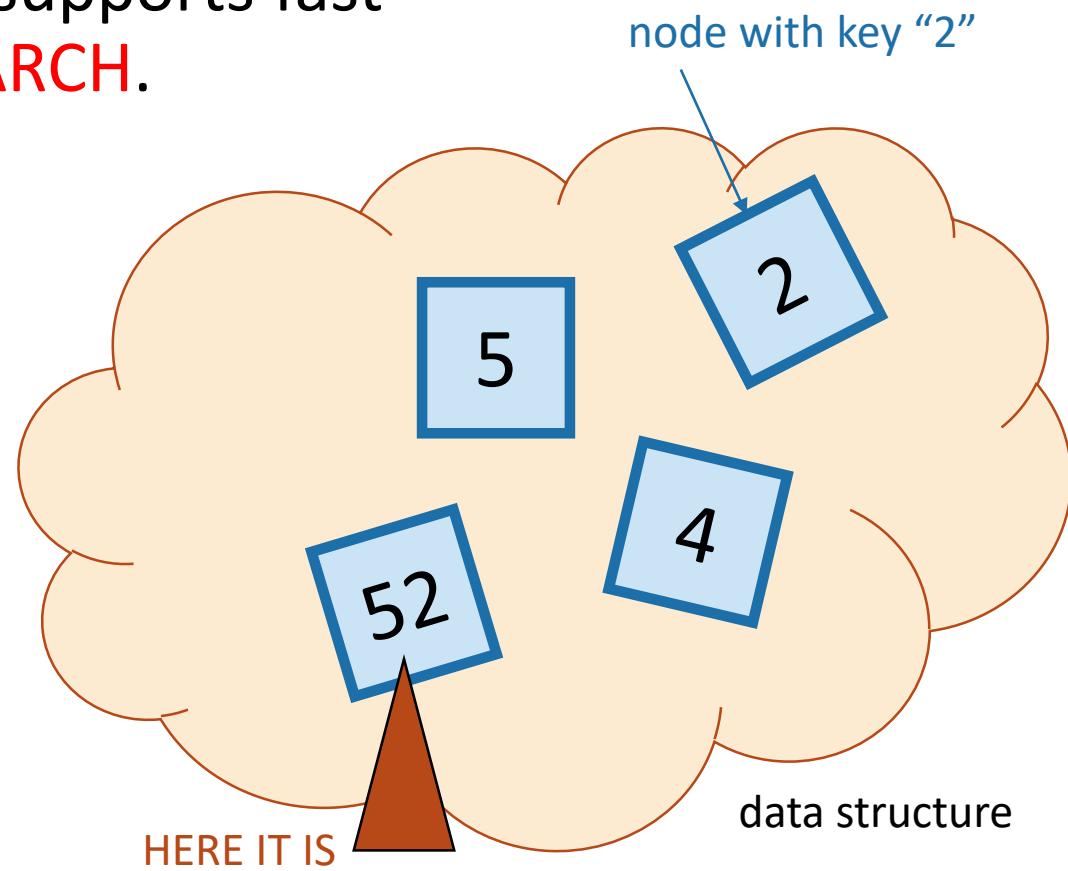


- Hash tables are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
- Hash families are the magic behind hash tables.
- Universal hash families are even more magic.

Goal

- We are interested in putting nodes with keys into a data structure that supports fast **INSERT/DELETE/SEARCH**.

- **INSERT**  5
- **DELETE**  4
- **SEARCH**  52



Today:

- Hash tables:
 - $O(1)$ expected time **INSERT/DELETE/SEARCH**
 - Worse worst-case performance, but often great in practice.

#evensweeterinpractice

eg, Python's `dict`, Java's `HashSet/HashMap`, C++'s `unordered_map`
Hash tables are used for databases, caching, object representation, ...



Applications of Dictionary

- Perhaps, the most popular data structures in CS
 - Database: Dictionary, Spell Checking/Correcting, Web Search
 - Compiler/Interpreter: Variable Name → Physical Address
 - Network Router: IP Address → Wire
 - Network Server: Port Number → Socket/App
 - Virtual Memory: Virtual Address → Physical Address
 - Substring Search: grep in UNIX, etc.
 - String Commonalities: DNA
 - File/directory synchronization: Dropbox
 - Cryptography: File Transfer & Identification, etc.
 - ...

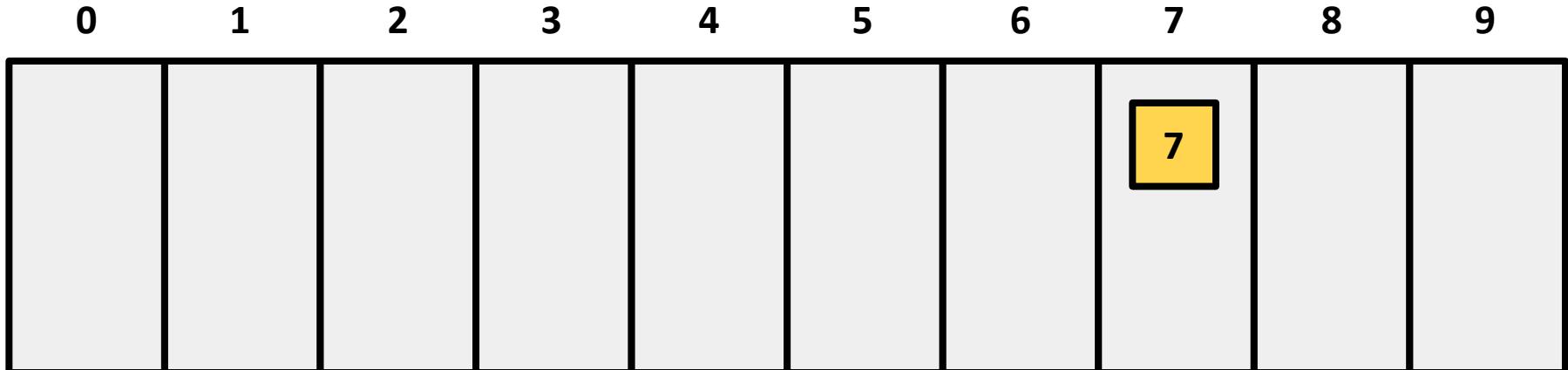
Direct Addressing

How might we get $O(1)$ -time?

Try direct addressing!

One type of item per address.

`insert(7)`



Direct Addressing

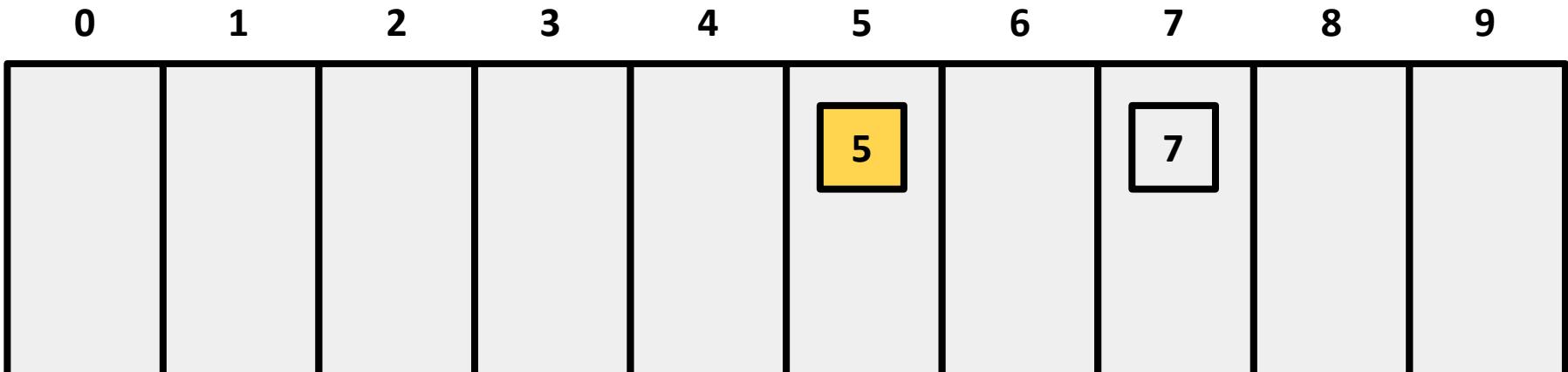
How might we get $O(1)$ -time?

Try direct addressing!

One type of item per address.

`insert(7)`

`insert(5)`



Direct Addressing

How might we get $O(1)$ -time?

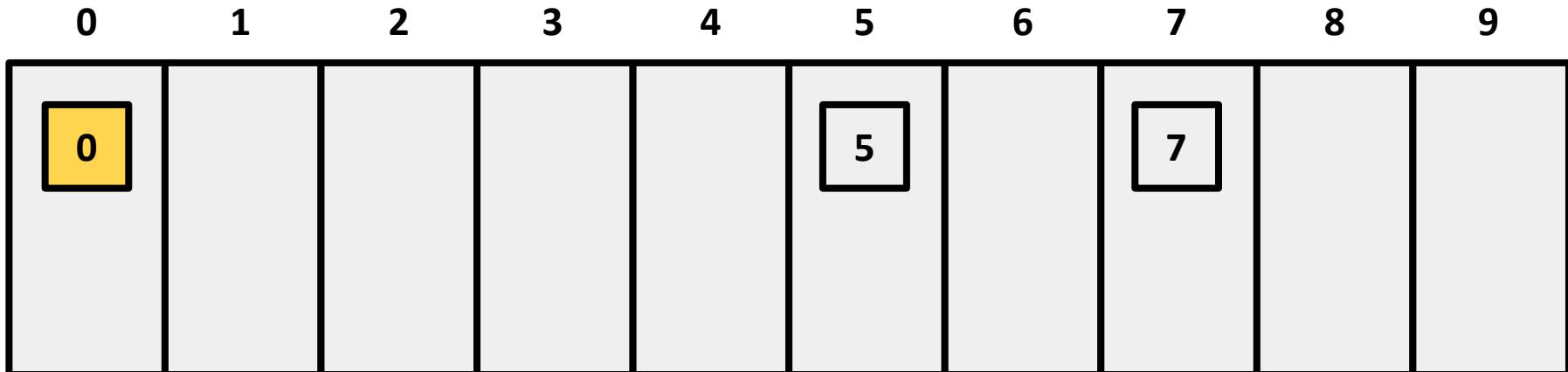
Try direct addressing!

One type of item per address.

`insert(7)`

`insert(5)`

`insert(0)`



Direct Addressing

How might we get $O(1)$ -time?

Try direct addressing!

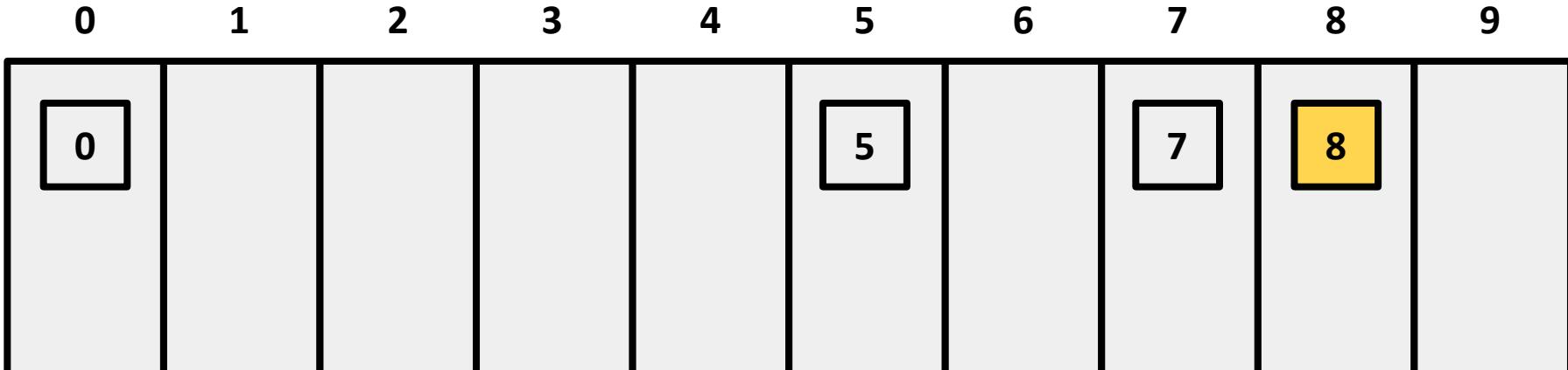
One type of item per address.

`insert(7)`

`insert(5)`

`insert(0)`

`insert(8)`



Direct Addressing

How might we get **O(1)**-time?

Try direct addressing!

One type of item per address.

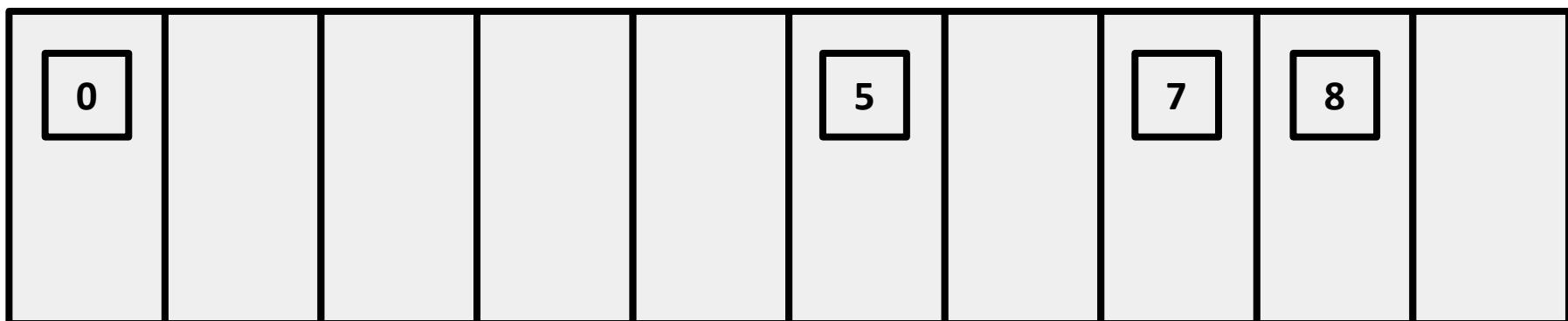
insert(7) search(7)

insert(5) search(2)

insert(0)

insert(8)

0 1 2 3 4 5 6 7 8 9



Direct Addressing

How might we get $O(1)$ -time?

Try direct addressing!



What's the issue with this approach?

Direct Addressing

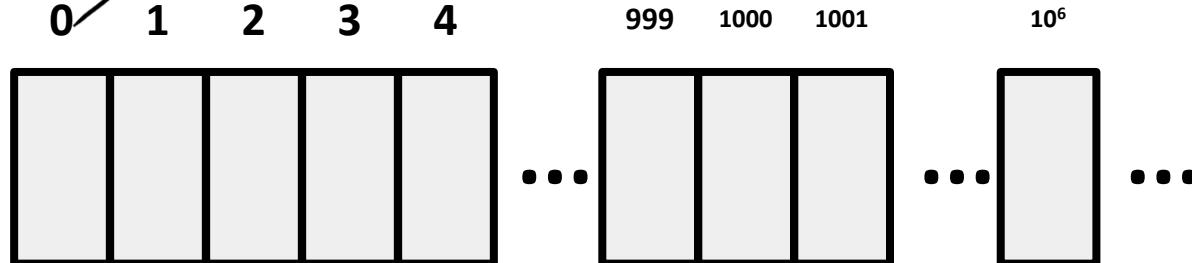
How might we get **O(1)**-time?

Try direct addressing!



What's the issue with this approach?

Similar to `counting_sort` and `bucket_sort` (for $k \leq \text{num_buckets}$), if the set of items being inserted/deleted (e.g. $\{0, 1, 2, \dots, 999, 1000, \dots, 10^6, \dots\}$) is **large**,
B21.011B2169↑
then the **space** required to maintain this data structure becomes an issue.



Direct Addressing

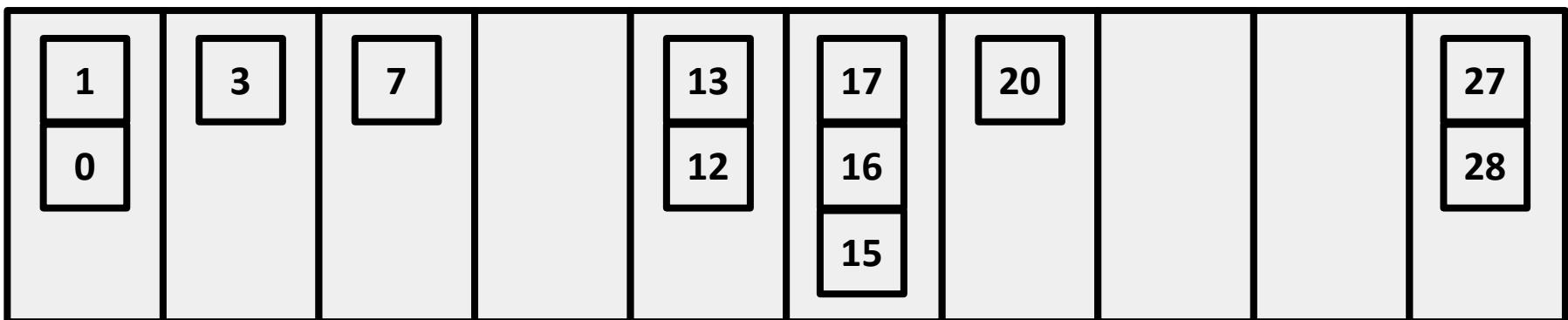
How might we get $O(1)$ -time?

Try direct addressing!

Can we fix this issue by assigning multiple types of item per address, like the case of `bucket_sort`?

Sometimes, this binning approach is useful. `search(12)` still runs pretty fast.

0-2 3-5 6-8 9-11 12-14 15-17 18-20 21-23 24-26 27-29



Direct Addressing

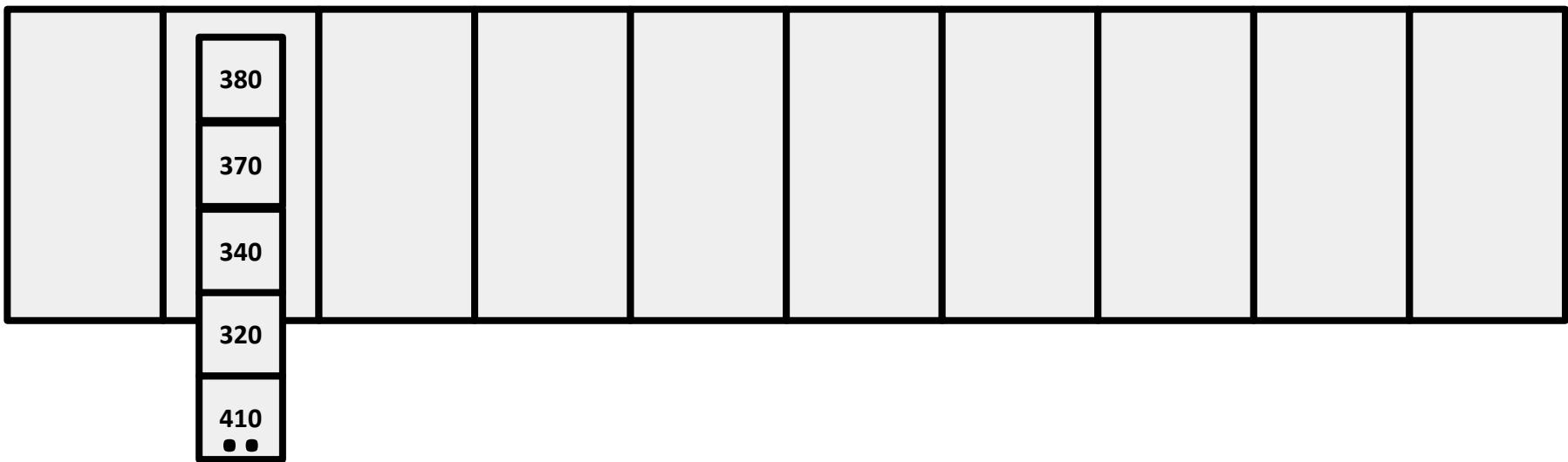
How might we get $O(1)$ -time?

Try direct addressing!

Can we fix this issue by assigning multiple types of item per address, like the case of `bucket_sort`?

Other times, it causes an issue. `search(432)` is slow.

0-299 300-599 600-899 900-1199 1200-1499 1500-1799 1800-2099 2100-2399 2400-2699 2700-2999



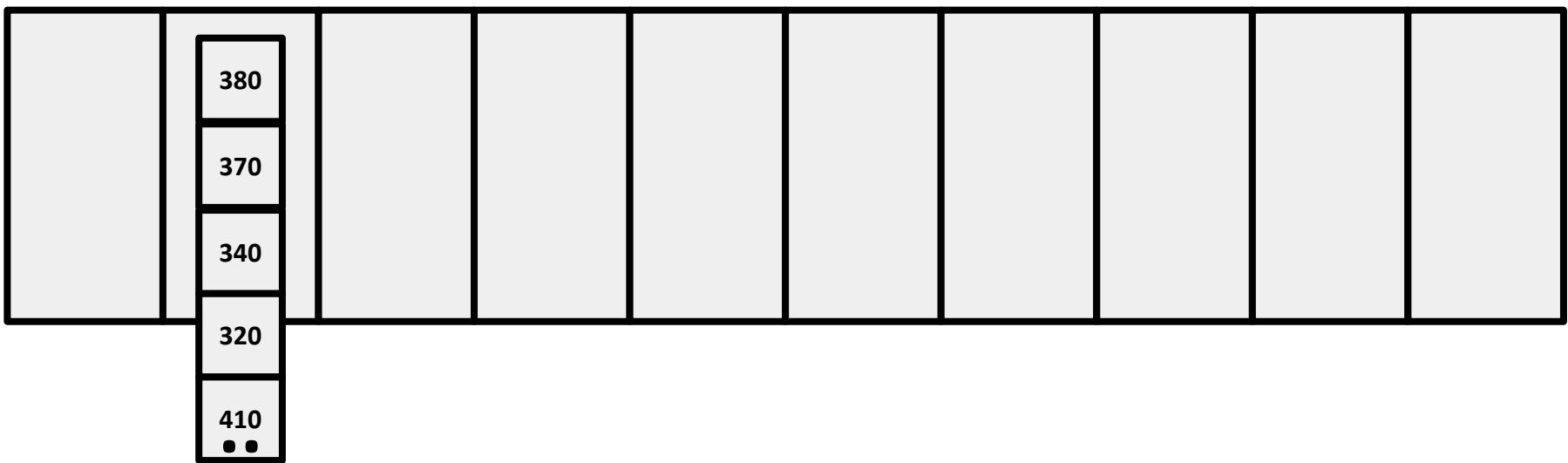
Direct Addressing

This is an example of a hash table.

One with a basic bucketing scheme.

Can we do better?

0-299 300-599 600-899 900-1199 1200-1499 1500-1799 1800-2099 2100-2399 2400-2699 2700-2999



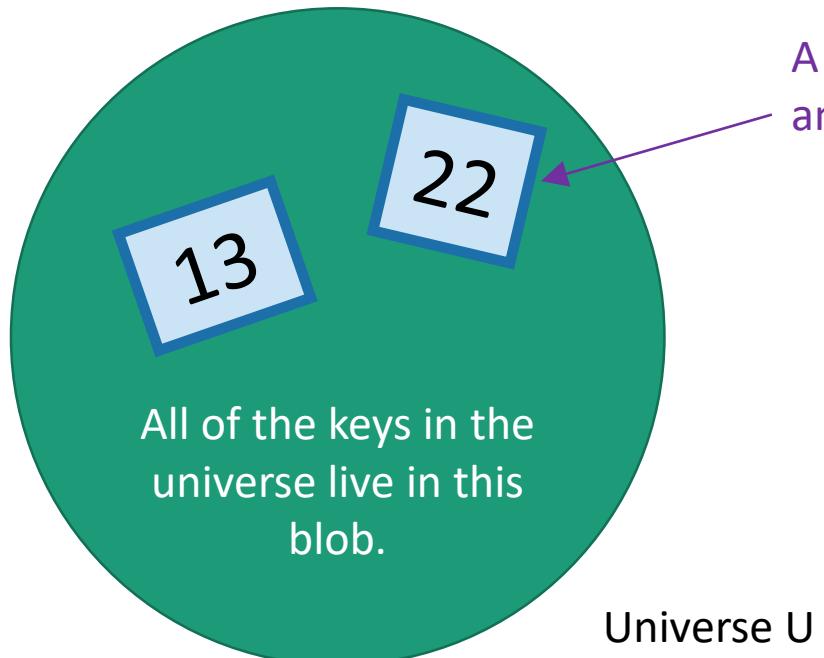
Hash tables

- That was an example of a hash table.
 - not a very good one, though.
- We will be **more clever** (and less deterministic) about our bucketing.
- This will result in fast (expected time)
INSERT/DELETE/SEARCH.

But first! Terminology.

→ 모든 가능한 Key 값들의 전체 set

- We have a **universe U**, of size M.
 - M is really big. → 엄청크다는 가정
- But only a few (say at most n for today's lecture) elements of M are going to show up. → 몇개만 mapping
 - M is waaaayyyyyy bigger than n. *Insert . search*
- But we don't know which ones will show up in advance.



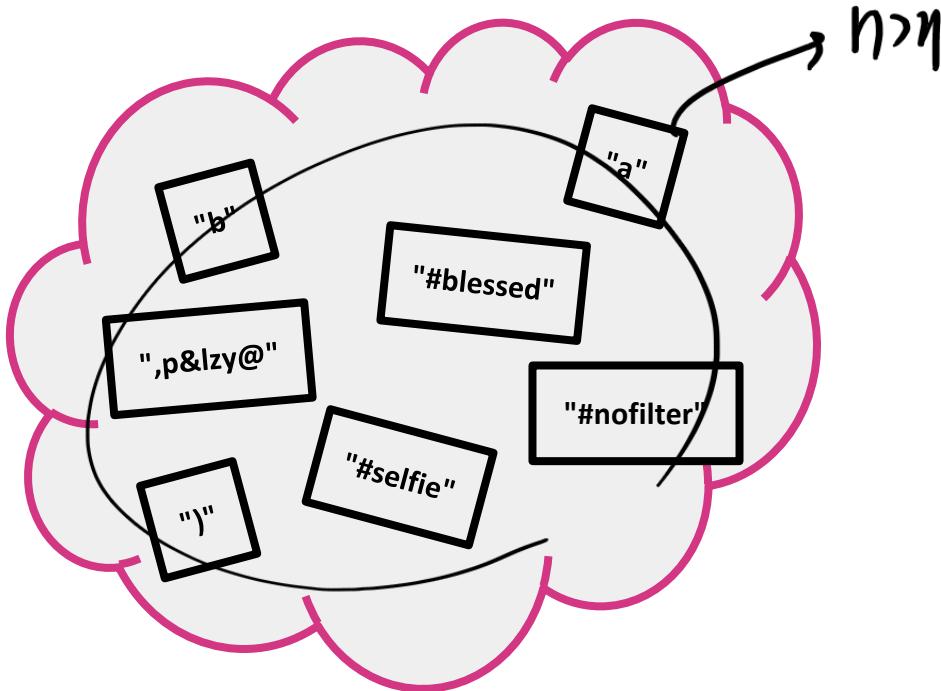
A few elements are special and will actually show up.

Example: U is the set of all strings of at most 140 ascii characters. 128^{140} of them.

The only ones which I care about are those which appear as trending hashtags on twitter. #hashinghashtags

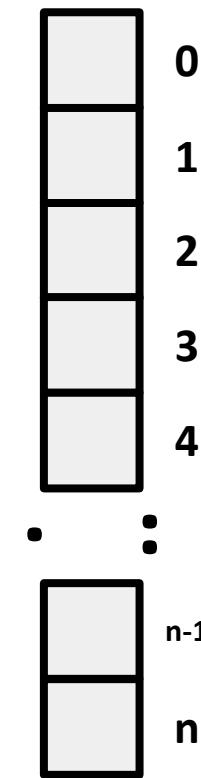
There are way fewer than 128^{140} of these.

An Example

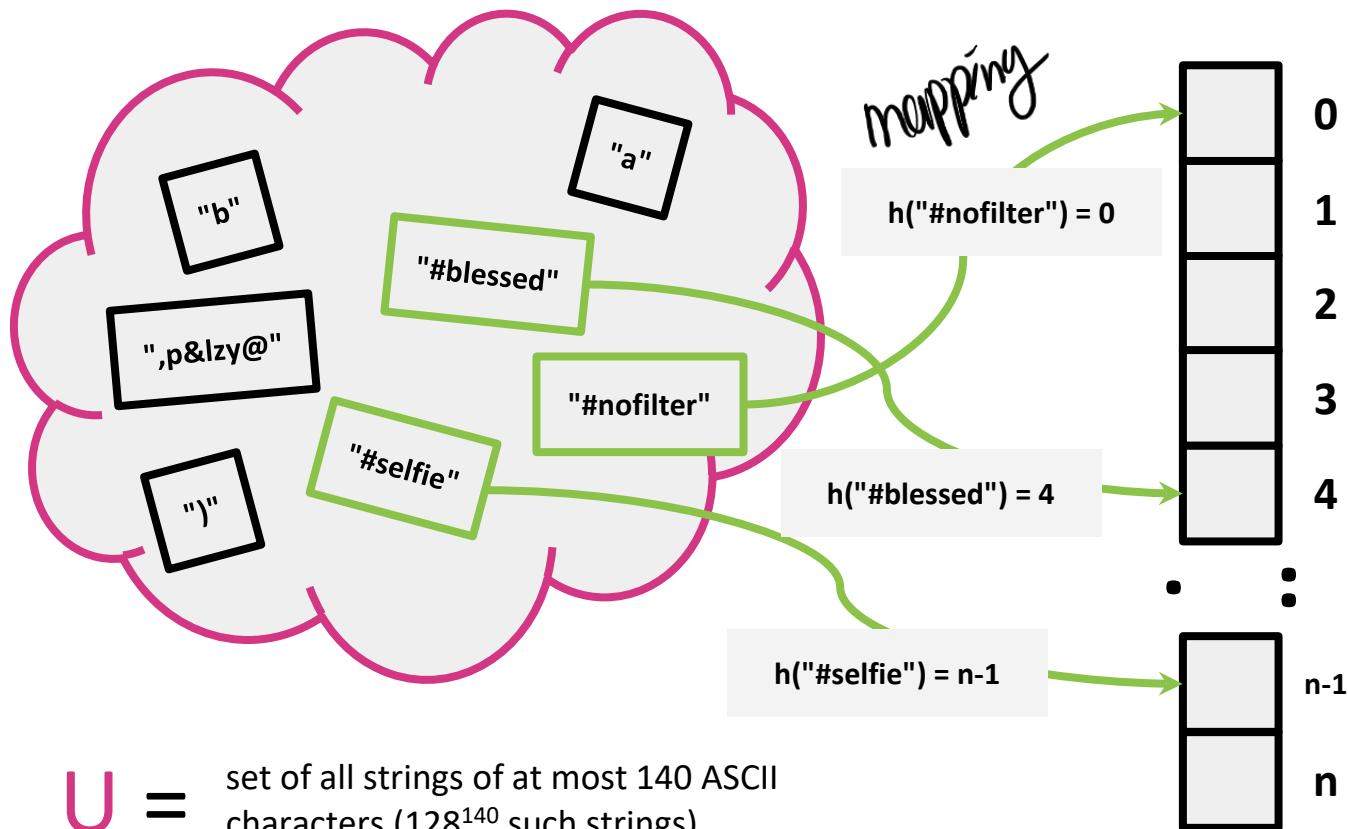


U = set of all strings of at most 140 ASCII characters (128^{140} such strings)

And we'll need to store a small subset of U
(say, the ones that might be trending hashtags on Twitter);
There are way fewer than 128^{140} of these.



An Example

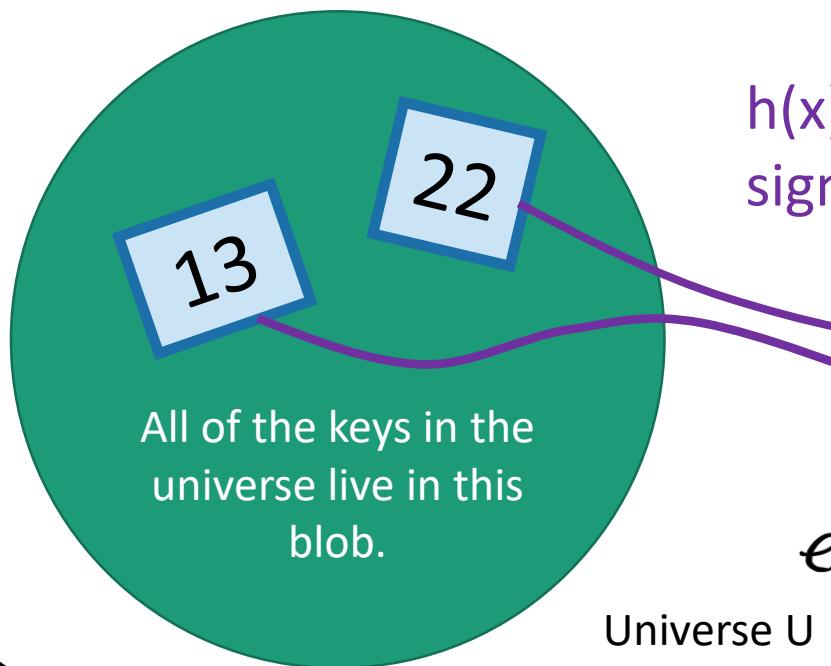


And we'll need to store a small subset of U
(say, the ones that might be trending hashtags on Twitter);
There are way fewer than 128^{140} of these.

- # The previous example with this terminology
- $n \ll M$
- We have a **universe U** , of size M .
 - at most n of which will show up.
 - M is **waaaayyyyyy** bigger than n .
 - We will put items of U into n **buckets**.

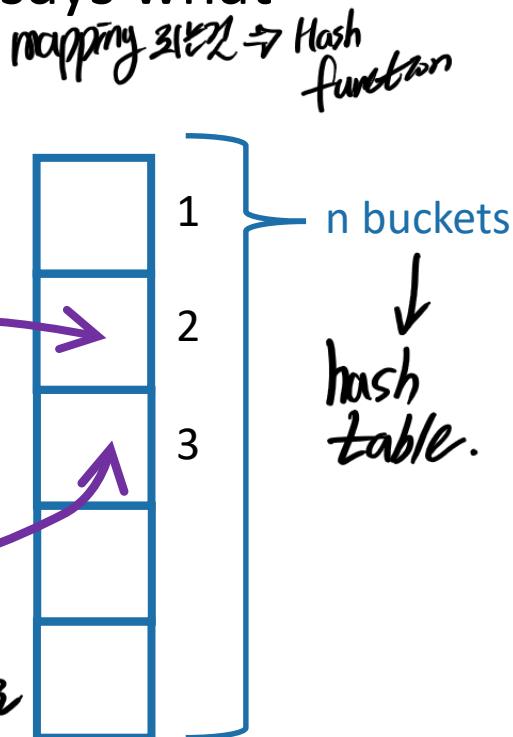
→ h maps the universe U of keys into the slots of hash table

- An element with key u_i hashes to slot $h(u_i)$
- $h(u_i)$ is the hash value of key u_i



$h(x) =$ least significant digit of x .

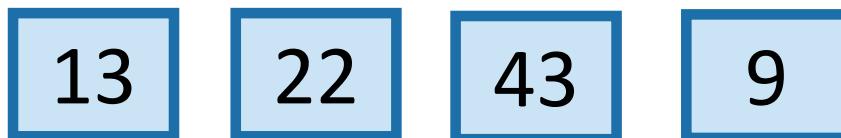
$h(22) = 2$
 $h(13) = 3$
 ex) least significant digit of key?



This is a hash table (with chaining)

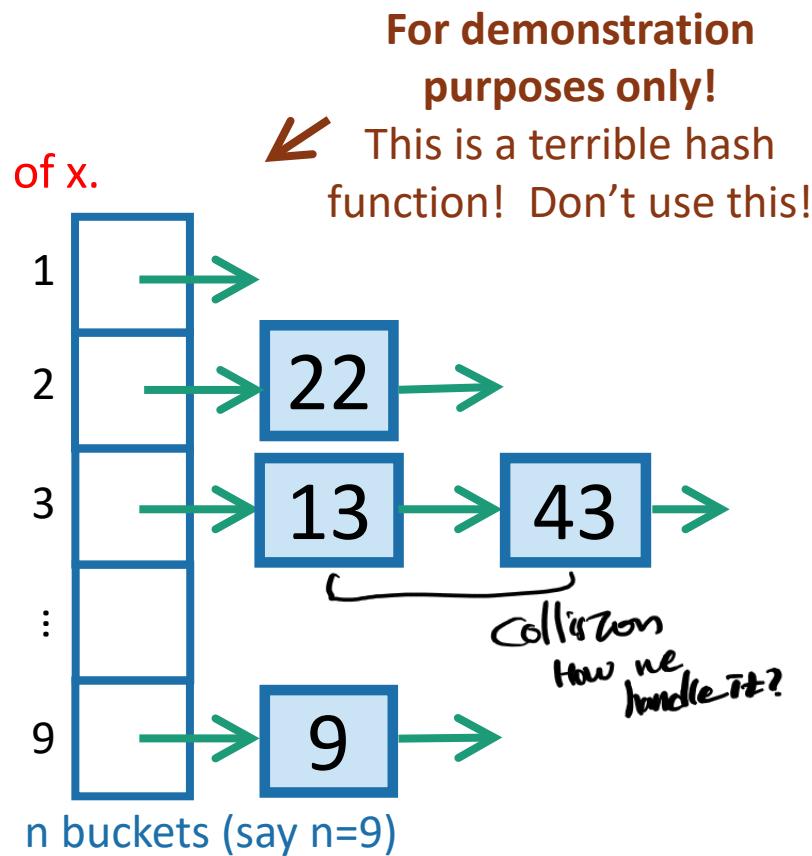
- Array of n buckets.
- Each bucket stores an unsorted linked list.
 - insert in $O(1)$ since it's unsorted;
 - search in $O(\text{length(list)})$.
- $h:U \rightarrow \{1, \dots, n\}$ can be any function:
 - but for concreteness,
let's stick with $h(x) = \text{least significant digit of } x$.

INSERT:



SEARCH 43:

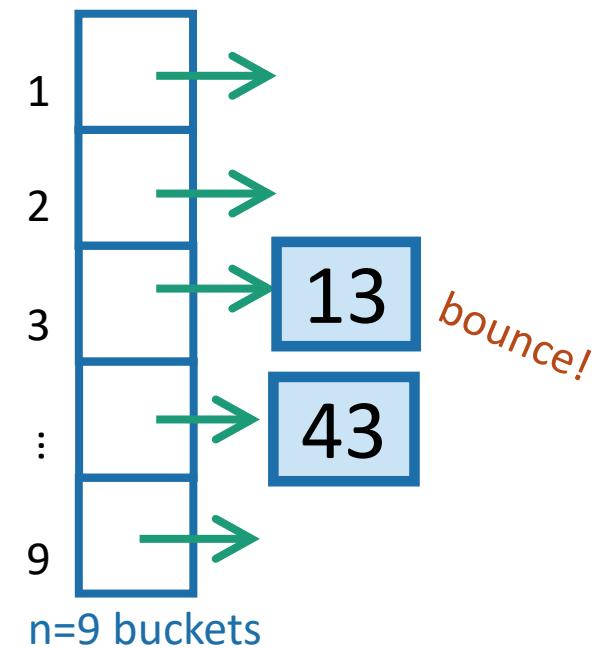
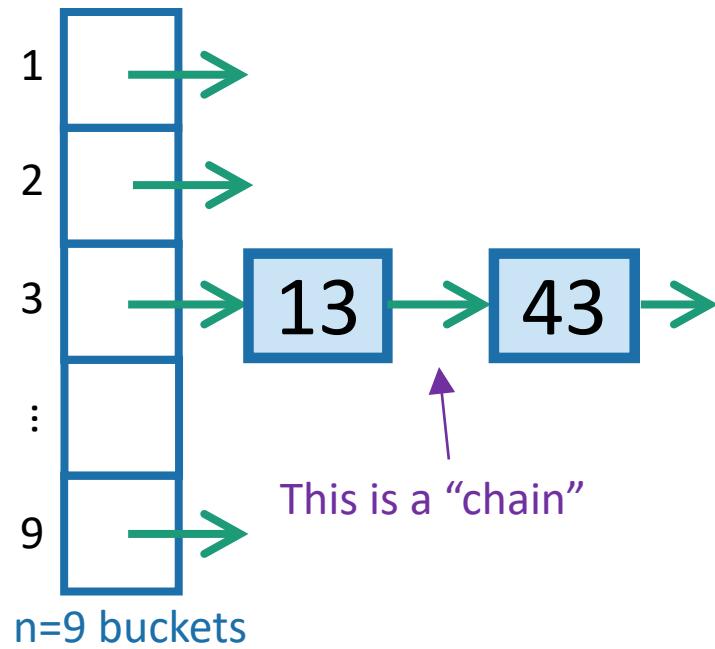
Scan through all the elements in
bucket $h(43) = 3$.



Aside: Hash tables with open addressing

From collection-oriented to open addressing

- The previous slide is about hash tables **with chaining**.
- There's also something called "**open addressing**"
- Read in CLRS 11.4 if you are interested!

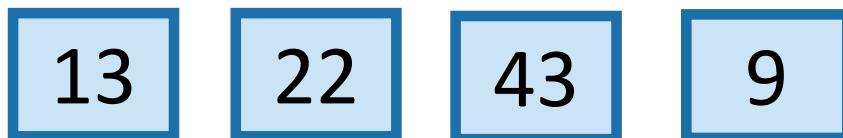


\end{Aside}

This is a **hash table** (with chaining)

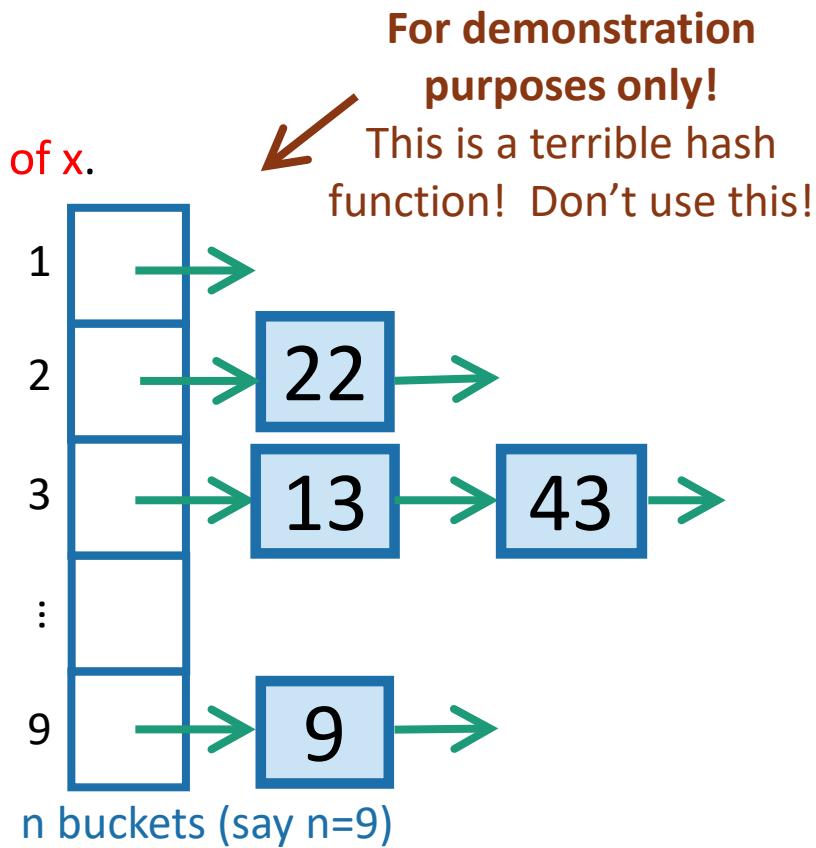
- Array of n buckets.
- Each bucket stores a linked list.
 - insert in $O(1)$ since it's unsorted;
 - search in $O(\text{length(list)})$.
- $h:U \rightarrow \{1, \dots, n\}$ can be any function:
 - but for concreteness,
let's stick with $h(x) = \text{least significant digit of } x$.

INSERT:



SEARCH 43:

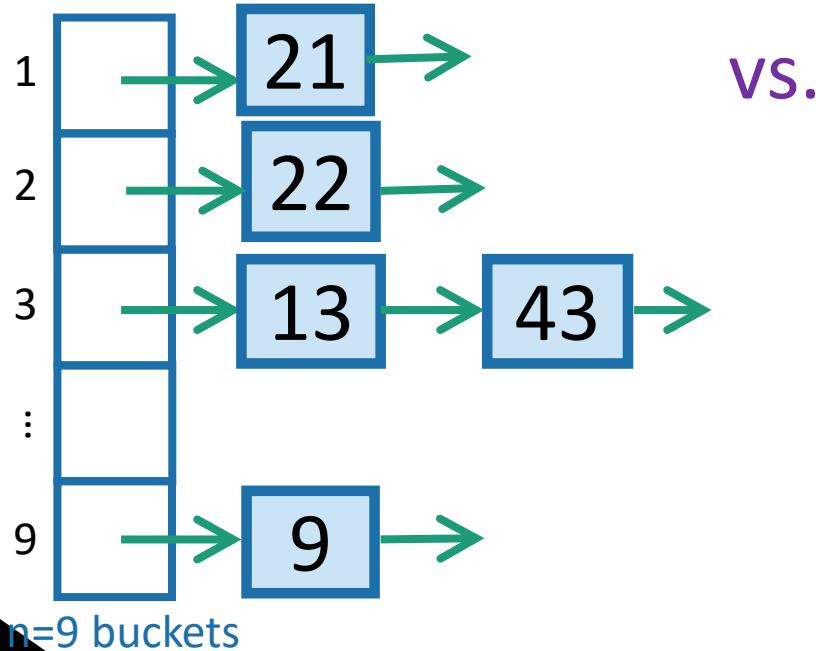
Scan through all the elements in
bucket $h(43) = 3$.



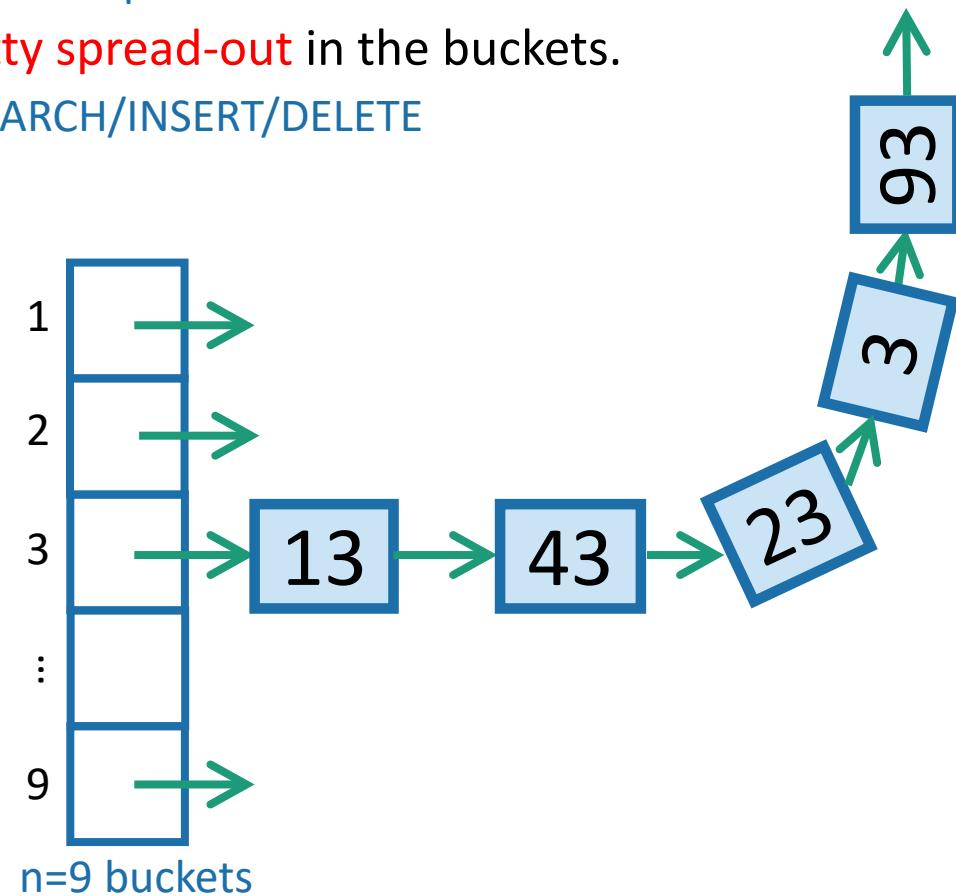
Sometimes this is a good idea

Sometimes this is a bad idea

- How do we pick that function so that this is a good idea?
 1. We want there to be not many buckets (say, n).
 - This means we don't use too much space
 2. We want the items to be pretty spread-out in the buckets.
 - This means it will be fast to SEARCH/INSERT/DELETE



vs.



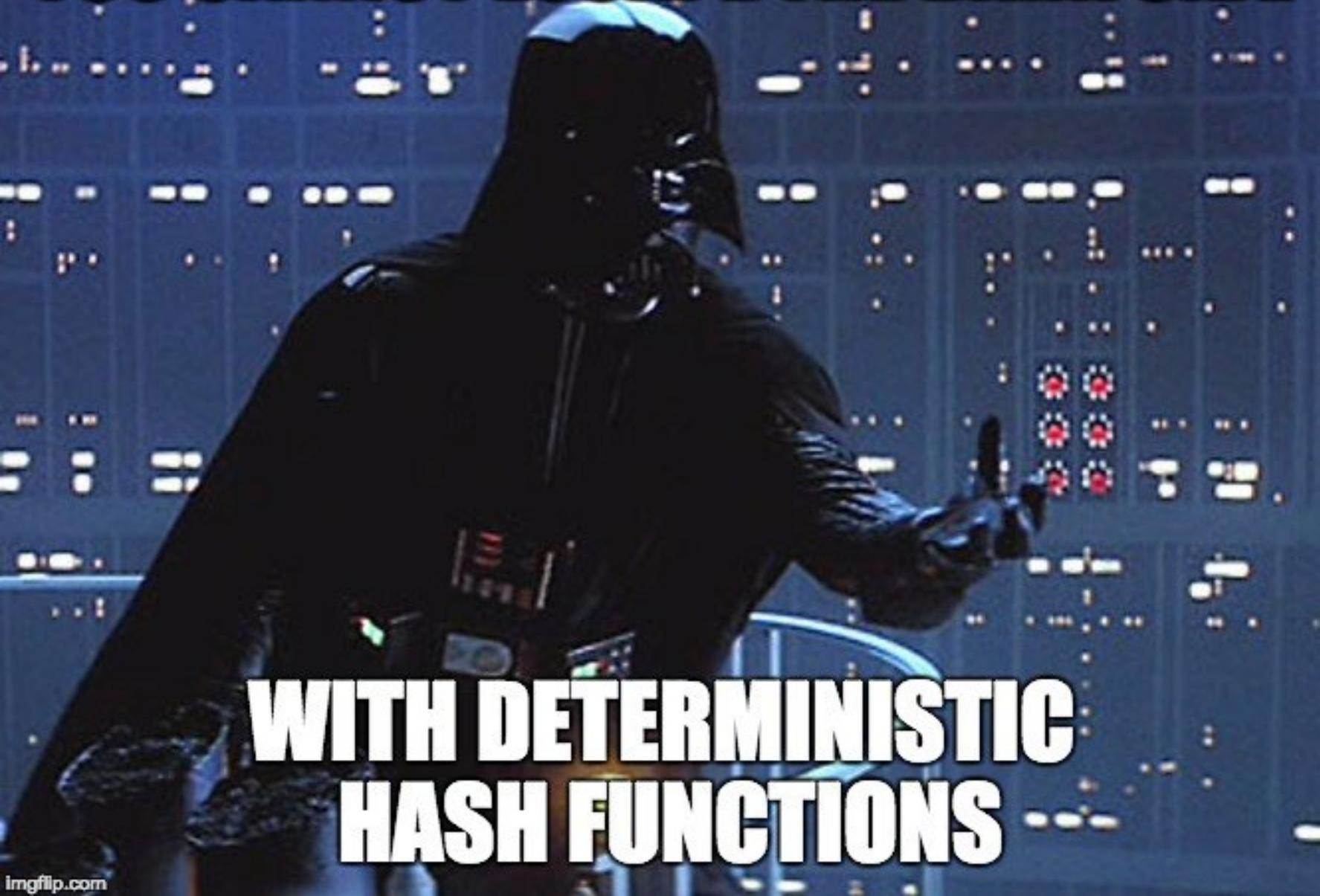
Worst-case analysis

- Design a function $h: U \rightarrow \{1, \dots, n\}$ so that:
 - No matter what input (*fewer than n items of U*) a **bad guy** chooses, the buckets will be **balanced**.
 - Here, **balanced** means $O(1)$ entries per bucket.
- If we had this, then we'd achieve our dream of $O(1)$ **INSERT/DELETE/SEARCH**

Can you come up with
such a function?



YOU CANNOT ESCAPE THE DARK SIDE

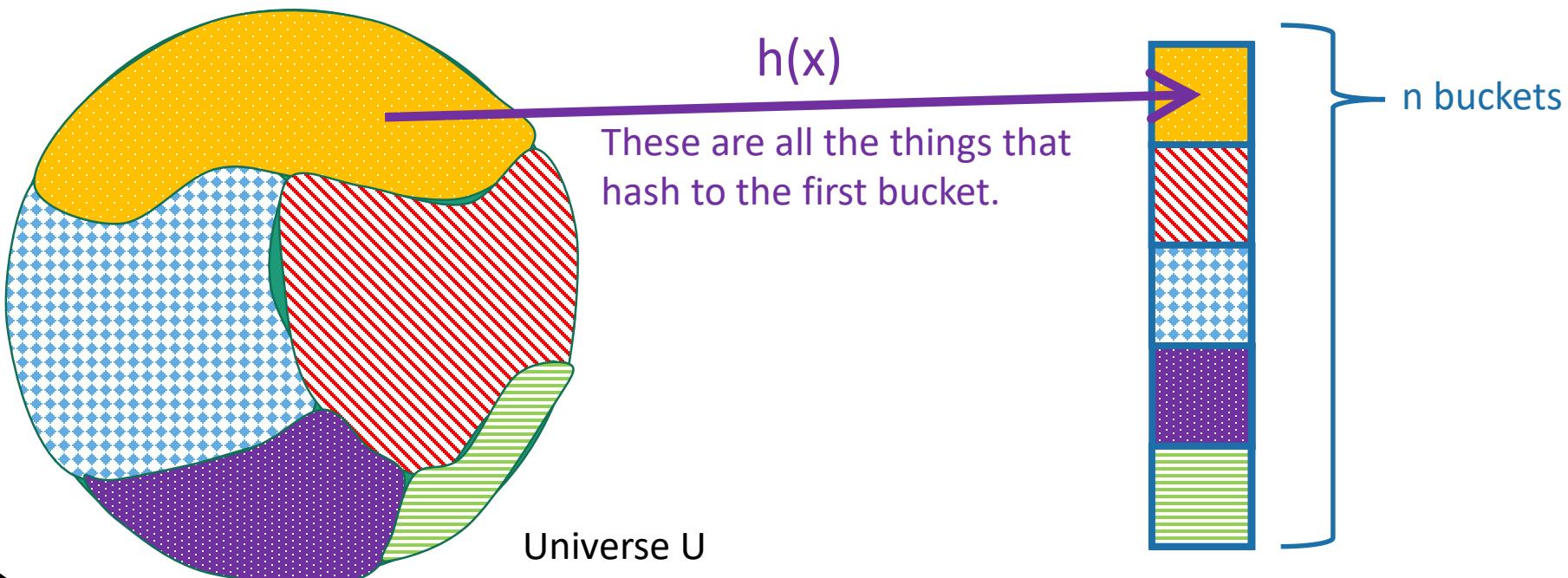
A dark-themed image of Darth Vader in his control room. He is wearing his iconic black helmet and suit, standing in front of a wall covered in glowing blue and red control panels. His right hand is raised, pointing towards the camera. The overall atmosphere is mysterious and powerful.

**WITH DETERMINISTIC
HASH FUNCTIONS**

We really can't beat the bad guy here.

- The universe U has M items
- They get hashed into n buckets
- At least one bucket has at least M/n items hashed to it.
- M is WAAYYYYYY bigger then n , so M/n is bigger than n .
- Bad guy chooses n of the items that landed in this very full bucket.

비둘기집 원리



Solution: Randomness

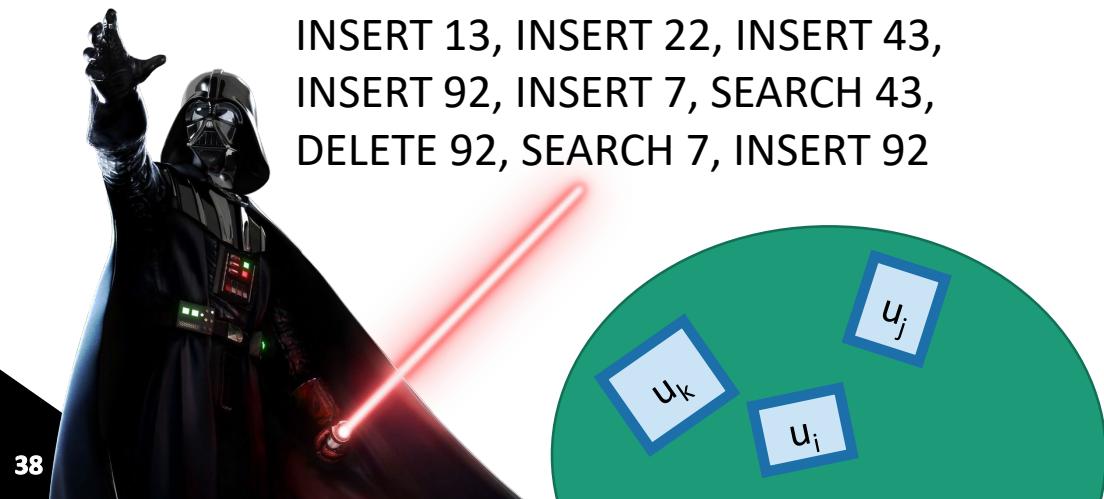


The game

1. An adversary chooses any n items $u_1, u_2, \dots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.



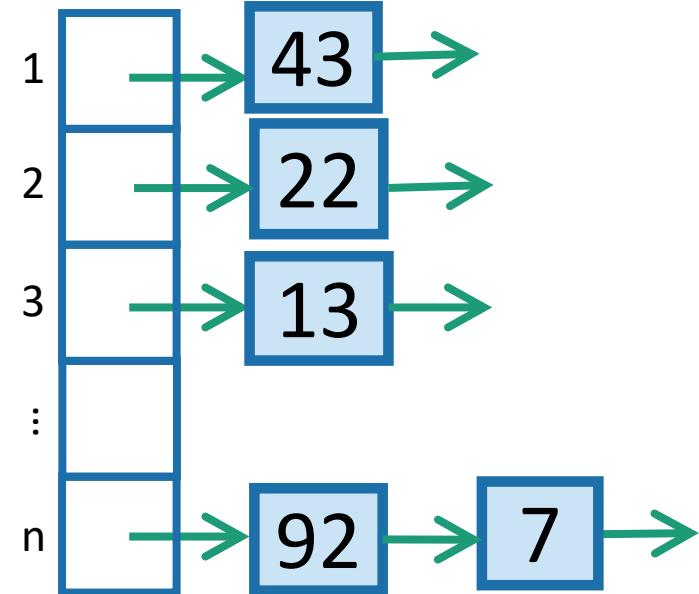
INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92



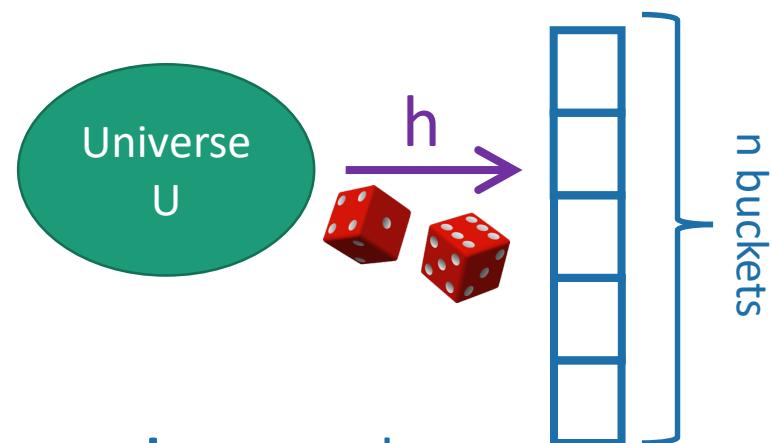
2. You, the algorithm, chooses a **random hash function** $h: U \rightarrow \{1, \dots, n\}$.



3. HASH IT OUT



Example



- Say that h is uniformly random.
 - That means that $h(1)$ is a uniformly random number between 1 and n .
 - $h(2)$ is also a uniformly random number between 1 and n , independent of $h(1)$.
 - $h(3)$ is also a uniformly random number between 1 and n , independent of $h(1), h(2)$.
 - ...
 - $h(n)$ is also a uniformly random number between 1 and n , independent of $h(1), h(2), \dots, h(n-1)$.

Why should that help?

Intuitively: The bad guy can't foil a hash function that he doesn't yet know.

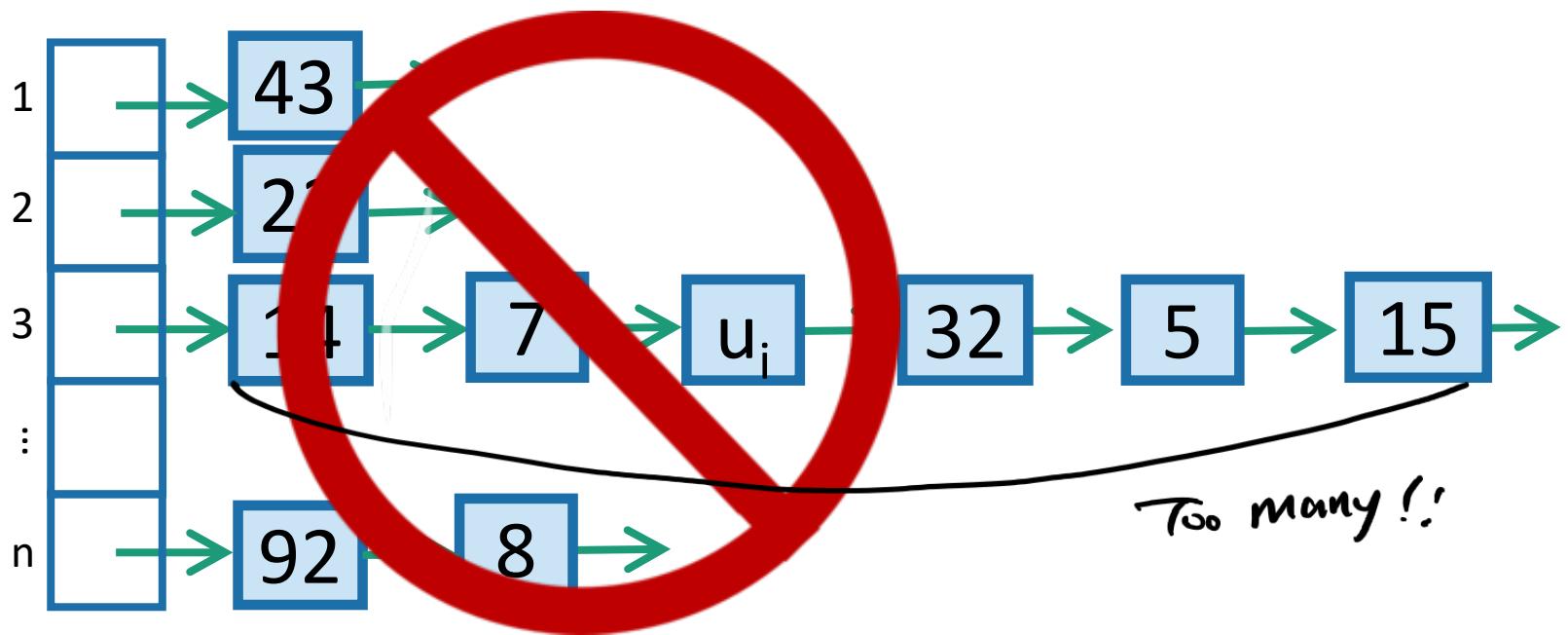


We'll need to do some analysis...

What do we want?

It's **bad** if lots of items land in u_i 's bucket.

So we want **not** that.

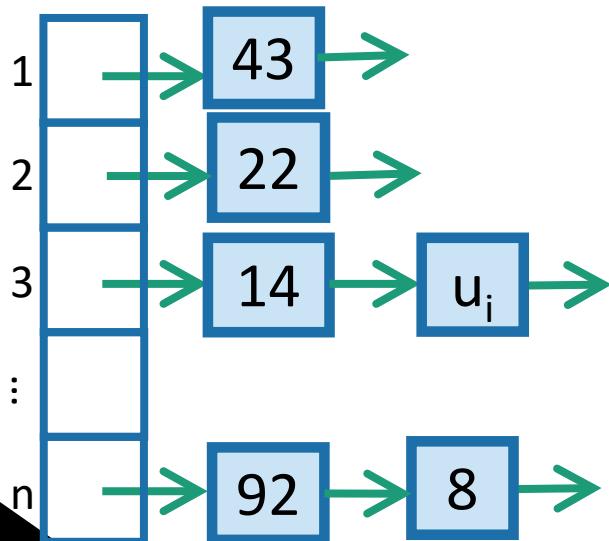


More precisely

- We want:
 - For all u_i that the bad guy chose
 - $E[\text{ number of items in } u_i \text{ 's bucket }] \leq 2.$
- If that were the case,
 - For each operation involving u_i
 - $E[\text{ time of operation }] = O(1)$

We could replace "2" here with any constant; it would still be good.

So, in expectation,
it would takes $O(1)$ time per
INSERT/DELETE/SEARCH
operation.



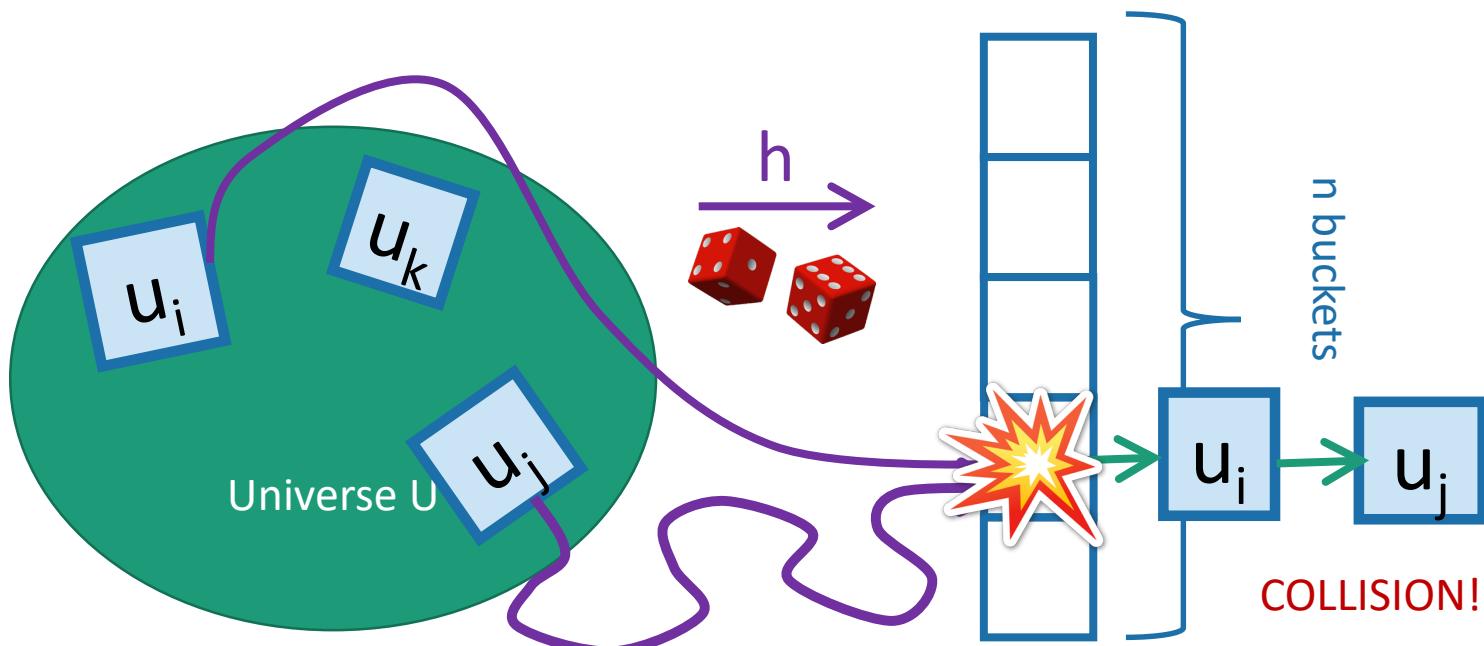
So we want:

- For all $i=1, \dots, n$,
 $E[\text{ number of items in } u_i \text{'s bucket }] \leq 2.$

Expected number of items in u_i 's bucket?

- $E[\quad] = \sum_{j=1}^n P\{ h(u_i) = h(u_j) \}$
 - $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j) \}$
 - $= 1 + \sum_{j \neq i} 1/n$ HOW?
 - $= 1 + \frac{n-1}{n} \leq 2.$
- ↑
in buckettail
item of size
12234

That's what we wanted.



$$P(h(U_i) = h(U_j)) = \frac{1}{n}$$

$$= \sum_{k=1}^n P(h(U_i) = k) \cdot \underbrace{P(h(U_i) = h(U_j) \mid h(U_i) = k)}_{P(h(U_j) = k)}$$

$$= \sum_{k=1}^n \frac{P(h(U_i) = k)}{\frac{1}{n}} \cdot \frac{P(h(U_j) = k)}{\frac{1}{n}}$$

$$= \sum_{k=1}^n \frac{1}{n^2} = \frac{1}{n}$$

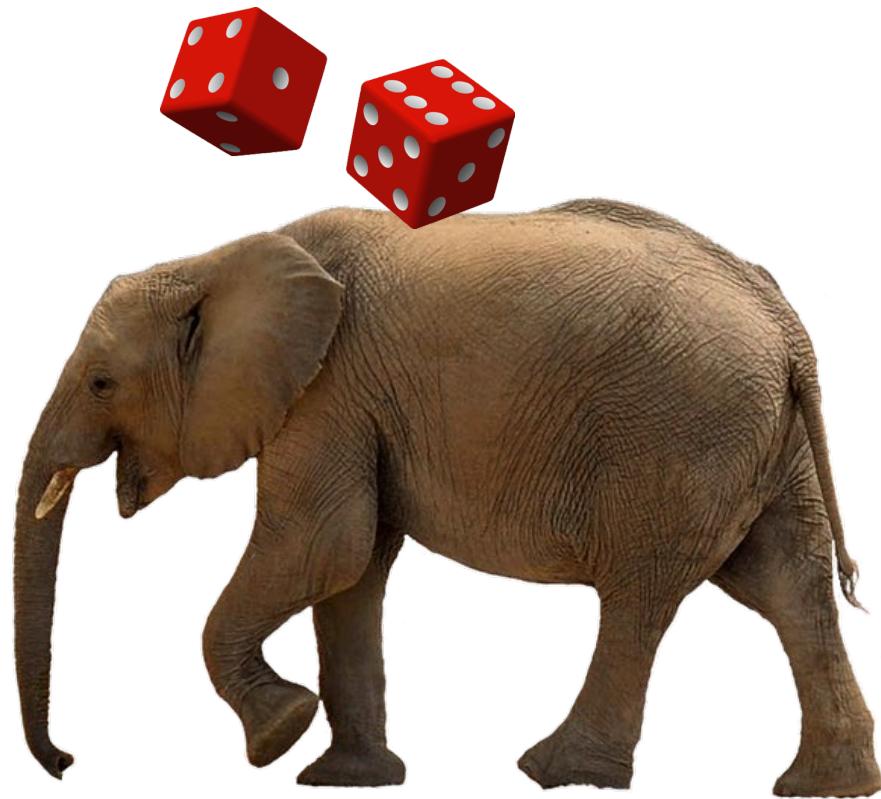
That's great!

- For all $i=1, \dots, n$,
 - $E[\text{ number of items in } u_i \text{ 's bucket}] \leq 2$
- This implies (as we saw before):
 - For any sequence of INSERT/DELETE/SEARCH operations on any n elements of U , the expected runtime (over the random choice of h) is ***O(1) per operation.***

So, the solution is:

pick a uniformly random hash function.

The elephant in the room



The elephant in the room



“Pick a uniformly
random hash function”

How do we do that?



The Bad News

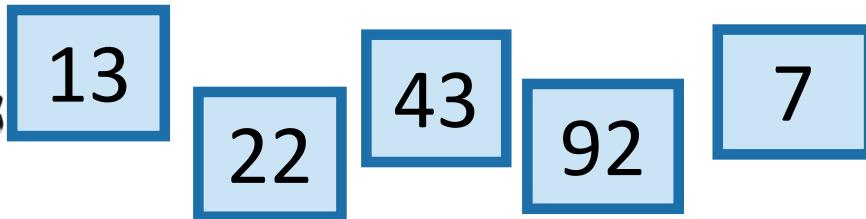
Issues:

- Suppose $U = \{ \text{all of the possible hashtags} \}$
- If we completely choose the random function up front, we have to iterate through all of U .
 - 128^{140} possible ASCII strings of length 140.
 - (More than the number of particles in the universe)
- And even ignoring the time considerations
 - We have to store $h(x)$ for every x .

Another thought...

- Just remember h on the relevant values

저장해야 한다.



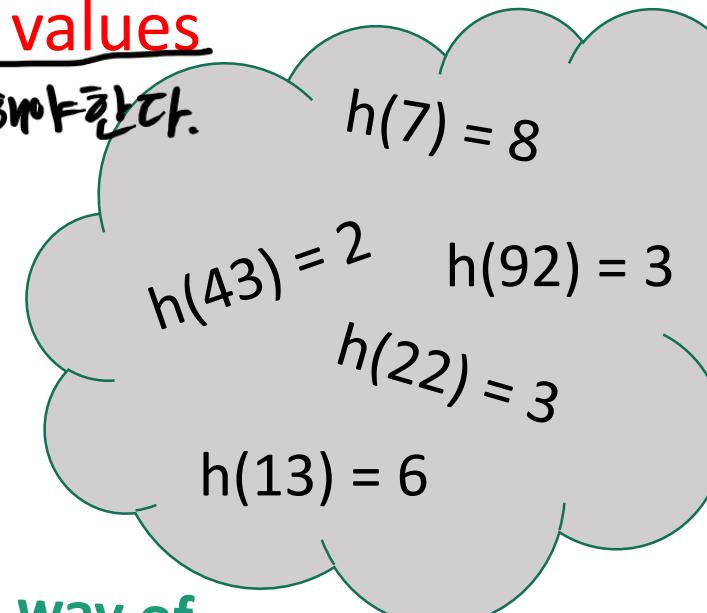
$$h(13) = 6$$

$$h(22) = 3$$

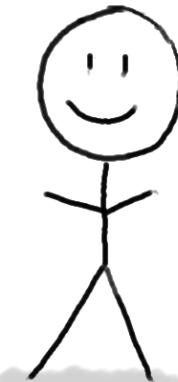
$$h(43) = 2$$

$$h(92) = 3$$

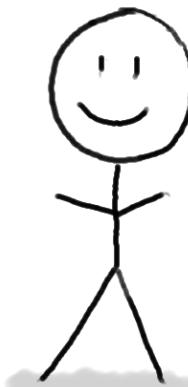
$$h(7) = 8$$



We need some way of
storing keys and values
with $O(1)$
INSERT/DELETE/SEARCH...



Algorithm now



Algorithm later

How much space does it take to store h?

얼마나 많은 공간이 필요

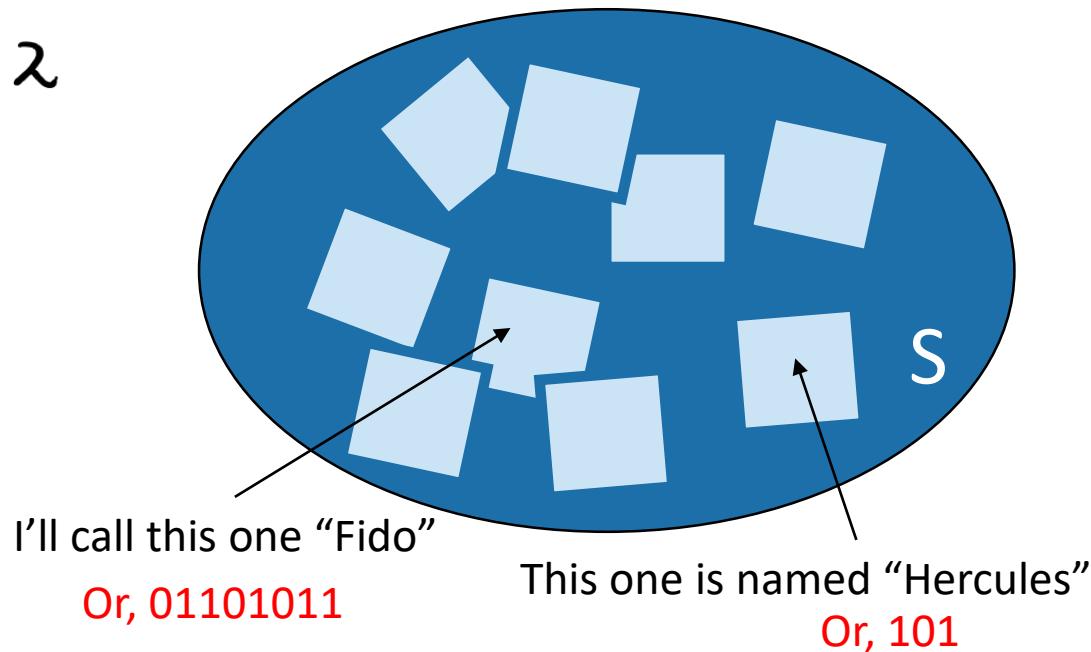
- Say I have a set S with s things in it.
- I get to write down the elements of S however I like.
 - (in binary)
- How many bits do I need?

ex) $S = \{a, b, c\}, n=2$

$$|H| = n^{\log_2} = 8$$

$$\log |H| = 3 \text{ bit}$$

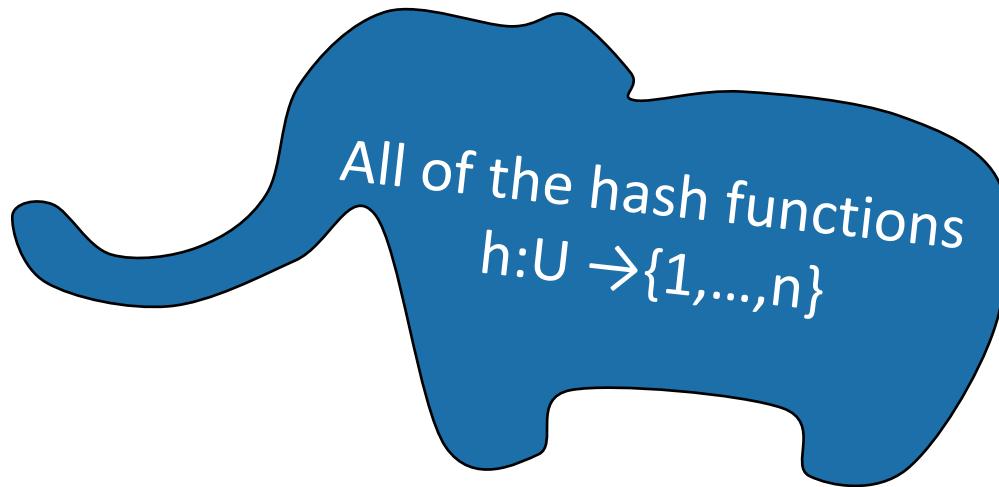
g : ; ; ; ; ; ; ;
c : ; ; ; ; ; ; ;



On board: the answer is $\log(s)$

Space needed to store a random fn h ?

- Say that this elephant-shaped blob represents the set of all hash functions.
- It has size n^M . (Really big!)
- To write down a random hash function, we need $\log(n^M) = M\log(n)$ bits. ☹



- In contrast, direct addressing would require M bits.

H Is Too Big

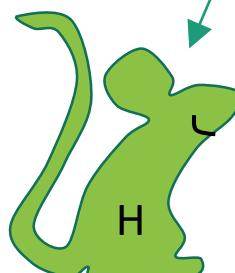
How can we fix this issue of the size of H?

Solution

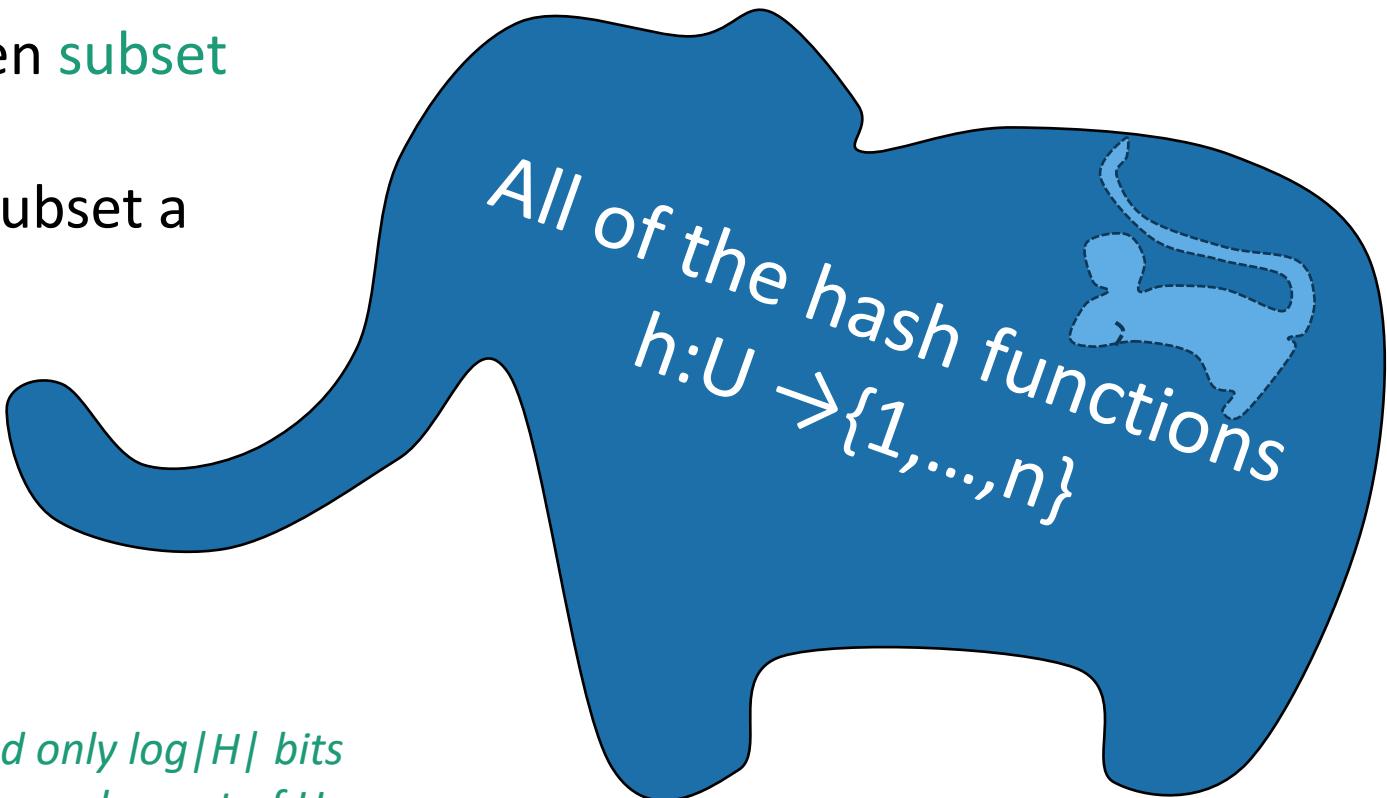
- Pick from a smaller set of functions.

A cleverly chosen **subset** of functions.

We call such a subset a **hash family**.

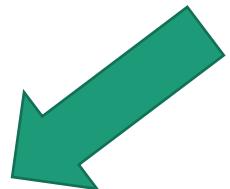


We need only $\log |H|$ bits to store an element of H .



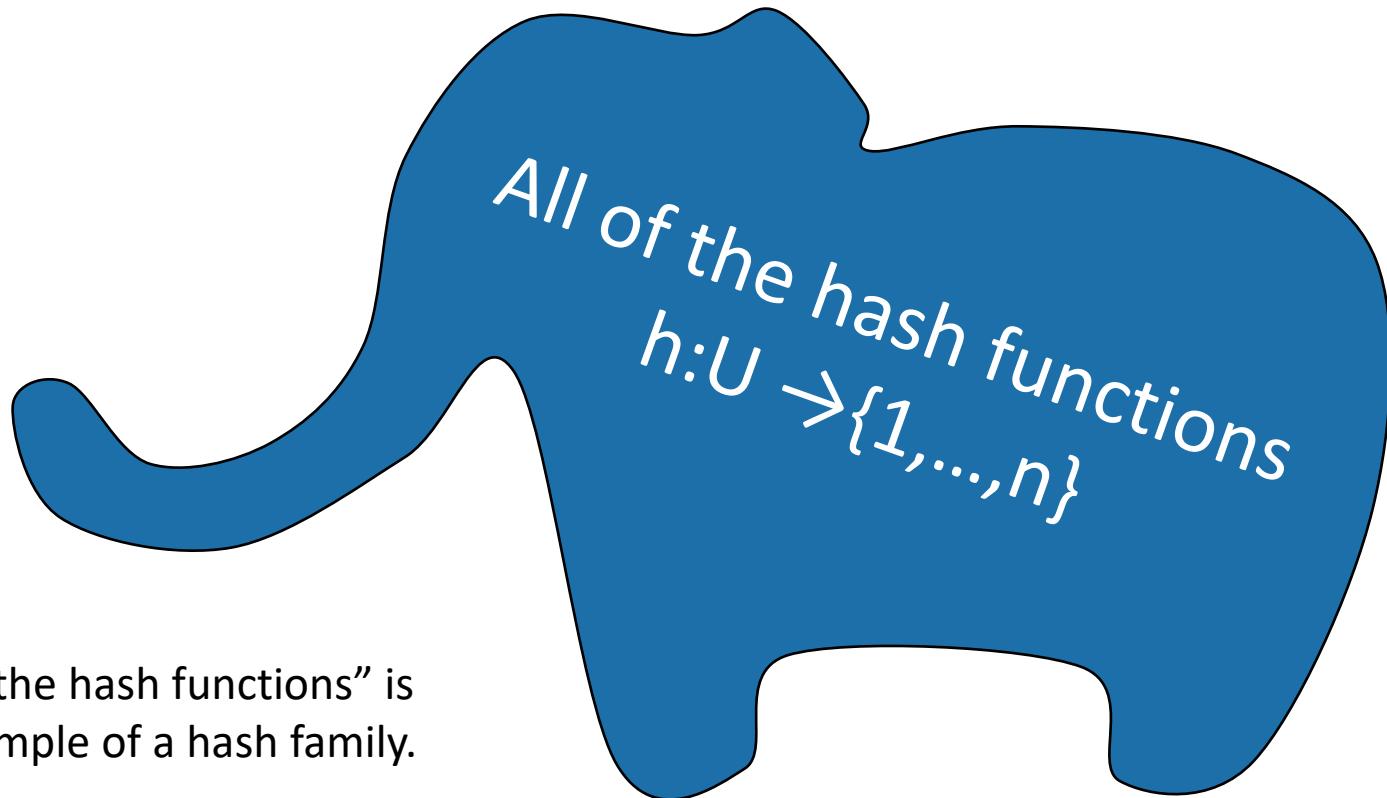
Outline

- Hash tables are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
 - like self-balancing binary trees
 - The difference is we can get better performance in expectation by using randomness.
- Hash families are the magic behind hash tables.
- Universal hash families are even more magic.



Hash families

- A hash family is a collection of hash functions.

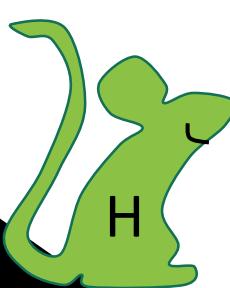
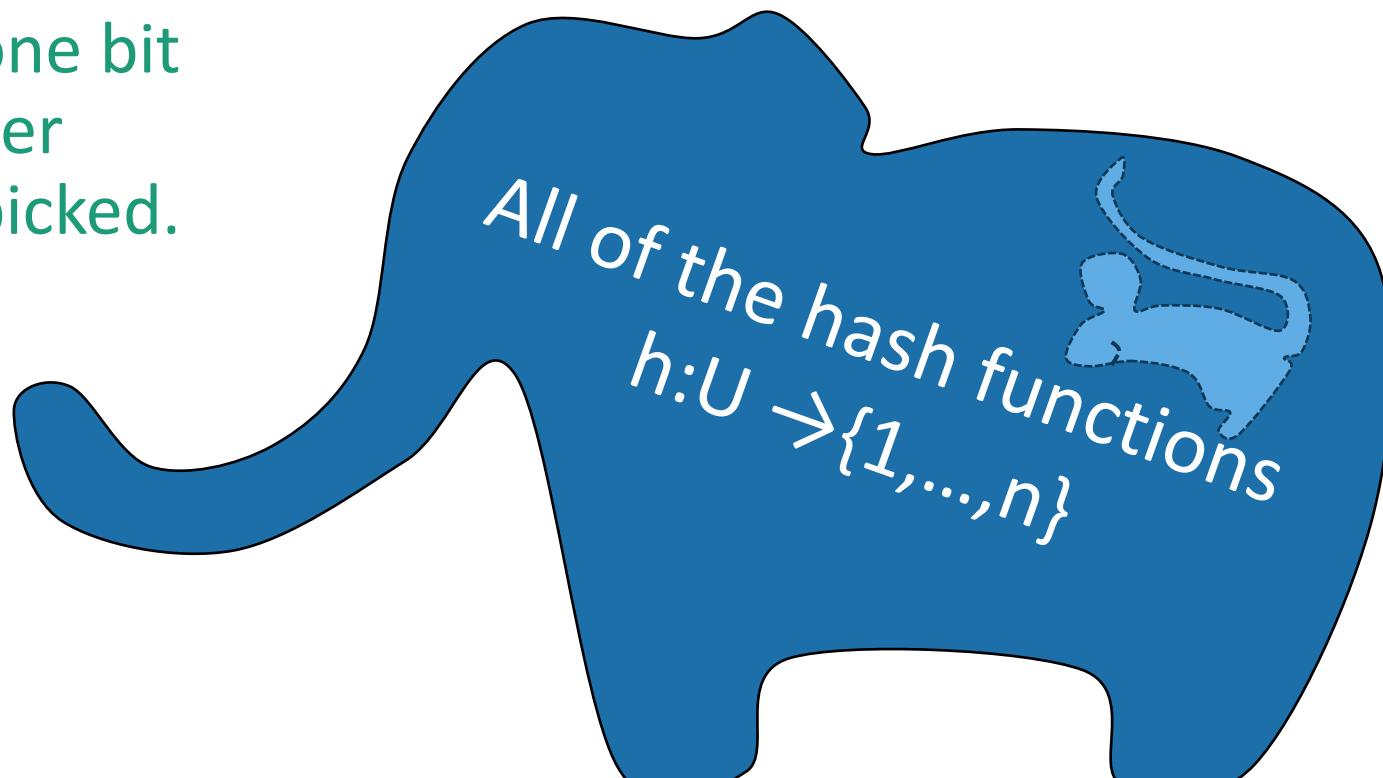


Example:

a smaller hash family

This is still a terrible idea!
Don't use this example!

- $H = \{$ function which returns the least sig. digit,
function which returns the most sig. digit $\}$
- Pick h in H at random.
- Store just one bit
to remember
which we picked.



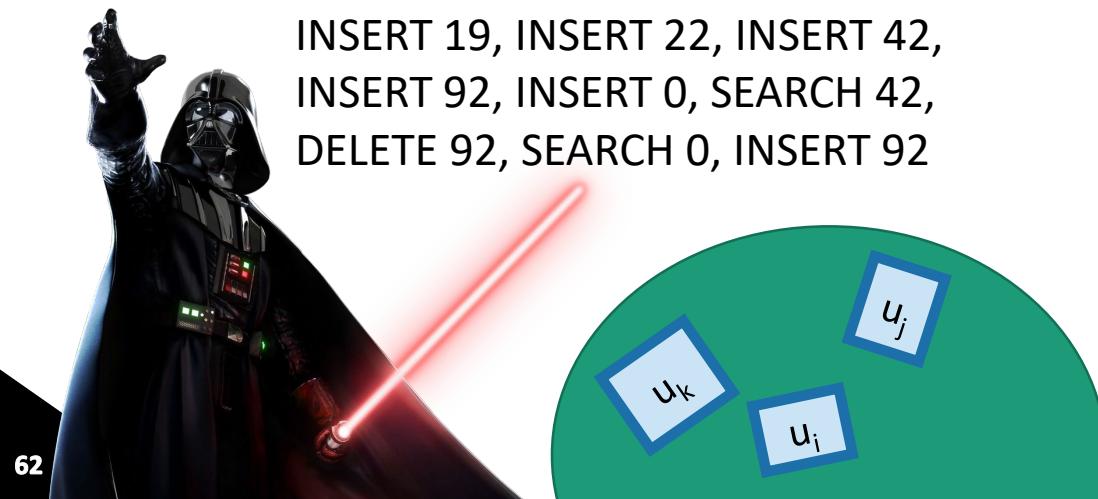
The game

h_0 = Most_significant_digit
 h_1 = Least_significant_digit
 $H = \{h_0, h_1\}$

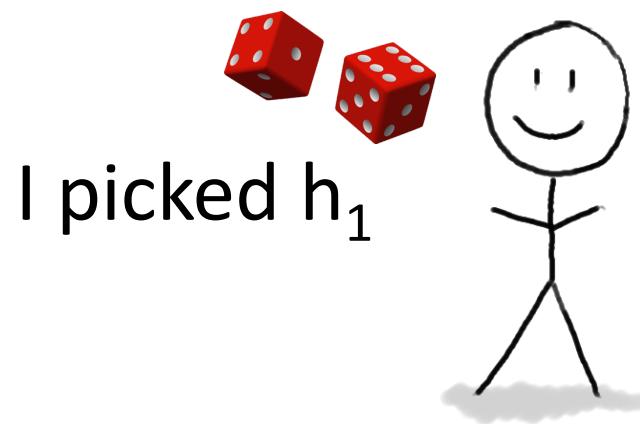
1. An adversary (who knows H) chooses any n items $u_1, u_2, \dots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.



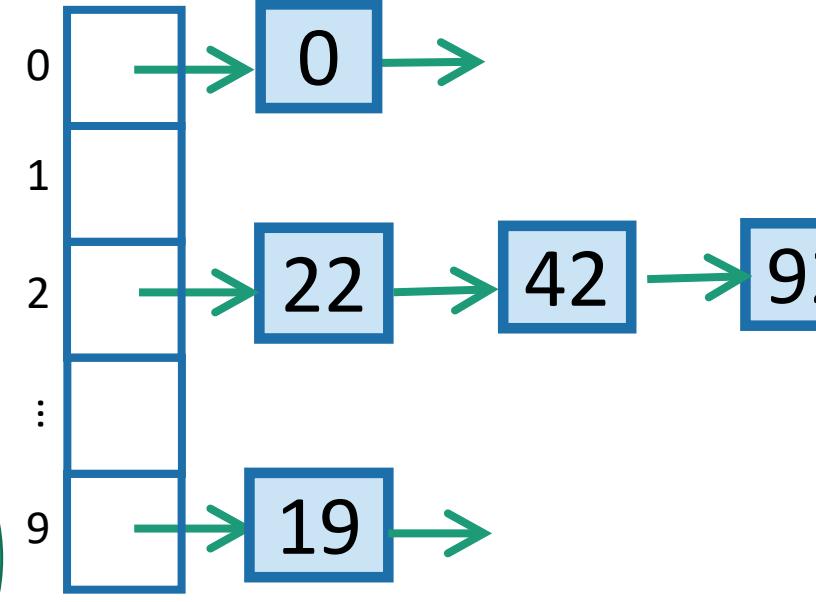
INSERT 19, INSERT 22, INSERT 42,
INSERT 92, INSERT 0, SEARCH 42,
DELETE 92, SEARCH 0, INSERT 92



2. You, the algorithm, chooses a random hash function $h: U \rightarrow \{0, \dots, 9\}$. Choose it randomly from H.



3. HASH IT OUT #hashpuns



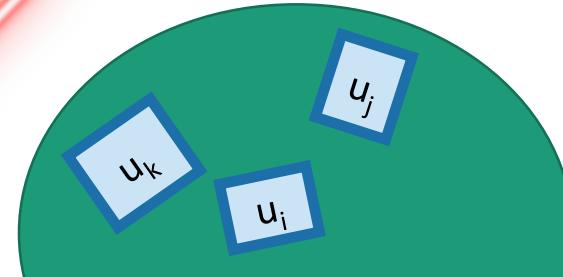
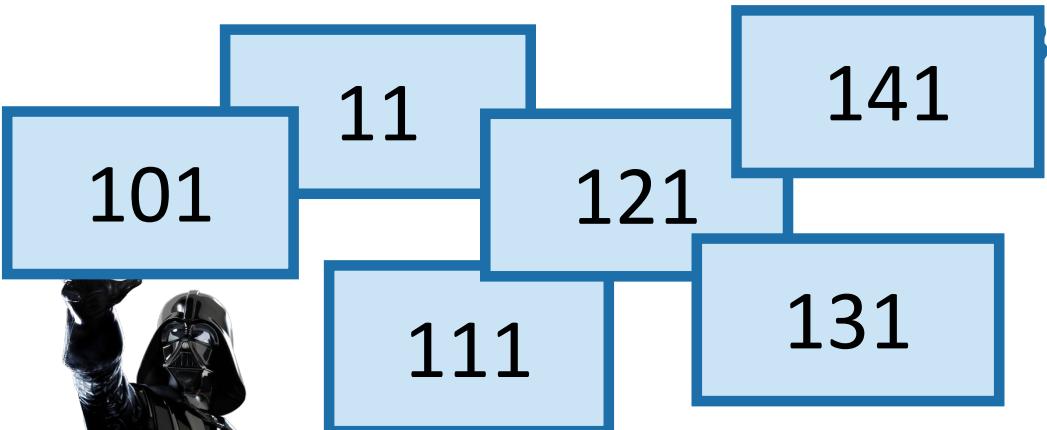
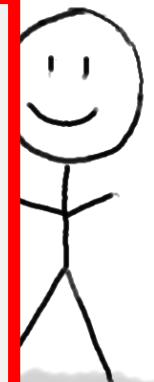
The game

h_0 = Most_significant_digit
 h_1 = Least_significant_digit
 $H = \{h_0, h_1\}$

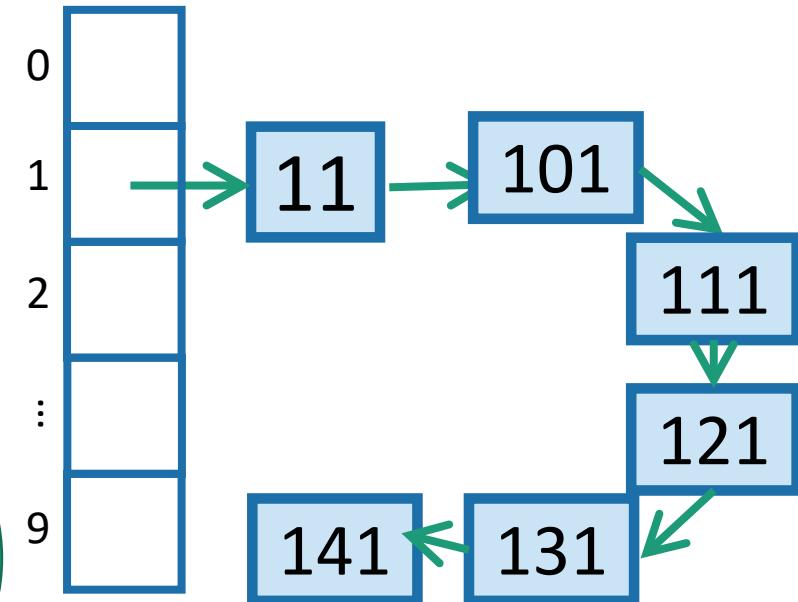
1. An adversary (who knows H) chooses items $u_1, u_2, \dots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

2. You, the algorithm, chooses a random hash function $h: U \rightarrow \{0, \dots, 9\}$. Choose it randomly from H .

This adversary could have been more adversarial!



• HASH IT OUT #hashpuns



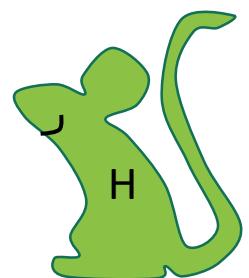
Outline

- **Hash tables** are another sort of data structure that allows fast **INSERT/DELETE/SEARCH**.
 - like self-balancing binary trees
 - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.



How to pick the hash family?

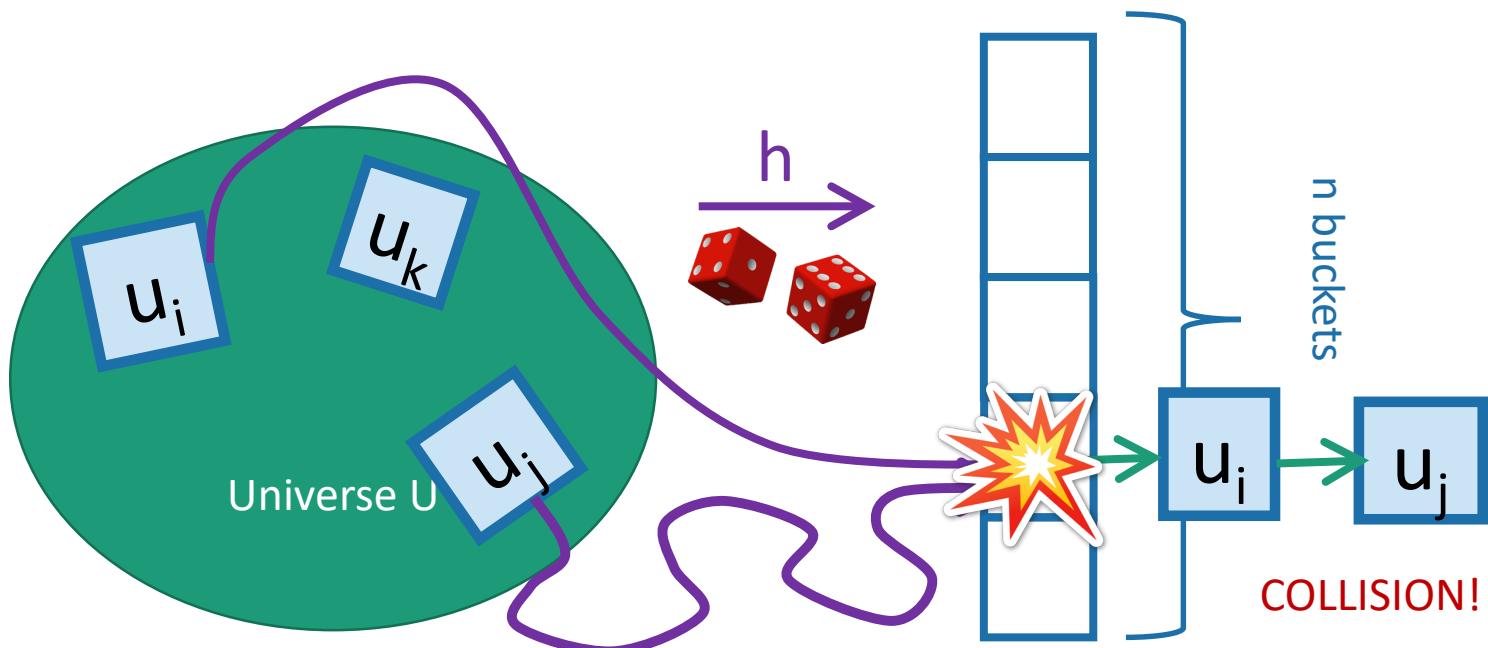
- Definitely not like in that example.
- Let's go back to that computation from earlier....



Expected number of items in u_i 's bucket?

- $E[\quad] = \sum_{j=1}^n P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} 1/n$ HOW?
- $= 1 + \frac{n-1}{n} \leq 2.$

So the number of items in u_i 's bucket is $O(1)$.



How to pick the hash family?

- Let's go back to that computation from earlier....
- $E[\text{ number of things in bucket } h(u_i)]$
- $= \sum_{j=1}^n P\{h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{h(u_i) = h(u_j)\}$
- $\leq 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2.$
- All we needed was that **this** $\leq 1/n$.



Strategy

- Pick a small hash family H , so that when I choose h randomly from H ,

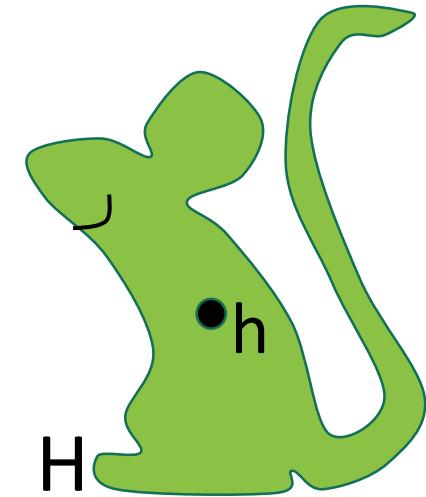
for all $u_i, u_j \in U$ with $u_i \neq u_j$,

$$P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

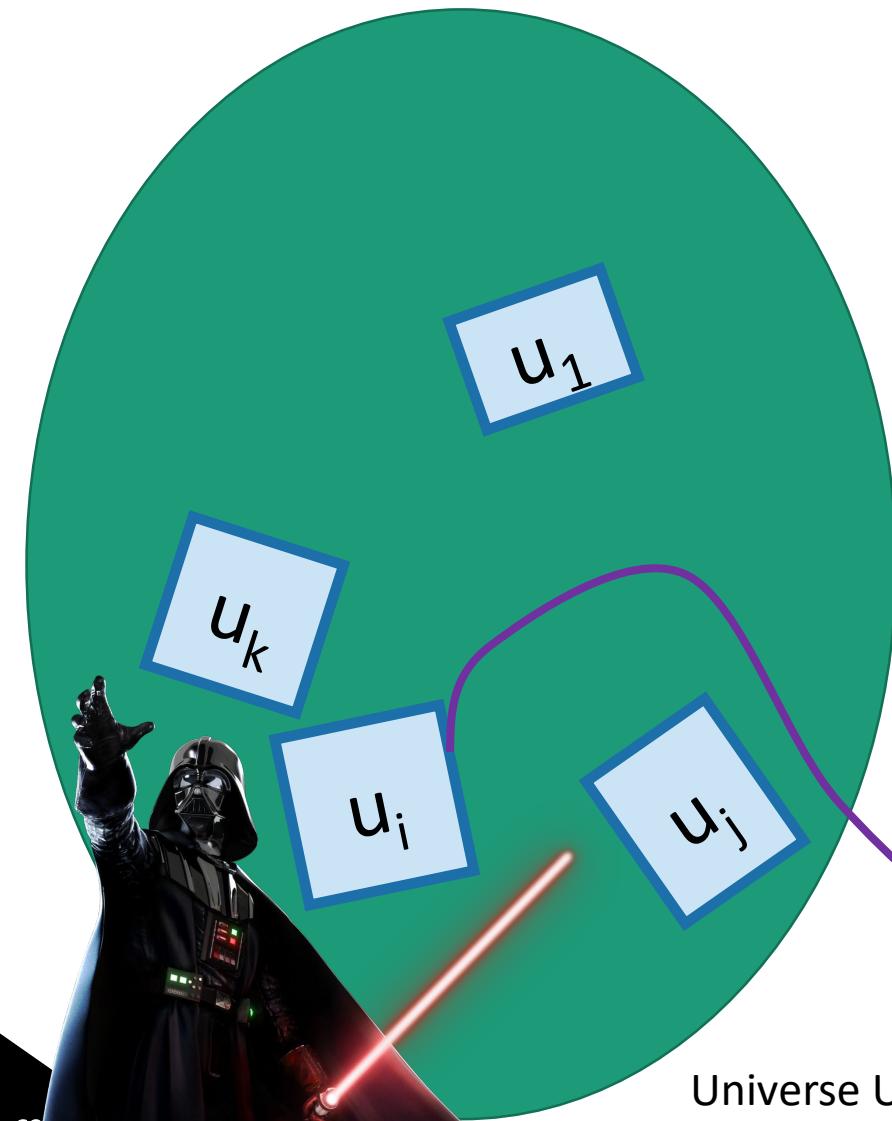
이건 만족해요. Universal hash family

In English: fix any two elements of U .
The probability that they collide under a random h in H is small.

- A hash family H that satisfies this is called a universal hash family.
- Then we still get $O(1)$ -sized buckets in expectation.
- But now the space we need is $\log(|H|)$ bits.
 - Hopefully pretty small!



So the whole scheme will be

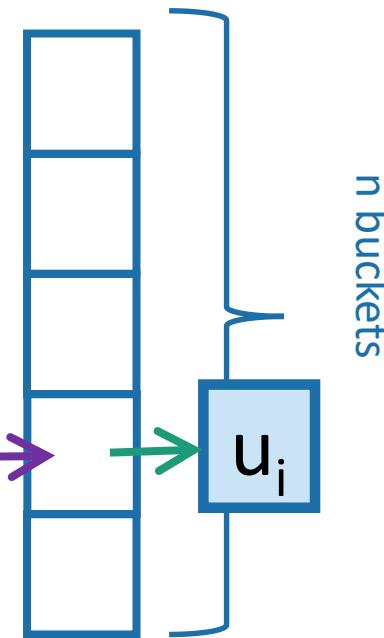


Choose h randomly
from a **universal hash
family H**



We can store h in small space
since H is so small.

Probably
these
buckets will
be pretty
balanced.



Universal hash family

Let's stare at this definition 정의

- H is a *universal hash family* if:
 - When h is chosen uniformly at random from H ,

for all $u_i, u_j \in U$ with $u_i \neq u_j$,

$$P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

Example

- Pick a small hash family H , so that when I choose h randomly from H ,

$$\text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j, \\ P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

- Uniformly random hash function h
 - [We just saw this]
 - [Of course, this one has other downsides...]

A small universal hash family??

- Here's one: To hash an integer x in $\{0, \dots, M-1\}$ to a bucket $\{1, \dots, n\}$:

- Pick a prime $p \geq M$.

- Define

$$f_{a,b}(x) = ax + b \quad \text{mod } p$$

prime number ($\frac{1}{2} \left(\frac{p-1}{2} \right)$)

$$h_{a,b}(x) = f_{a,b}(x) \quad \text{mod } n$$

- Claim:

$$H = \{ h_{a,b}(x) : a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\} \}$$

is a universal hash family.

$$|n| = p(p-1)$$



A small universal hash family??

- Claim:

$$H = \{ h_{a,b}(x) : a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\} \}$$

is a universal hash family.

To select an $h_{a,b}$ from this family:



p

a

b

1. Determine $M = |U|$.

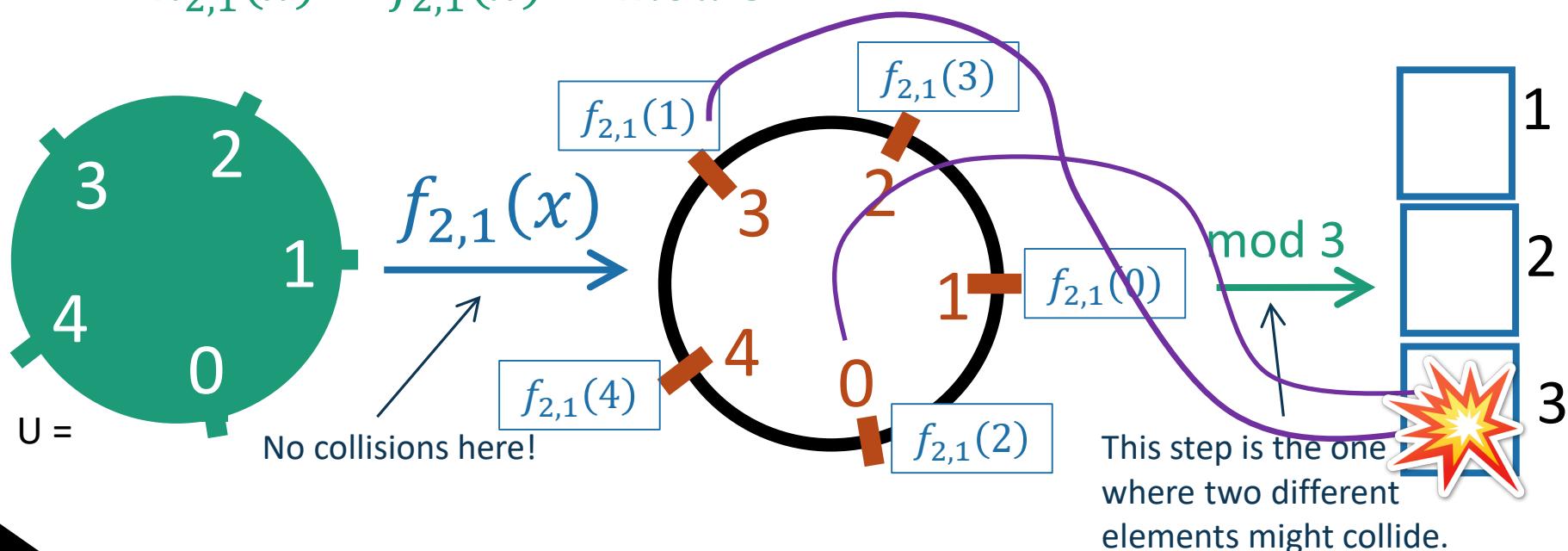
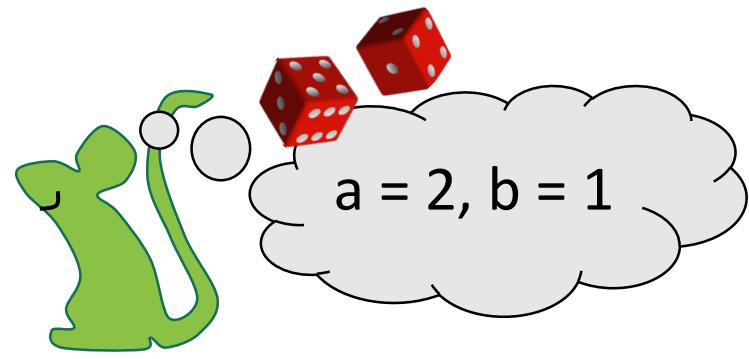
2. Find the smallest prime $p \geq M$.

3. Let a be a random number in $\{1, \dots, p-1\}$.

3. Let b be a random number in $\{0, \dots, p-1\}$.

Say what?

- Example: $M = p = 5$, $n = 3$
- To draw h from H :
 - Pick a random a in $\{1, \dots, 4\}$, b in $\{0, \dots, 4\}$
- As per the definition:
 - $f_{2,1}(x) = 2x + 1 \pmod{5}$
 - $h_{2,1}(x) = f_{2,1}(x) \pmod{3}$



Another way to see this using only the size of H

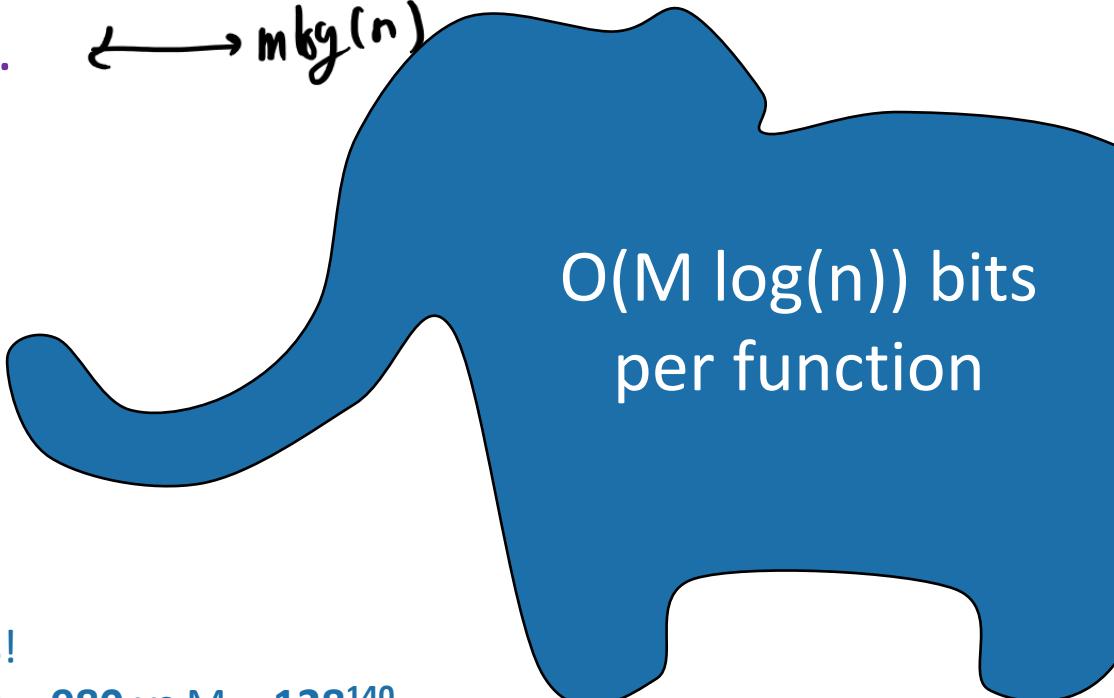
- We have $p-1$ choices for a , and p choices for b .
- So $|H| = p(p-1) = O(M^2)$ $\longleftrightarrow (n^m)$
- Space needed to store an element h :
 - $\log(M^2) = O(\log(M))$. $\longleftrightarrow \log(n)$

$O(\log(M))$ bits
per function



Compare: direct addressing was M bits!

Twitter example: $\log(M) = 140 \log(128) = 980$ vs $M = 128^{140}$

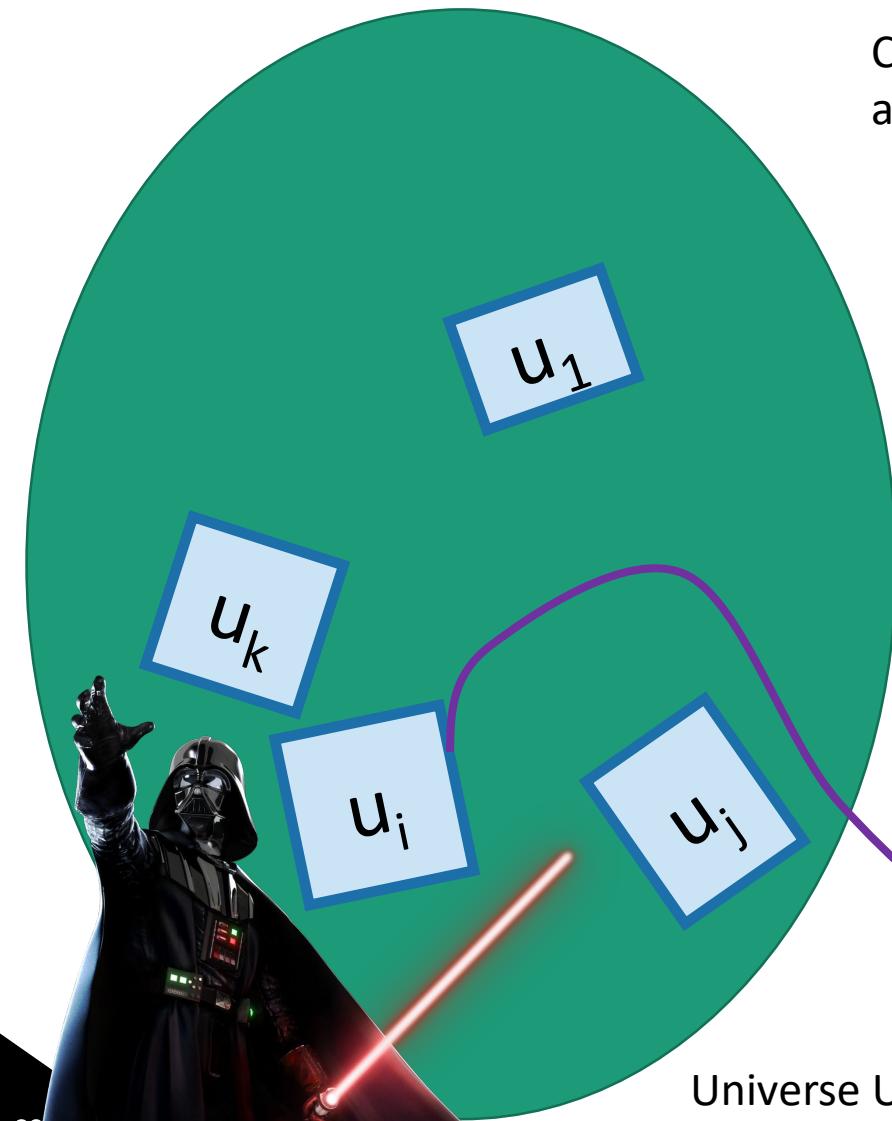


Why does this work?

- This is actually a little complicated.
- The thing we have to show is that the collision probability is not very large.
- **Intuitively**, this is because:
 - for any (fixed, not random) pair $x \neq y$ in $\{0, \dots, p-1\}$,
 - If a and b are random,
 - $ax + b$ and $ay + b$ are independent random variables. (why?)



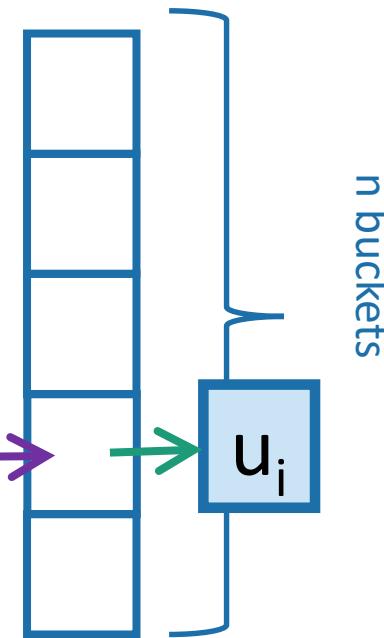
So the whole scheme will be



Choose a and b at random
and form the function $h_{a,b}$



We can store h in space
 $O(\log(M))$ since we just need
to store a and b .



Probably
these
buckets will
be pretty
balanced.

Outline

- Hash tables are another sort of data structure that allows fast **INSERT/DELETE/SEARCH**.
- Hash families are the magic behind hash tables.
- Universal hash families are even more magic.

Recap



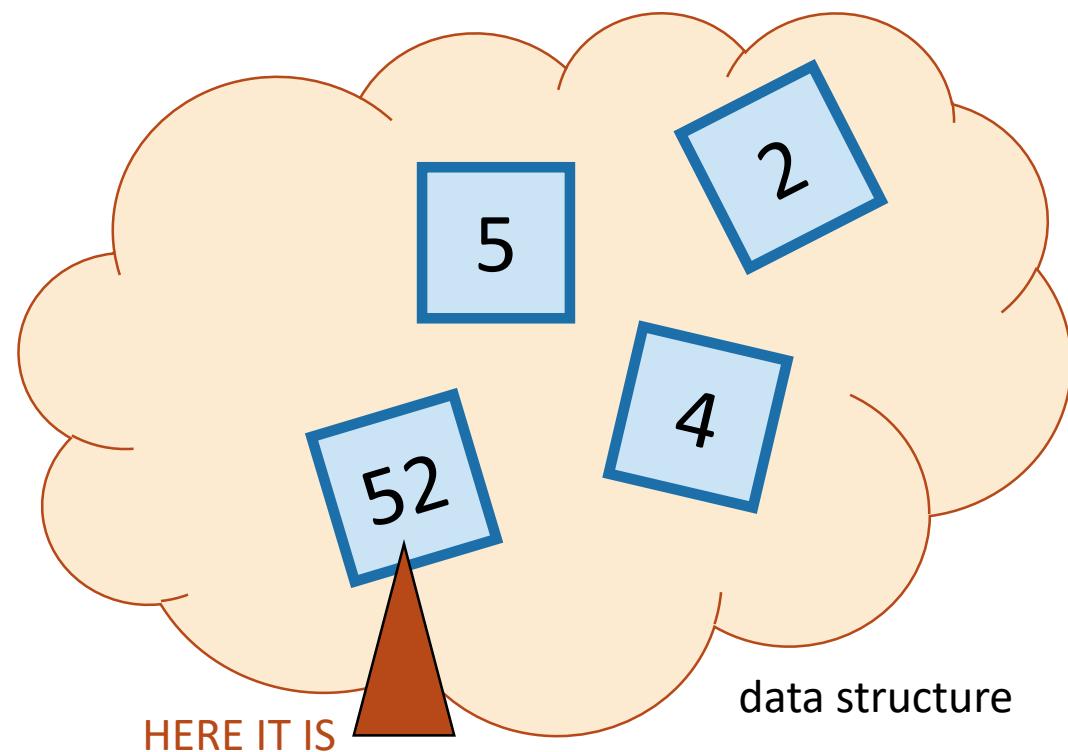
자료구조

Want O(1)

INSERT/DELETE/SEARCH

- We are interesting in putting nodes with keys into a data structure that supports fast
INSERT/DELETE/SEARCH.

- **INSERT** 
- **DELETE** 
- **SEARCH** 

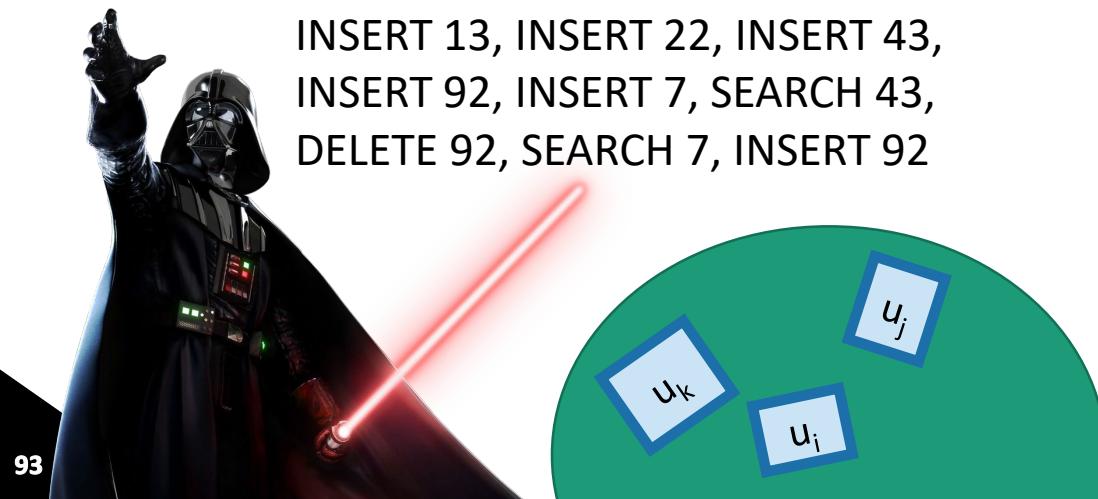


We studied this game

1. An adversary chooses any n items $u_1, u_2, \dots, u_n \in U$, and any sequence of L INSERT/DELETE/SEARCH operations on those items.



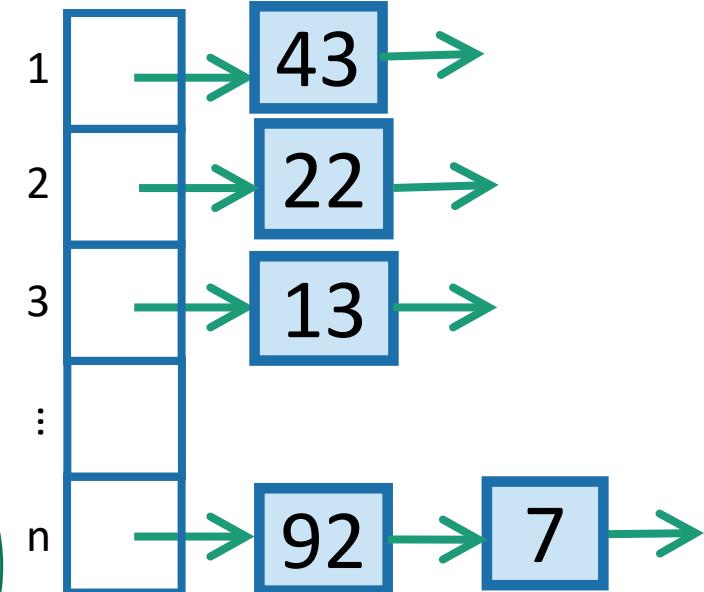
INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92



2. You, the algorithm, chooses a **random** hash function $h: U \rightarrow \{1, \dots, n\}$.



3. HASH IT OUT



Uniformly random h was good

- If we choose h uniformly at random,
for all $u_i, u_j \in U$ with $u_i \neq u_j$,
$$P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$
- That was enough to ensure that, in expectation,
a bucket isn't too full.

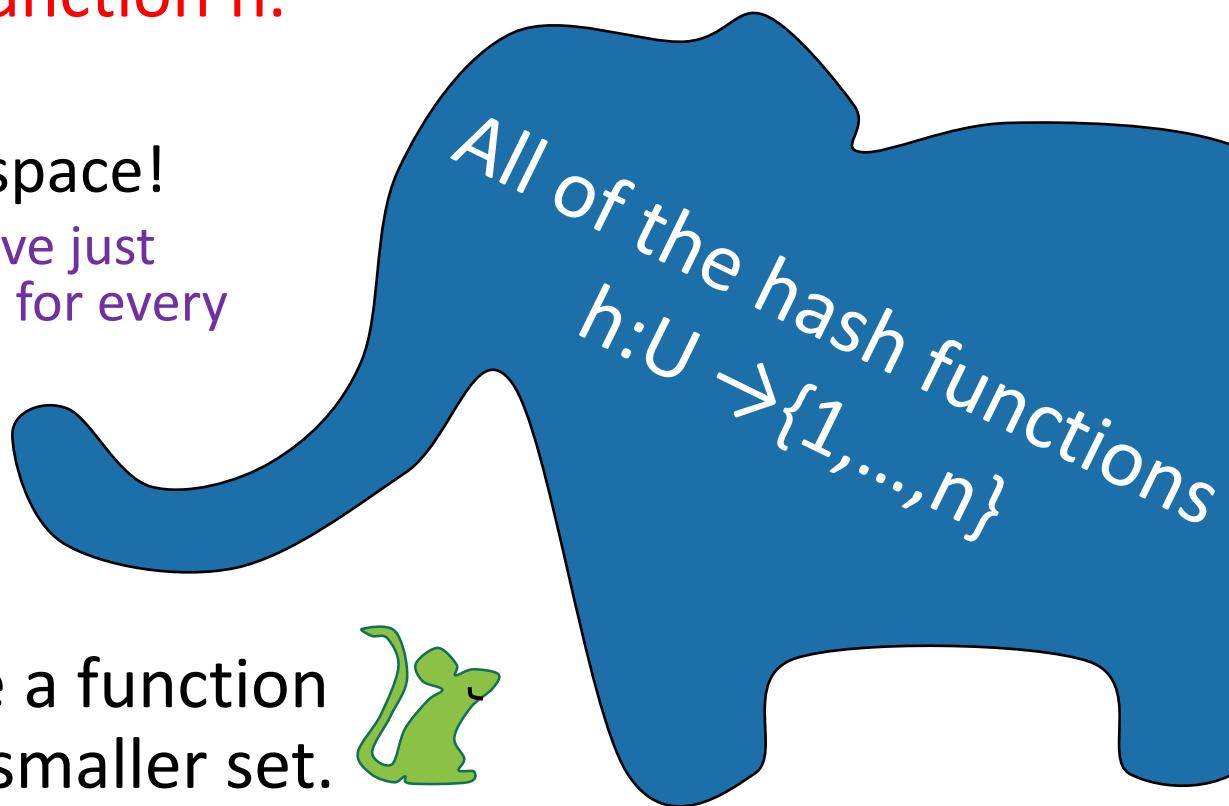
*aka, collision
probability is
small*

A bit more formally:

For any sequence of INSERT/DELETE/SEARCH operations
on any n elements of U , the expected runtime (over the
random choice of h) is $O(1)$ per operation.

Uniformly random h was bad

- If we actually want to implement this, **we have to store the hash function h .**
- That takes a lot of space!
 - We may as well have just initialized a bucket for every single item in U .
- Instead, we chose a function randomly from a smaller set.



We needed a **smaller** set that still has this property

- If we choose h uniformly at random,

for all $u_i, u_j \in U$ with $u_i \neq u_j$,

$$P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

This was all we needed to make sure that the buckets were balanced in expectation!

- We call any set with that property a **universal hash family**.
- We gave an example of a really small one ☺



Conclusion:

- We can build a hash table that supports **INSERT/DELETE/SEARCH** in $O(1)$ expected time,
 - if we know that only n items are every going to show up, where n is waaaayyyyyy less than the size M of the universe.
- The space to implement this hash table is
$$O(n \log(M)) \text{ bits.}$$
 - $O(n)$ buckets
 - $O(n)$ items with $\log(M)$ bits per item
 - $O(\log(M))$ to store the hash fn.
- M is waaayyyyyy bigger than n , but $\log(M)$ probably isn't.

Goal

- Hash tables:
 - $O(1)$ expected time **INSERT/DELETE/SEARCH**



- Goal

Design a function $h: U \rightarrow \{1, \dots, n\}$ so that:

- No matter what input (**fewer than n items of U**)
a bad guy chooses, the buckets will be **balanced**.
- Here, **balanced** means **$O(1)$** entries per bucket.

Then, we can achieve our goal of
 $O(1)$ **INSERT/DELETE/SEARCH**

Ideas



- Direct addressing
 - Pros: $O(1)$ INSERT/DELETE/SEARCH
 - Cons: large space required
- Hash table with chaining
 - Pros: small space than direct addressing
 - Issues: insert in $O(1)$, search in $O(\text{length}(\text{list}))$.
 - We want the items to be pretty spread-out in the buckets.
 - How to design a hash function?
- Deterministic hash function
 - Issue: cannot achieve $O(1)$ search (proof)
- Uniformly random hash function
 - Pros: can achieve $O(1)$ search (proof)
 - Issue: huge space needed to store a random hash function

CSE301 Introduction to Algorithms

Tree Algorithms

Fall 2022



Instructor : Hoon Sung Chwa

Course Overview

- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- **Tree Algorithms**
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

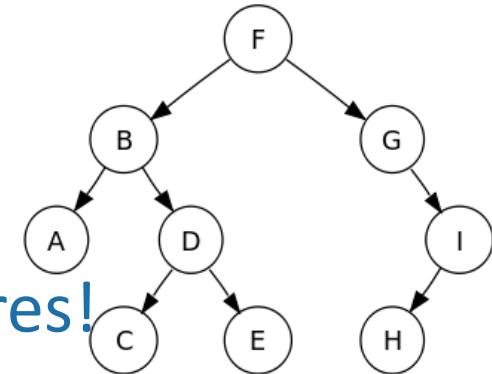
- Tree Algorithms
 - Reading: CLRS 12.1, 12.2, 12.3 and 13

Today

- Begin a brief summary into **data structures!**
- Binary search trees
 - They are better when they're balanced.

this will lead us to...

- Self-Balancing Binary Search Trees
 - **Red-Black** trees.



Why are we studying self-balancing BSTs?

1. The punchline is **important**:
 - A data structure with $O(\log(n))$ INSERT/DELETE/SEARCH
2. The idea behind **Red-Black Trees** is clever
 - It's good to be exposed to clever ideas.

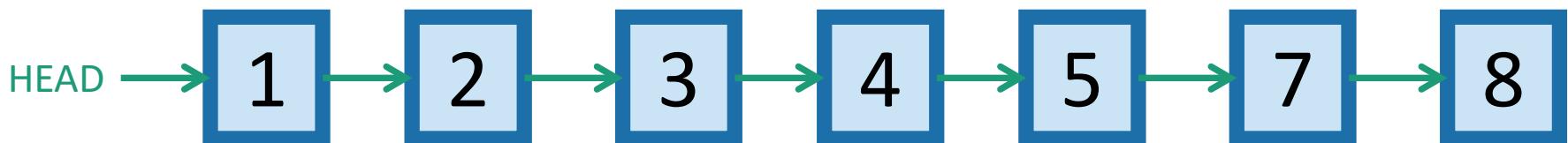
priority queue

Some data structures for storing objects like 5 (aka, nodes with keys)

- (Sorted) arrays:



- (Sorted) linked lists:



- Some basic operations:
 - INSERT, DELETE, SEARCH

Sorted Arrays

1	2	3	4	5	7	8
---	---	---	---	---	---	---

- $O(n)$ INSERT/DELETE:

1	2	3	4	4.5	7	8
---	---	---	---	-----	---	---

- $O(\log(n))$ SEARCH:

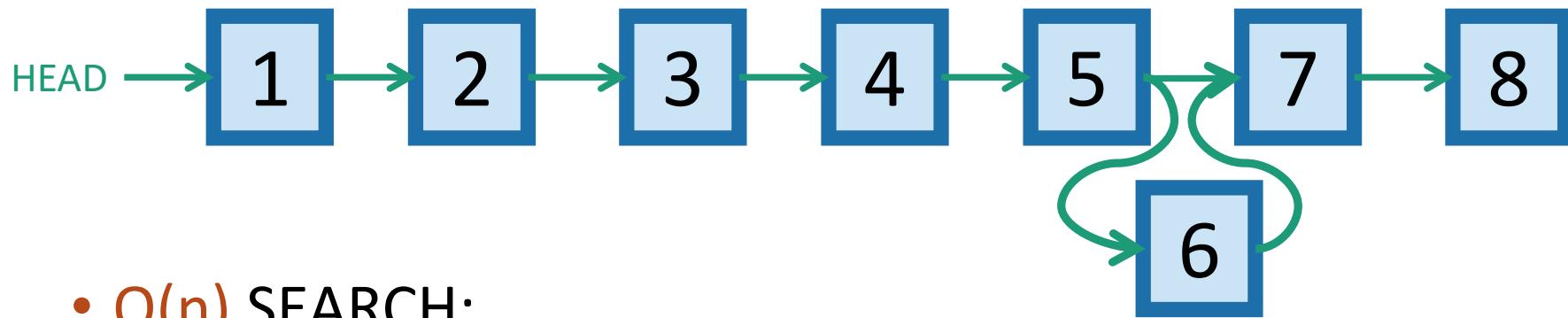
1	2	3	4	5	7	8
---	---	---	---	---	---	---

eg, Binary search to see if 3 is in A.

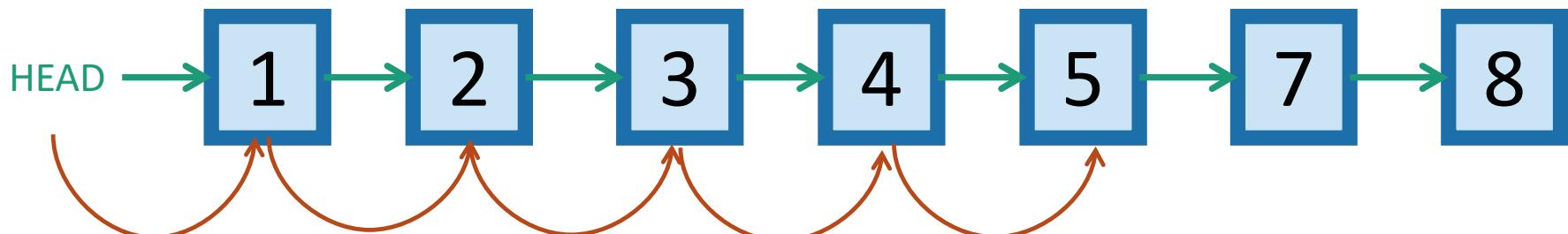
Sorted linked lists



- $O(1)$ INSERT/DELETE:
 - (assuming we have a pointer to the location of the insert/delete)



- $O(n)$ SEARCH:



Motivation for Binary Search Trees

TODAY!

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$

Binary tree terminology

Each node has two **children**

2 is a descendant of **5**

The **left child** of **3** is **2**

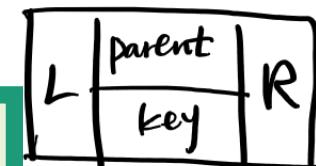
The **right child** of **3** is **4**

Both **children** of **1** are **NIL**

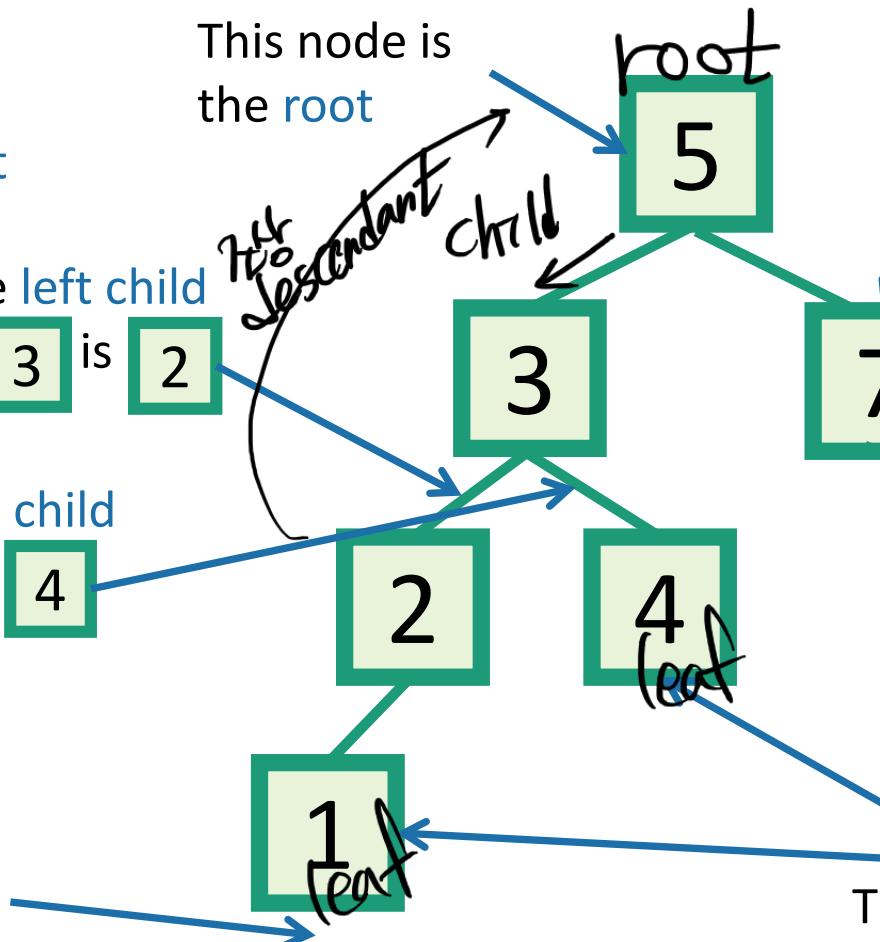
This node is the **root**

This is a **node**.
It has a **key** (7).

Each node has a pointer to its left child, right child, and parent.

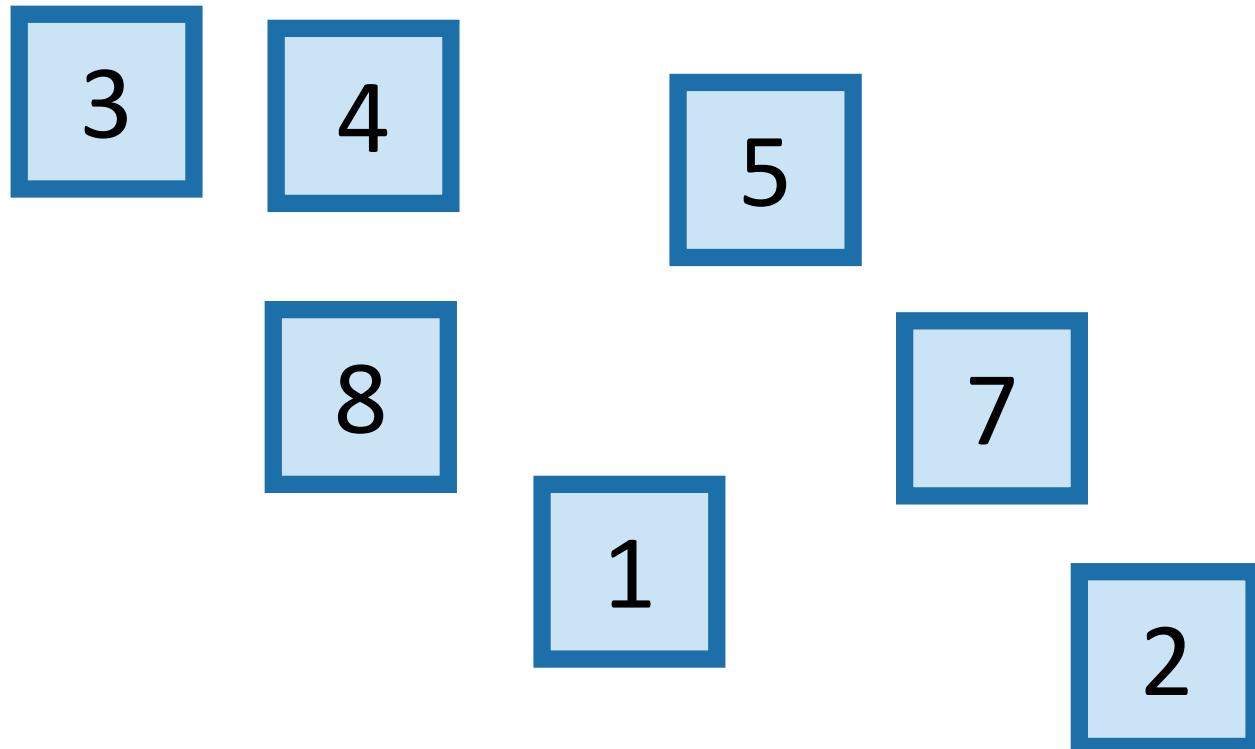


These nodes are **leaves**.



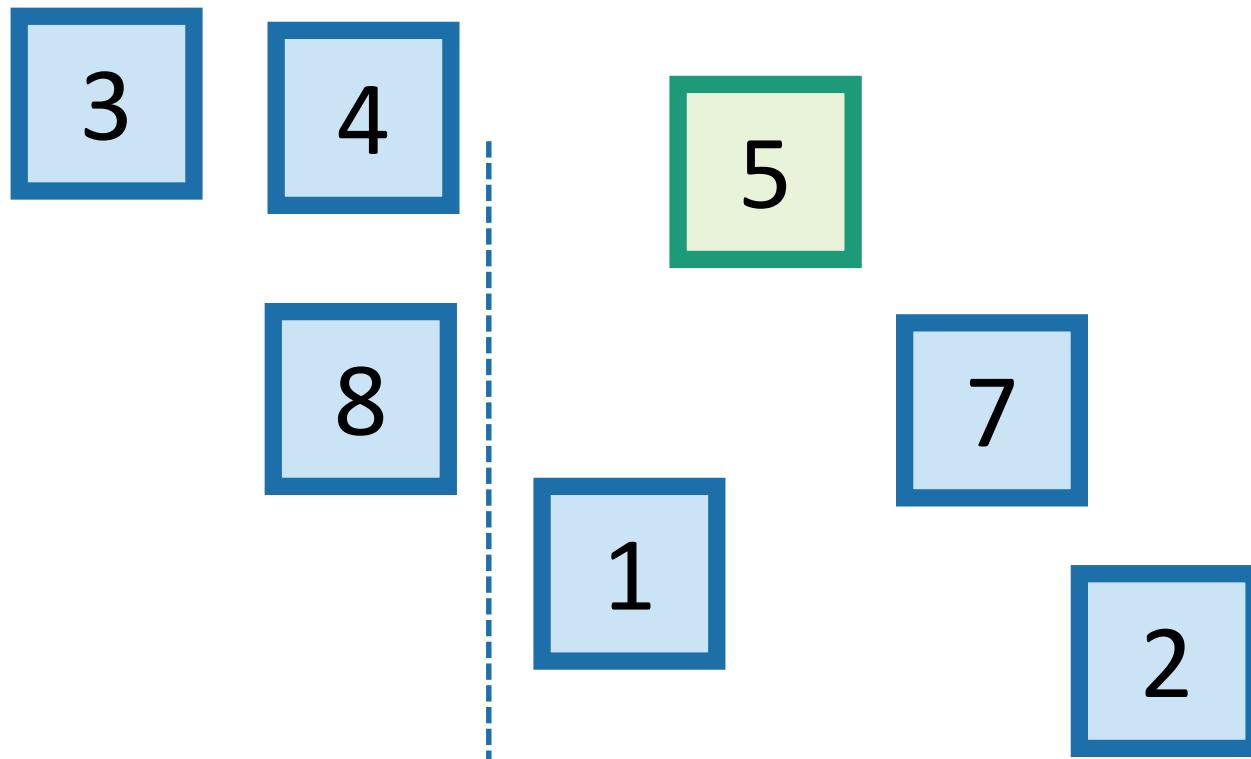
Binary Search Trees

- It's a **binary tree** so that:
 - Every LEFT descendant of a node has key less than that node.
 - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



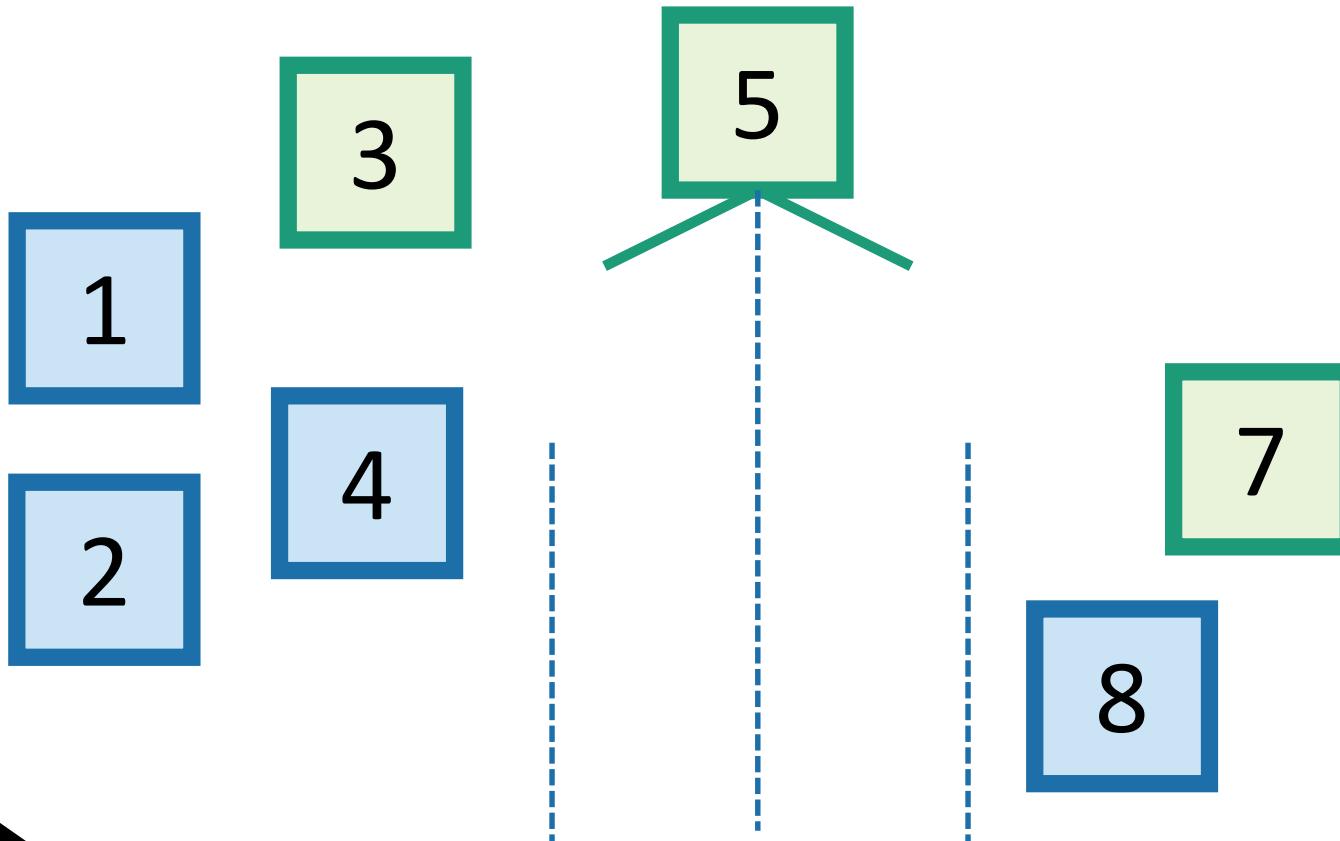
Binary Search Trees

- It's a **binary tree** so that:
 - Every LEFT descendant of a node has key less than that node.
 - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



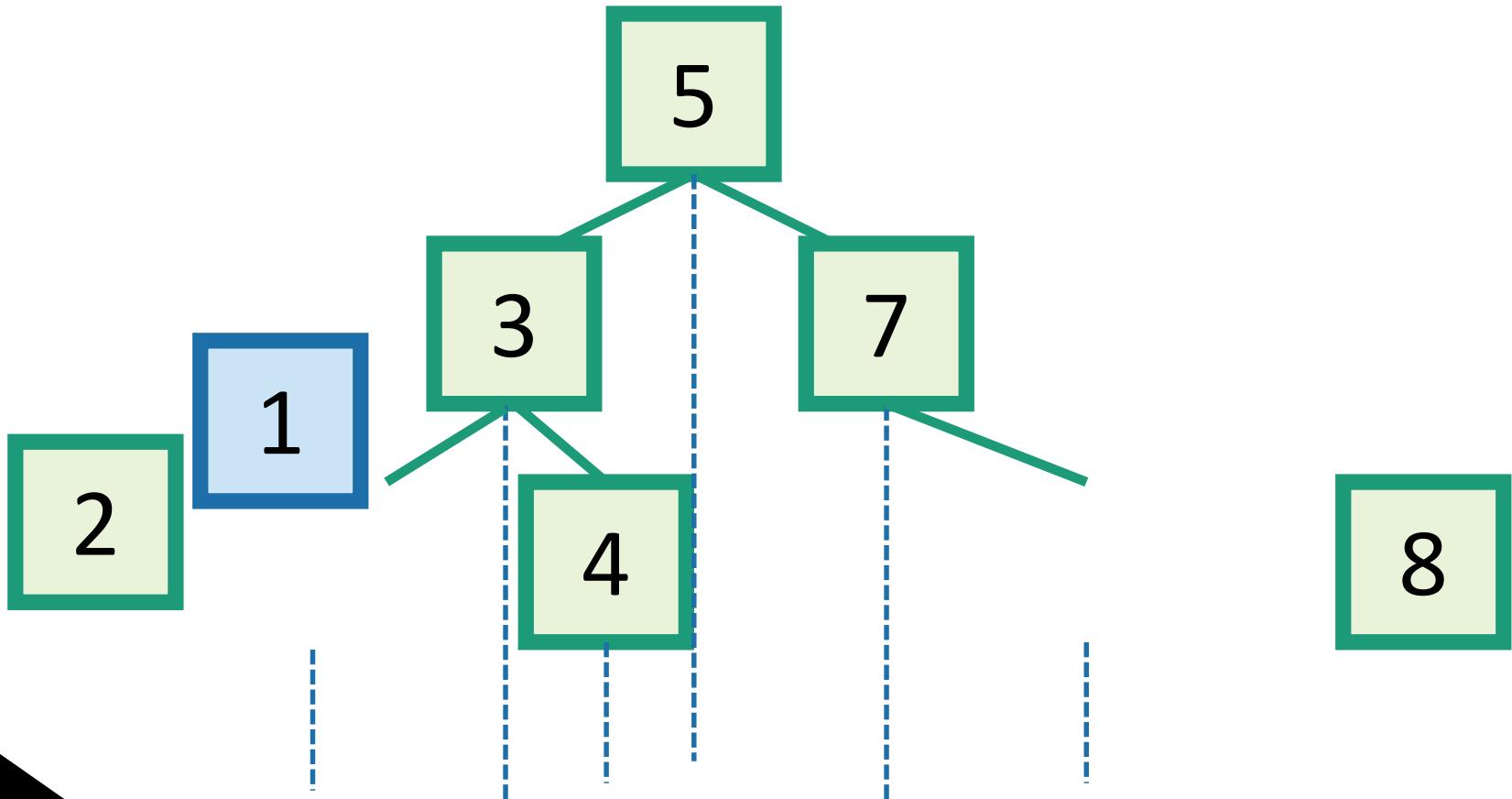
Binary Search Trees

- It's a **binary tree** so that:
 - Every LEFT descendant of a node has key less than that node.
 - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



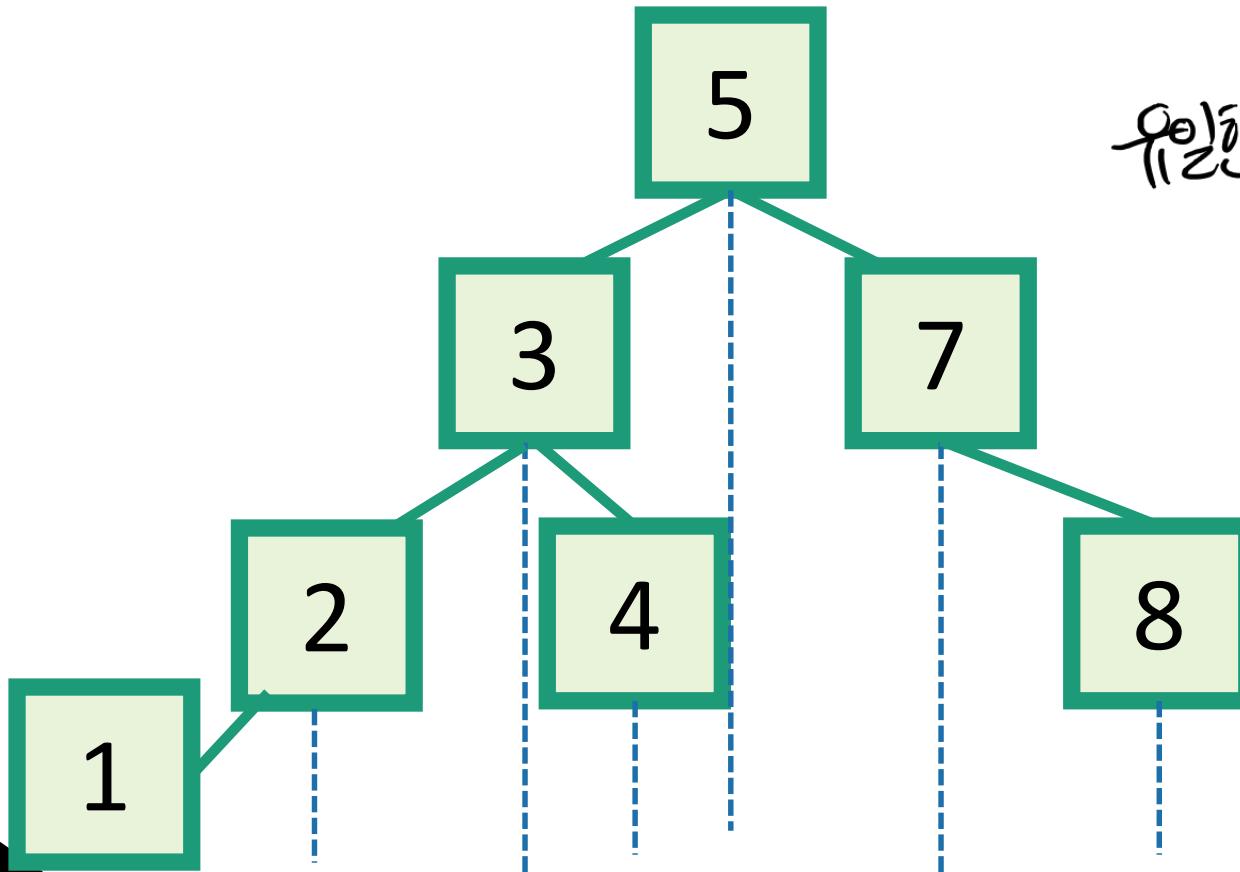
Binary Search Trees

- It's a **binary tree** so that:
 - Every LEFT descendant of a node has key less than that node.
 - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



Binary Search Trees

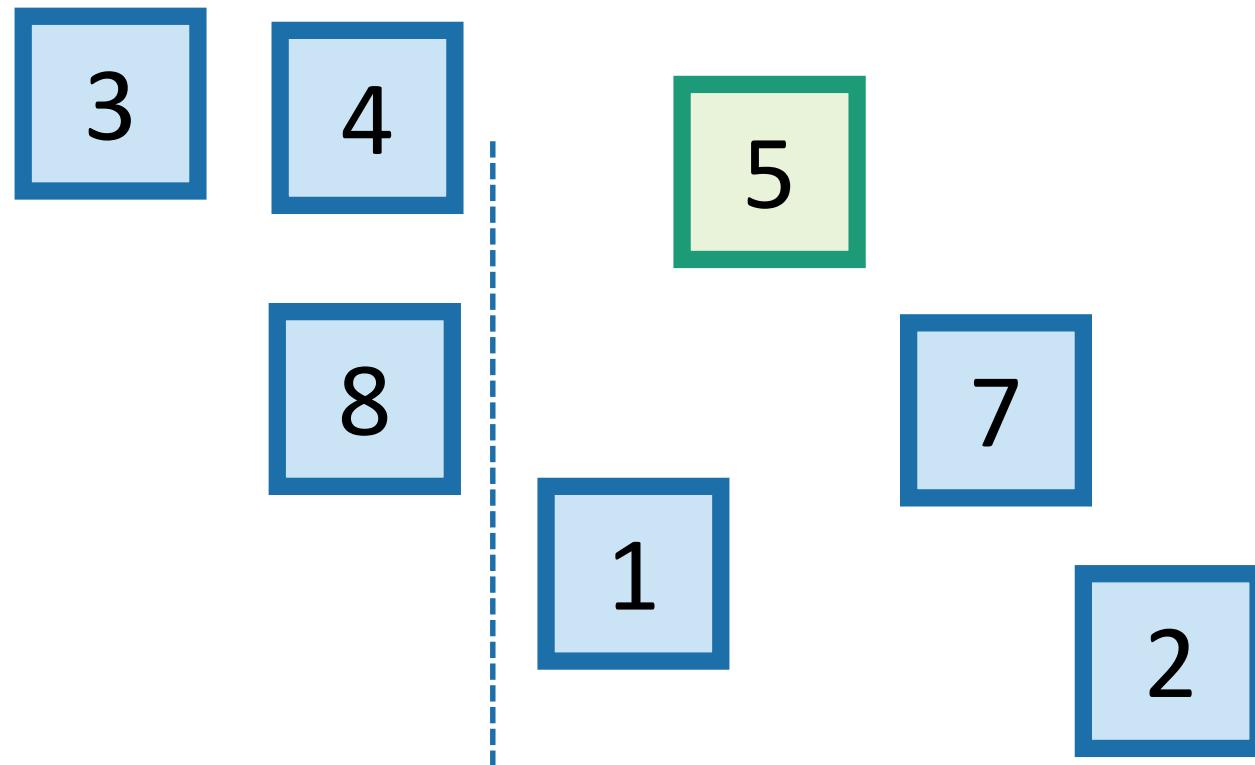
- It's a **binary tree** so that:
 - Every LEFT descendant of a node has key less than that node.
 - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



Q: Is this the only
이유)한가? binary search tree I
could possibly build
with these values?

No!
A: No. I made
choices about
which nodes to
choose when. Any
choices would
have been fine.

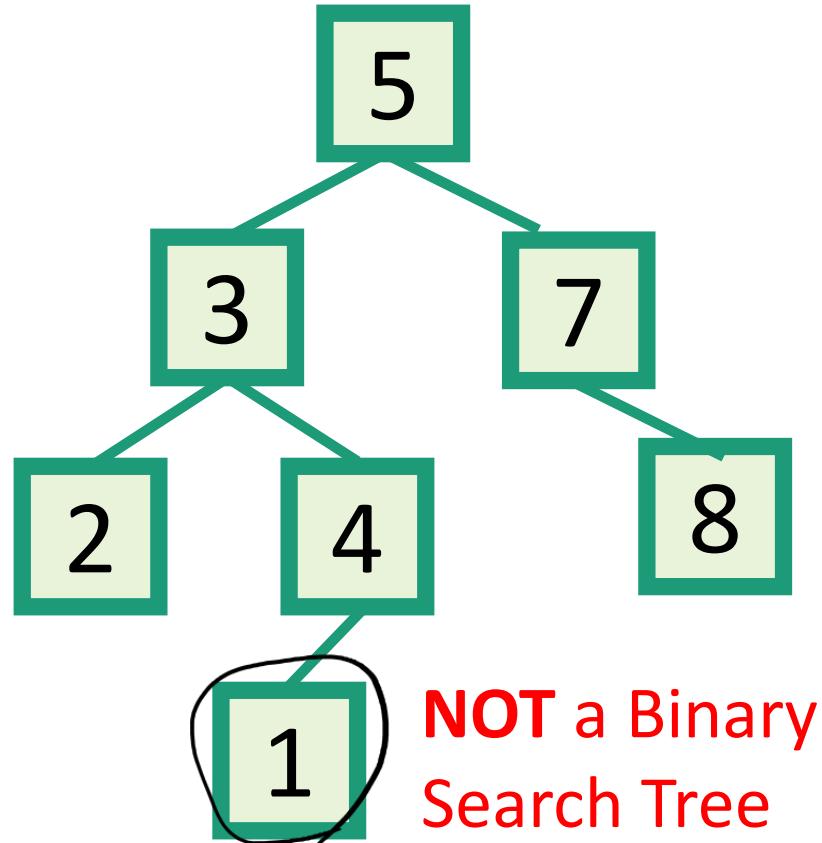
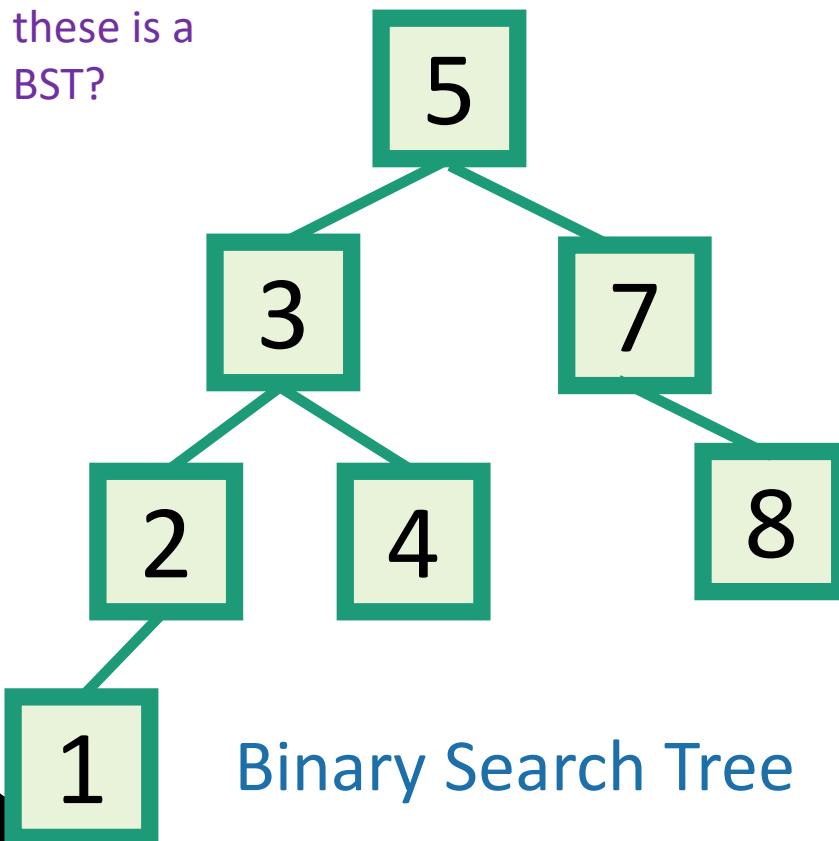
Aside: this should look familiar
kinda like QuickSort



Binary Search Trees

- It's a **binary tree** so that:
 - Every **LEFT** descendant of a node has key less than that node.
 - Every **RIGHT** descendant of a node has key larger than that node.

Which of
these is a
BST?



Remember the goal

Fast **SEARCH**/**INSERT**/**DELETE**

Can we do these?

Recall:

Why are we studying self-balancing BSTs?

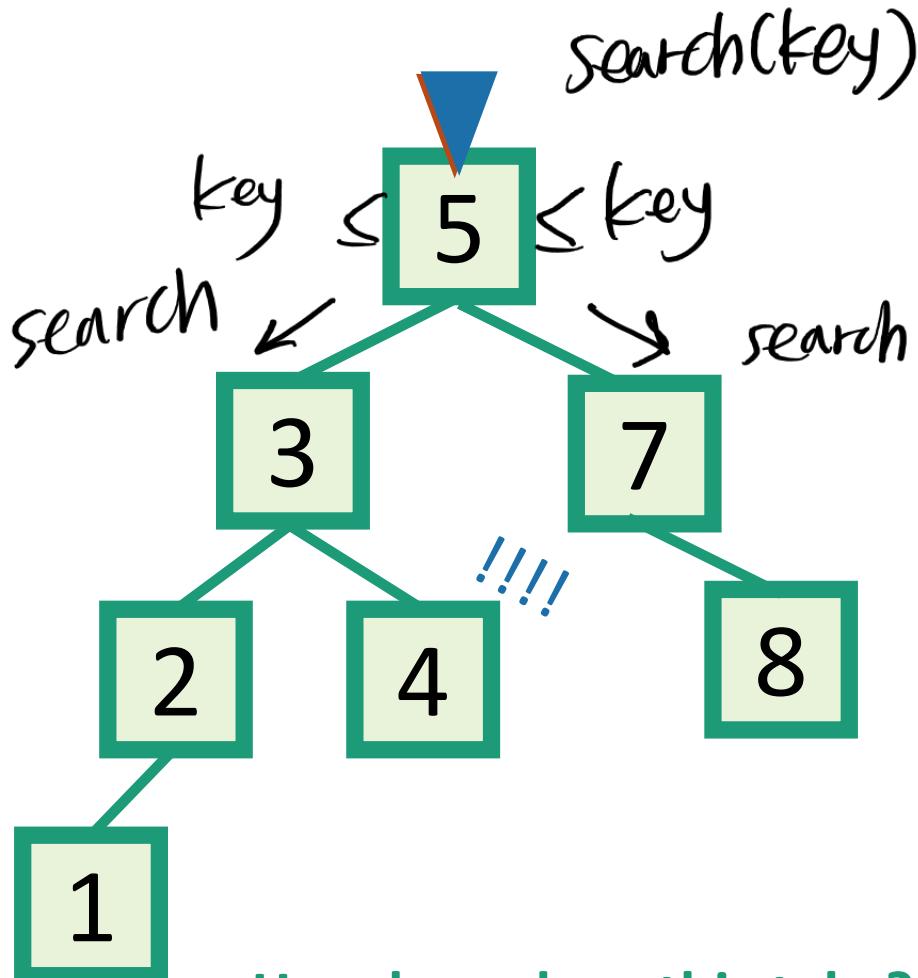
1. The punchline is **important**:
 - A data structure with $O(\log(n))$ INSERT/DELETE/SEARCH
2. The idea behind **Red-Black Trees** is clever
 - It's good to be exposed to clever ideas.

Motivation for Binary Search Trees

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$

SEARCH in a Binary Search Tree

definition by example



How long does this take?

$O(\text{length of longest path}) = O(\text{height})$

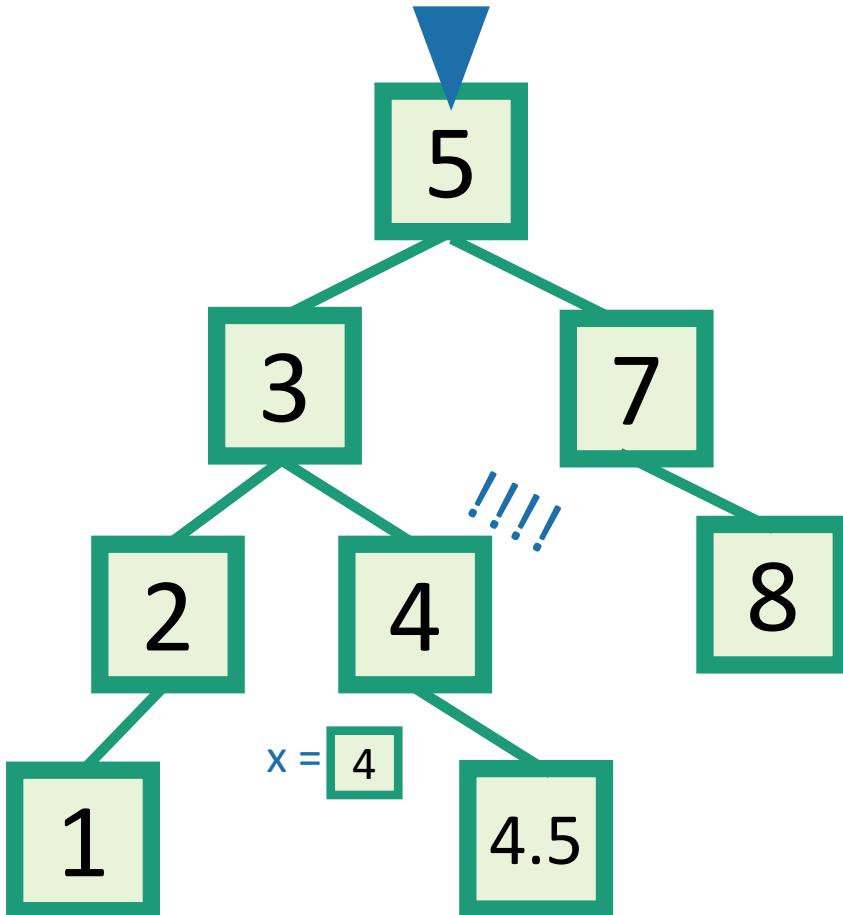
EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

- It turns out it will be convenient to **return 4** in this case
- (that is, **return** the last node before we went off the tree)

Write pseudocode
(or actual code) to
implement this!

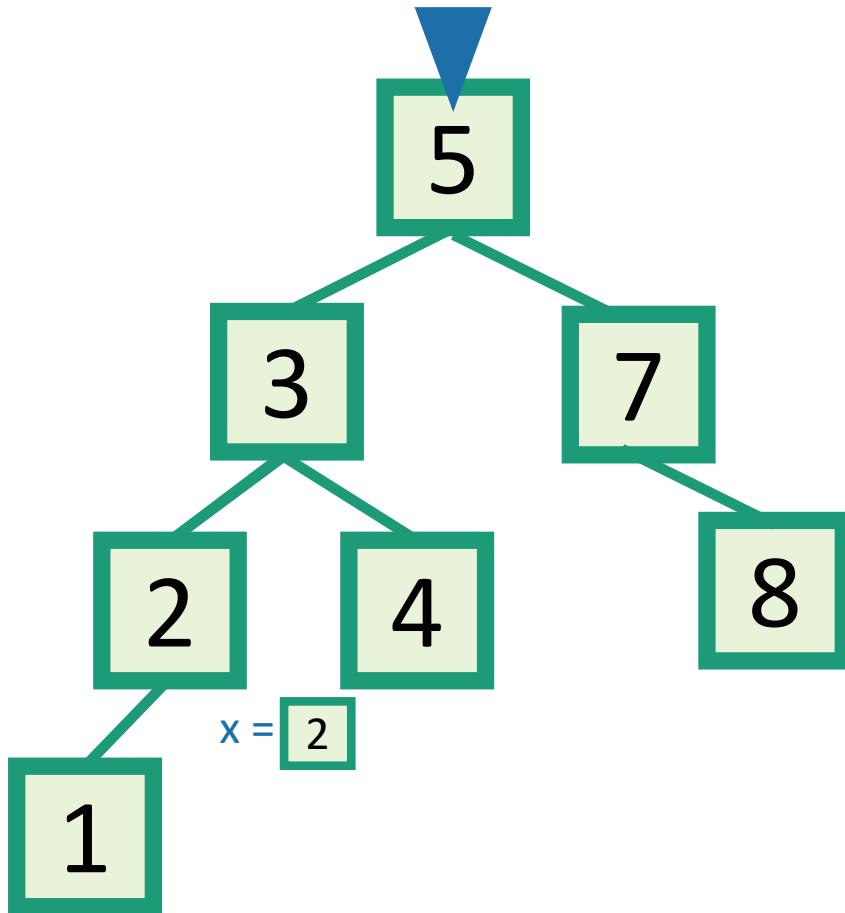
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- **INSERT(key):** $\log n$
 - $x = \text{SEARCH}(\text{key})$
 - **if key > x.key:**
 - Make a new node with the correct key, and put it as the right child of x.
 - **if key < x.key:**
 - Make a new node with the correct key, and put it as the left child of x.
 - **if x.key == key:**
 - **return**

DELETE in a Binary Search Tree



EXAMPLE: Delete 2

- **DELETE(key):**
 - $x = \text{SEARCH}(\text{key})$
 - if $x.\text{key} == \text{key}$:
 -delete x

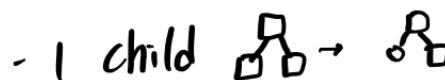


This is a bit more complicated...

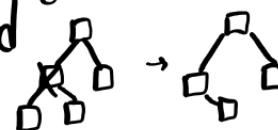
3 cases

- no child

- 1 child



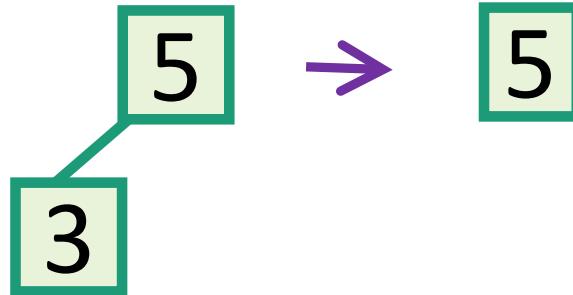
- 2 child



DELETE in a Binary Search Tree

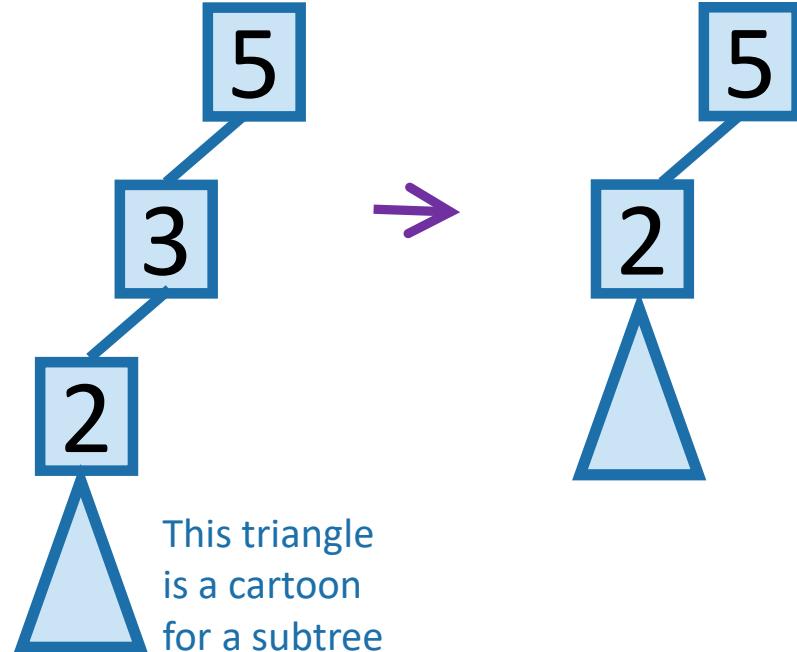
several cases (by example)

say we want to delete 3



Case 1: if 3 is a leaf,
just delete it.

Write pseudocode for all of these!

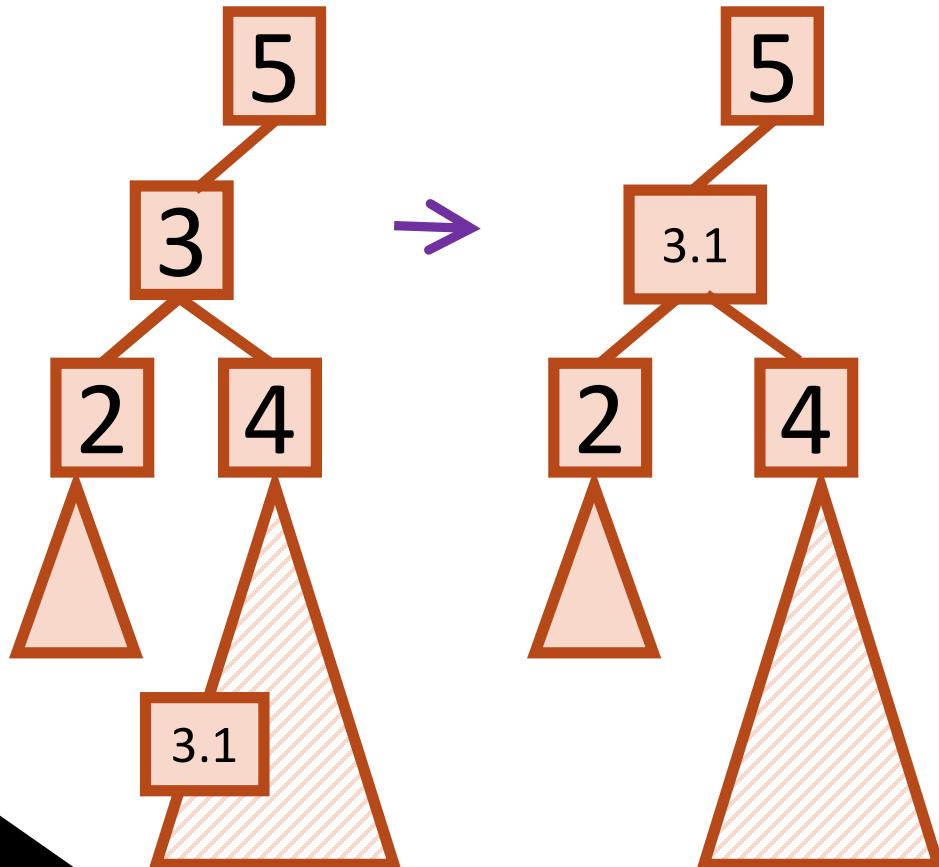


Case 2: if 3 has just one child,
move that up.

DELETE in a Binary Search Tree

ctd.

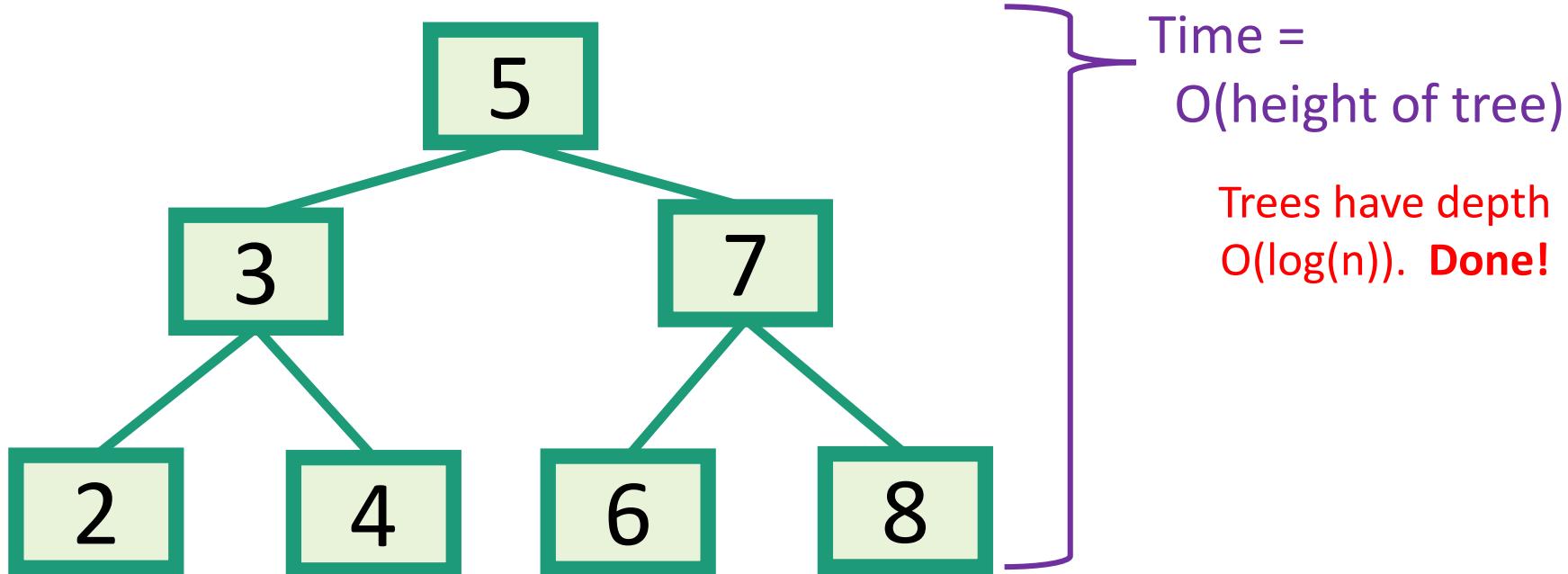
Case 3: if 3 has two children,
replace 3 with its **immediate successor**.
(aka, next biggest thing after 3)



- Does this maintain the BST property?
 - Yes.
- How do we find the immediate successor?
 - SEARCH for 3 in the subtree under 3.right
- How do we remove it when we find it?
 - If [3.1] has 0 or 1 children, do one of the previous cases.
- What if [3.1] has two children?
 - It doesn't.

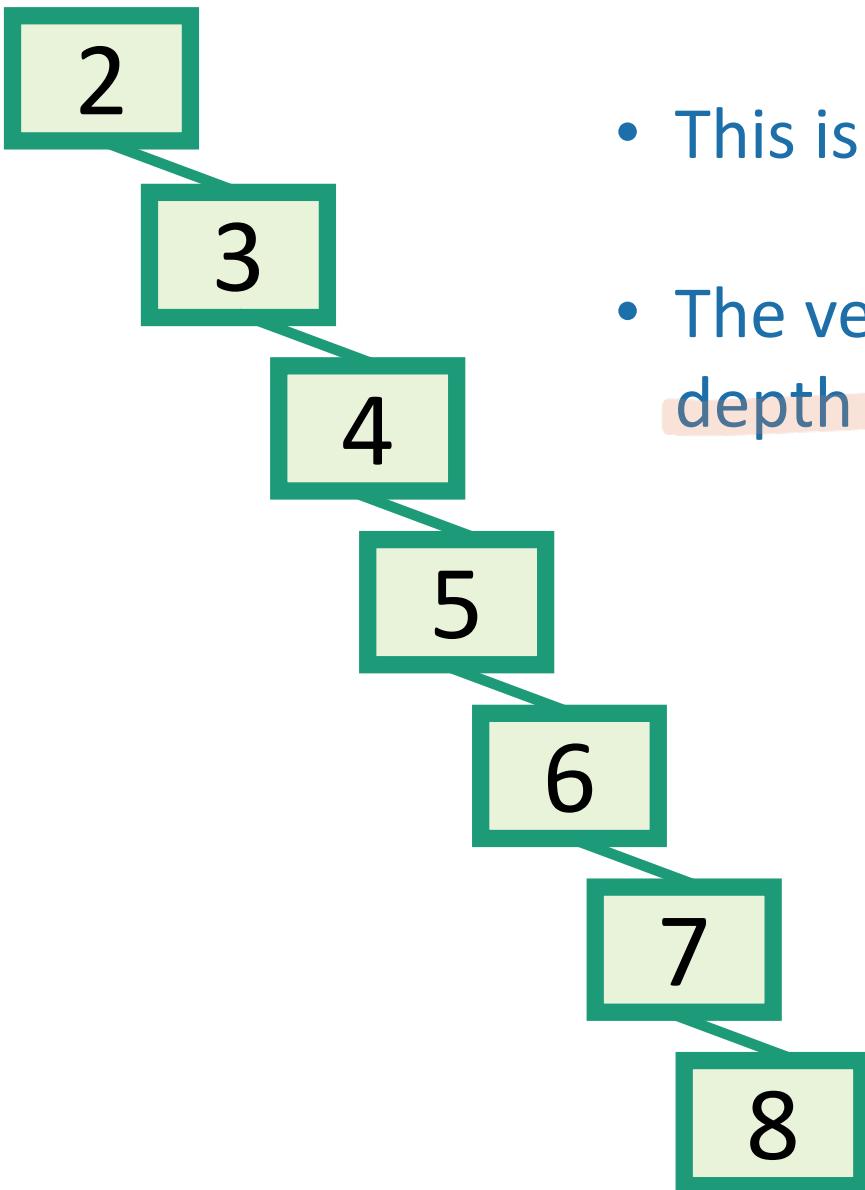
How long do these operations take?

- **SEARCH** is the big one.
 - Everything else just calls **SEARCH** and then does some small $O(1)$ -time operation.



How long does search take?

Wait...



- This is a valid binary search tree.
- The version with n nodes has depth n , **not** $O(\log(n))$.

Could such a tree show up?
In what order would I have to
insert the nodes?

Inserting in the order
2,3,4,5,6,7,8 would do it.

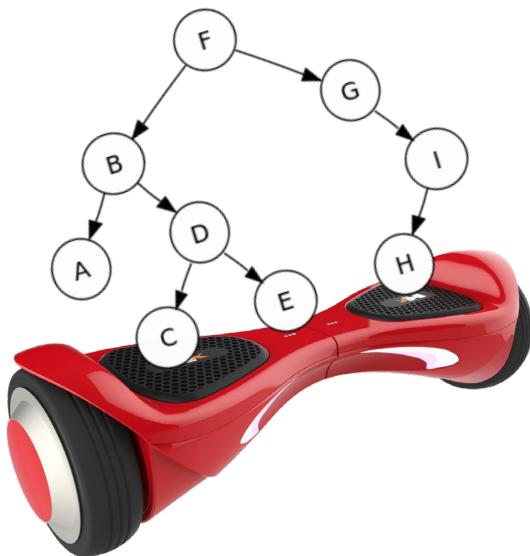
So this **could** happen.

How often is “every so often” in the worst case?
It’s actually pretty often!

What to do?

- Goal: Fast **SEARCH**/**INSERT**/**DELETE**
- All these things take time $O(\text{height})$
- And the height might be big!!! 😞
- Idea 0:
 - Keep track of how deep the tree is getting.
 - If it gets too tall, re-do everything from scratch.
 - At least $\Omega(n)$ every so often....
- Turns out that’s not a great idea. Instead we turn to...

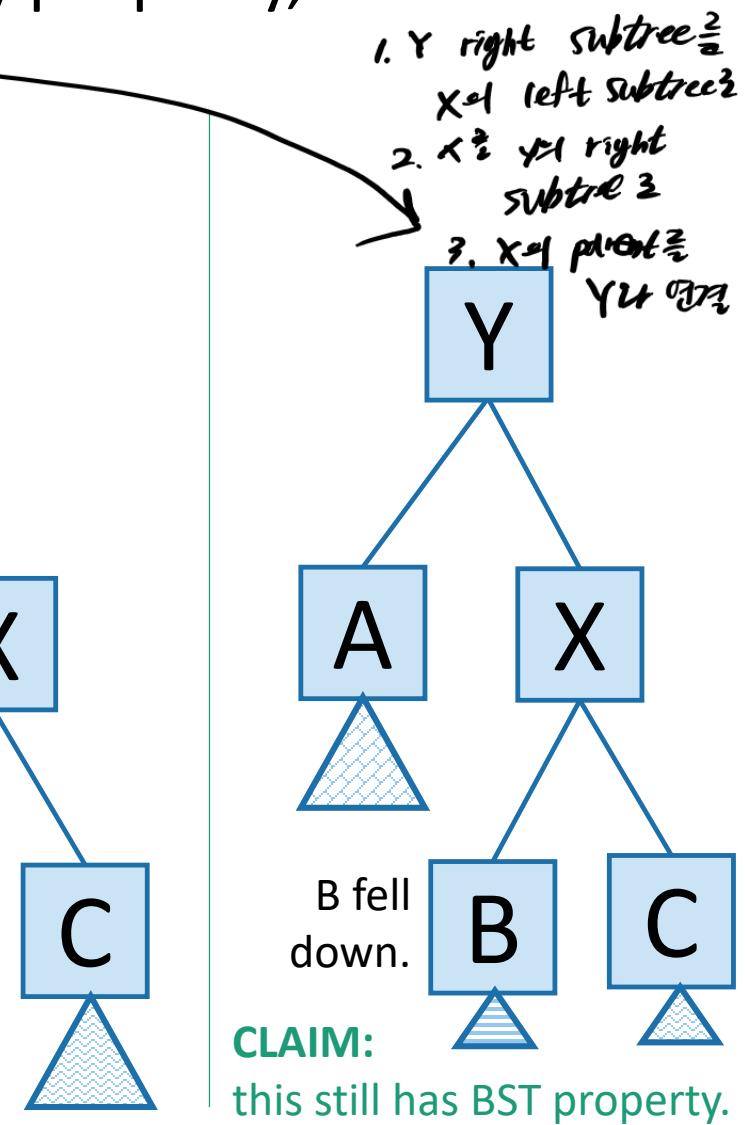
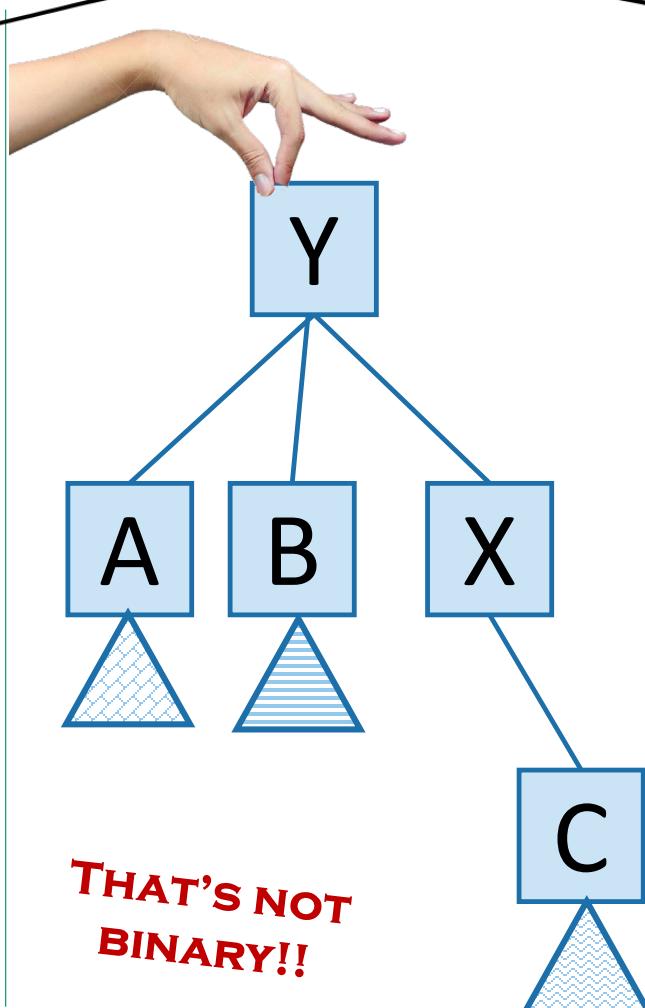
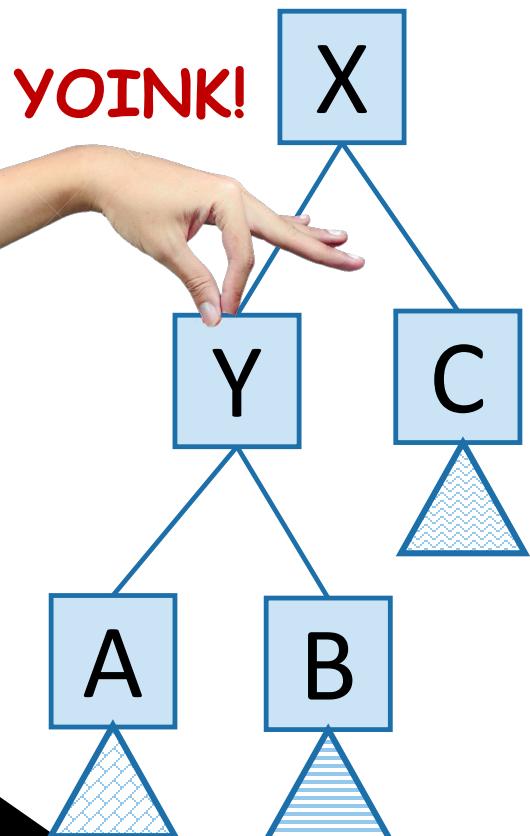
Self-Balancing Binary Search Trees



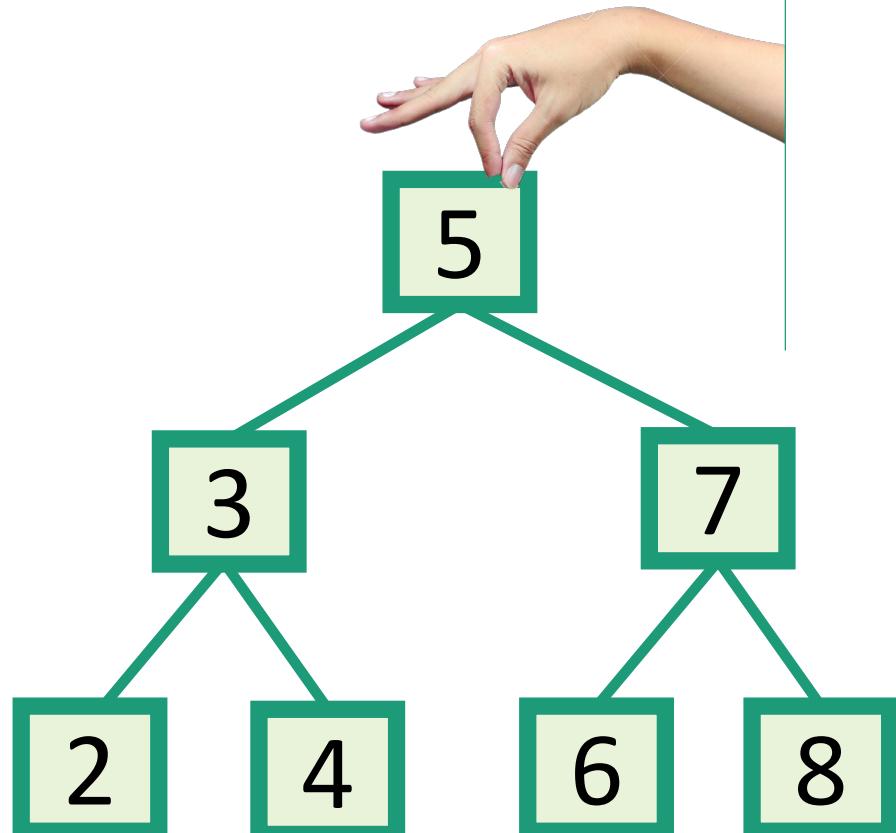
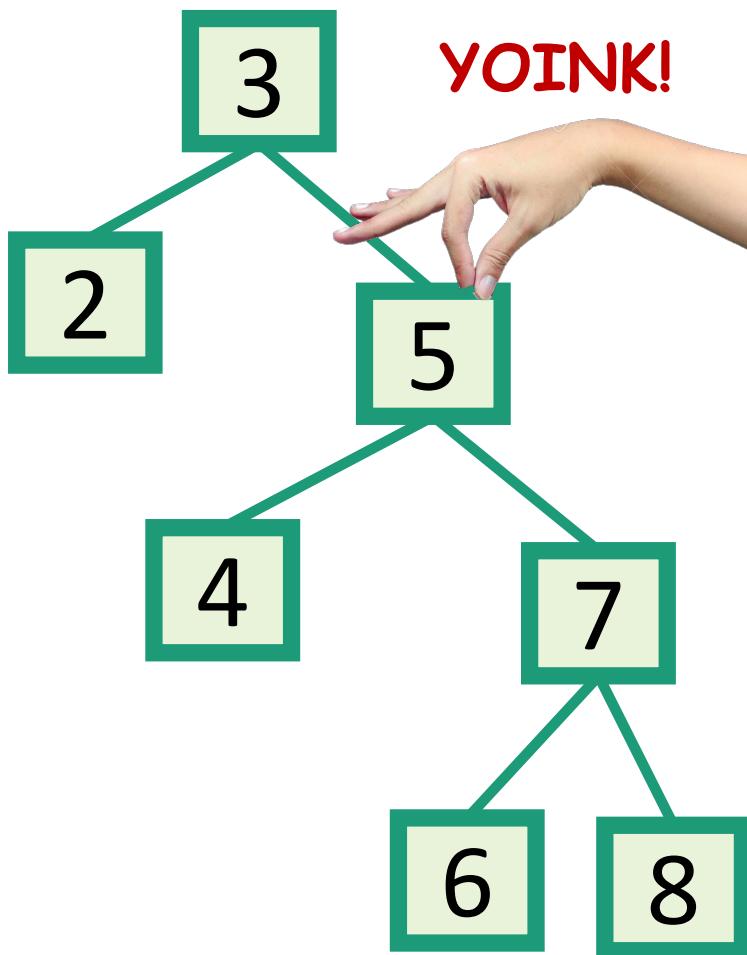
Idea 1: Rotations

No matter what lives underneath A,B,C,
this takes time O(1). (Why?)

- Maintain Binary Search Tree (BST) property, while moving stuff around.



This seems helpful



Does this work?

- Whenever something seems unbalanced, do rotations until it's okay again.



Even for me this is pretty vague.
What do we mean by “seems unbalanced”?
What’s “okay”?

Idea 2: have some proxy for balance

- Maintaining **perfect balance** is too hard.
- Instead, come up with some **proxy for balance**:
 - If the tree satisfies **[SOME PROPERTY]**, then it's pretty balanced.
 - We can maintain **[SOME PROPERTY]** using rotations.



There are actually several ways to do this, but today we'll see...

Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!

Red-Black tree!

Maintain balance by stipulating that **black nodes** are balanced, and that there aren't too many **red nodes**.

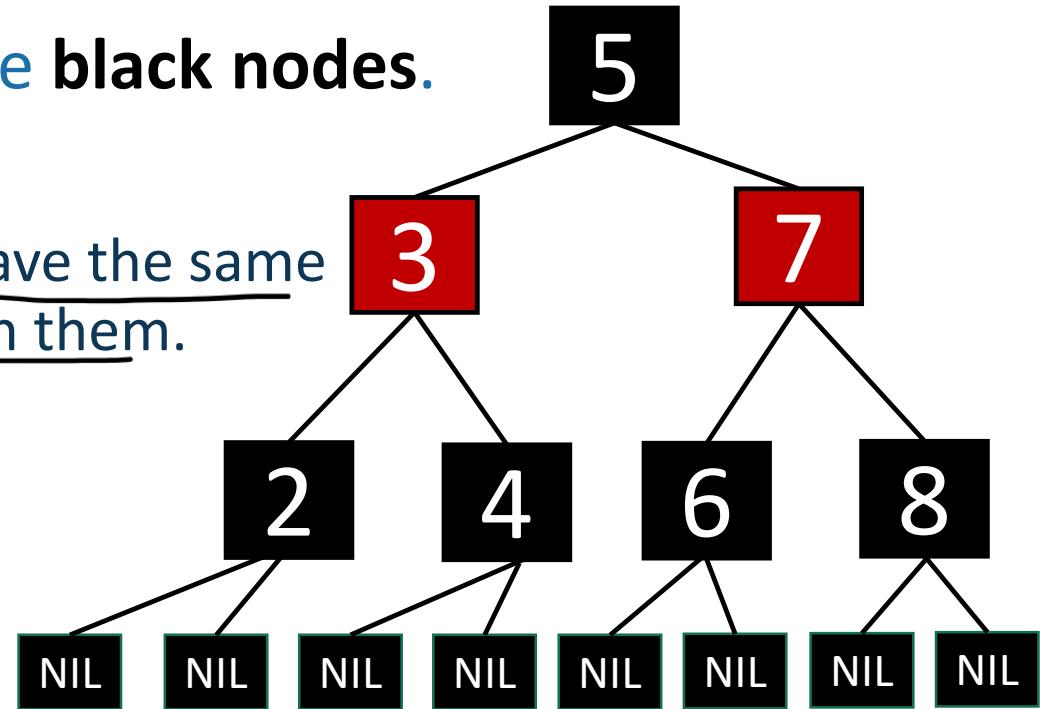
It's just good sense!



Red-Black Trees

these rules are the **proxy for balance**

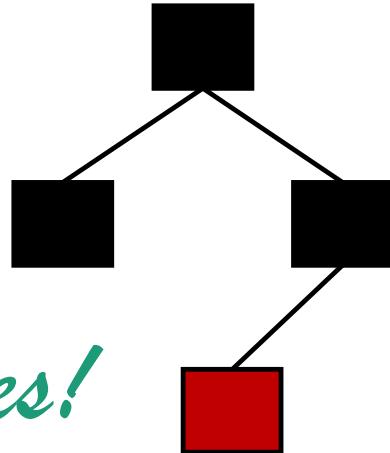
- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
 - all paths from x to NIL's have the same number of **black nodes** on them.



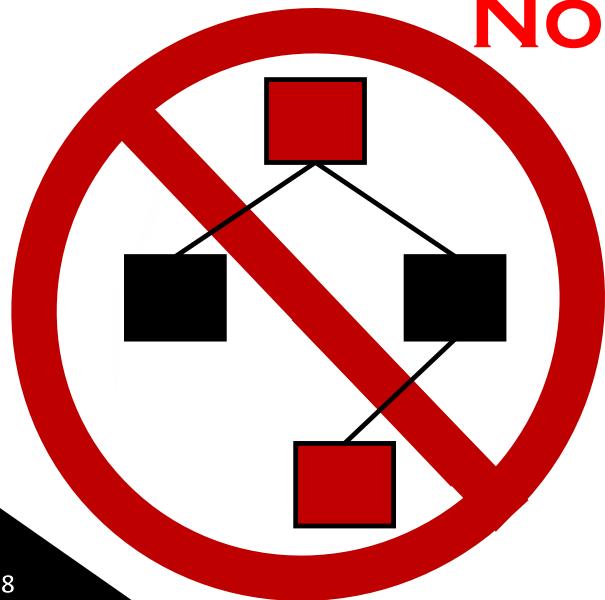
Examples(?)

- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
 - all paths from x to NIL's have the same number of **black nodes** on them.

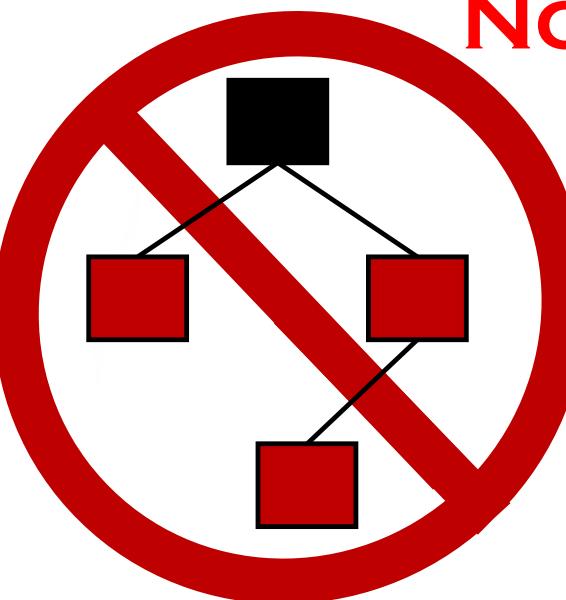
Yes!



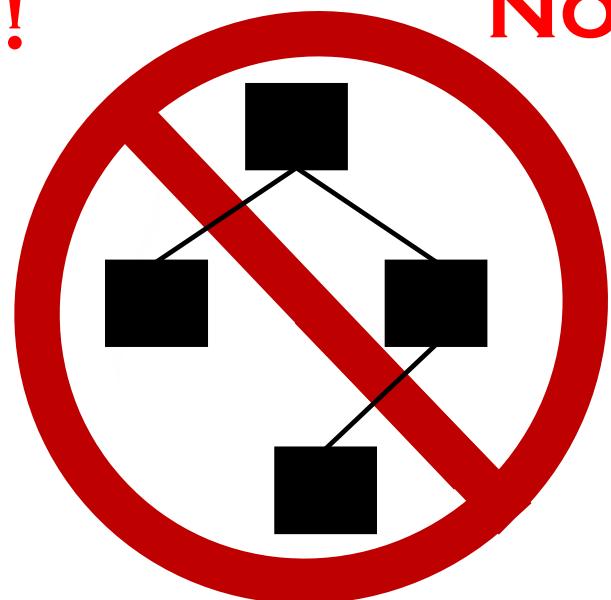
No!



No!

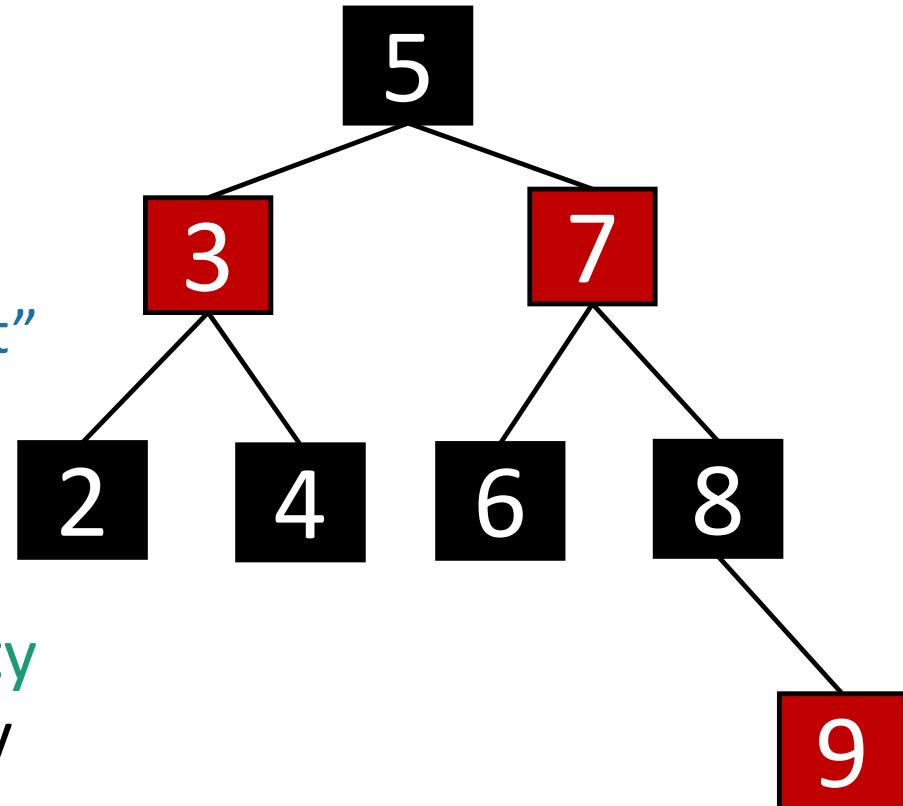


No!



Why??????

- This is **pretty balanced**.
 - The **black nodes** are balanced
 - The **red nodes** are “spread out” so they don’t mess things up too much.
- We can **Maintain this property** as we insert/delete nodes, by using **rotations**.



This is the really clever idea!

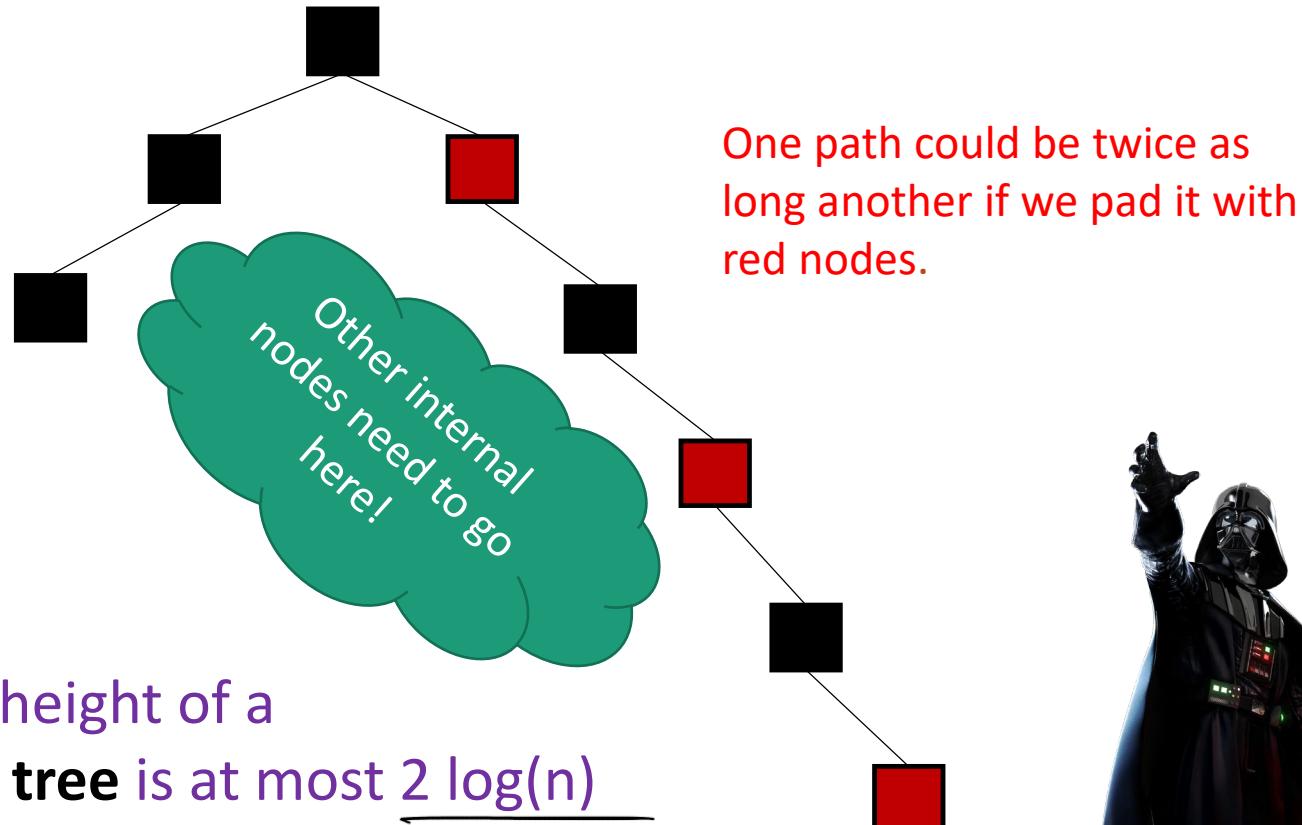
This **Red-Black** structure is a **proxy for balance**.

It's just a little weaker than perfect balance, but we can actually maintain it!



This is “pretty balanced”

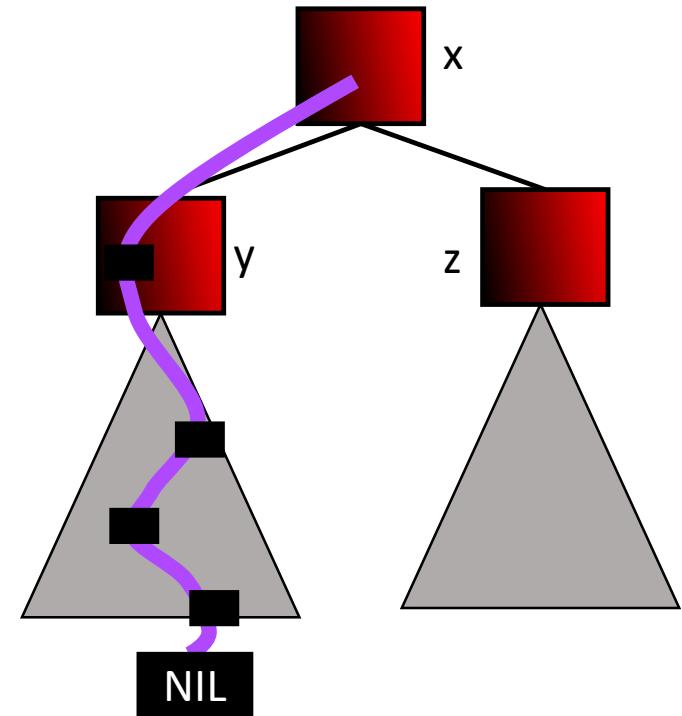
- To see why, intuitively, let’s try to build a Red-Black Tree that’s **unbalanced**.



That turns out to be basically right.

[proof sketch]

- Say there are $b(x)$ black nodes in any path from x to NIL.
 - (excluding x , including NIL).
- Claim:
 - Then there are at least $2^{b(x)} - 1$ non-NIL nodes in the subtree underneath x . (Including x).
 - [Proof by induction]



Then:

$$\begin{aligned} n &\geq 2^{b(\text{root})} - 1 && \text{using the } \underline{\text{Claim}} \\ &\geq 2^{\frac{\text{height}}{2}} - 1 && b(\text{root}) \geq \text{height}/2 \text{ because of RBTree rules.} \end{aligned}$$

Rearranging:

$$n + 1 \geq 2^{\frac{\text{height}}{2}} \Rightarrow \text{height} \leq 2\log(n + 1)$$

Base case: $b(x) = 0$. $2^0 - 1 = 0$ non-NIL descendants

Inductive step: $d(x)$: # of non-NIL descendants of x .

$$d(x) = 1 + d(x.\text{left}) + d(x.\text{right})$$

$$\geq 1 + (2^{b(x)} - 1) + (2^{b(x)} - 1)$$

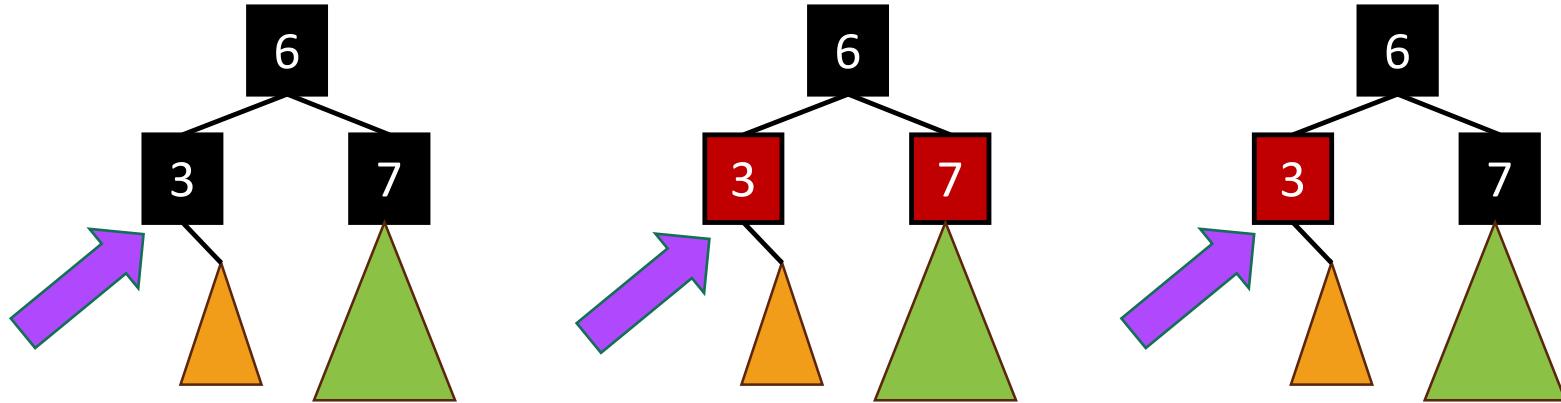
$$= 2^{b(x)} - 1$$

Okay, so it's balanced...

...but can we maintain it?

- Yes!
- For the rest of lecture:
 - sketch of how we'd do this.
- See CLRS for more details.
- (You are not responsible for the details for this class – but you should understand the main ideas).

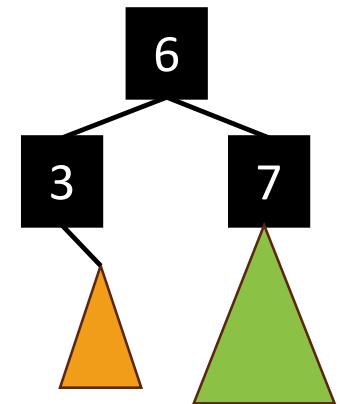
Many cases



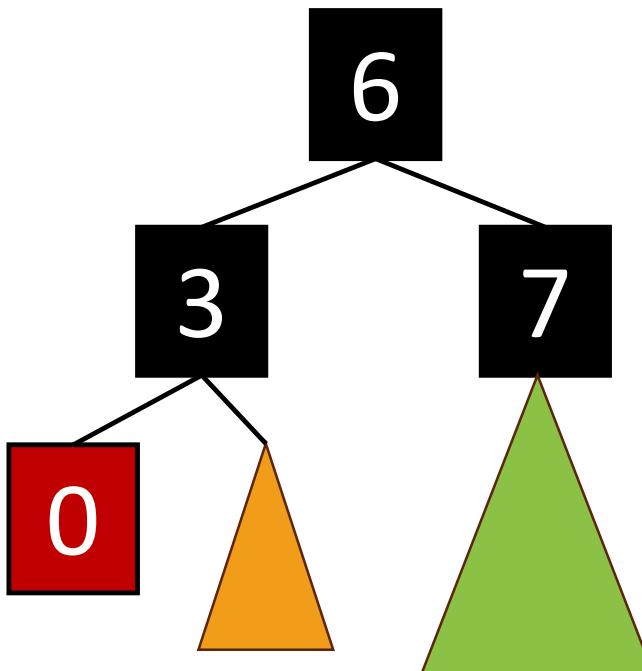
- Suppose we want to insert **here**.
 - eg, want to insert 0.

Inserting into a Red-Black Tree

- Make a new **red node**.
- Insert it as you would normally.



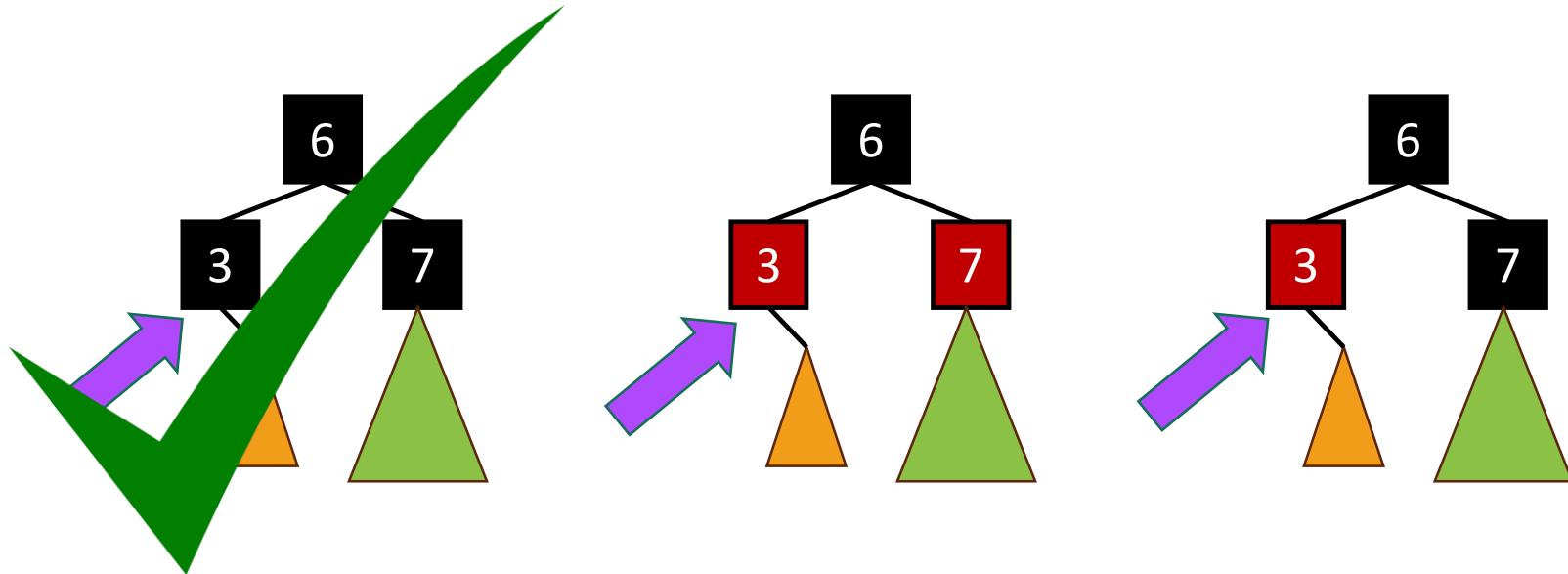
What if it looks like this?



Example: insert 0



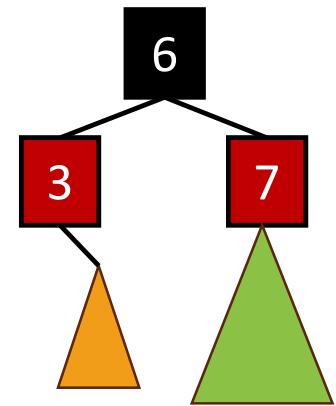
Many cases



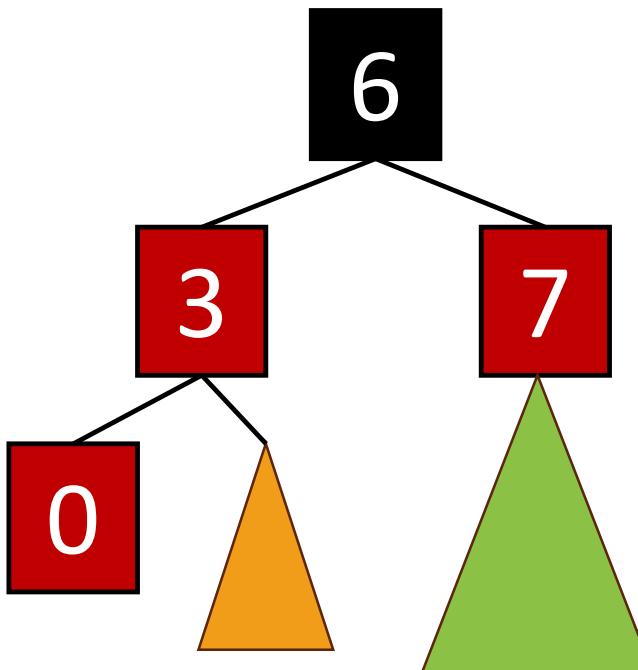
- Suppose we want to insert **here**.
 - eg, want to insert 0.

Inserting into a Red-Black Tree

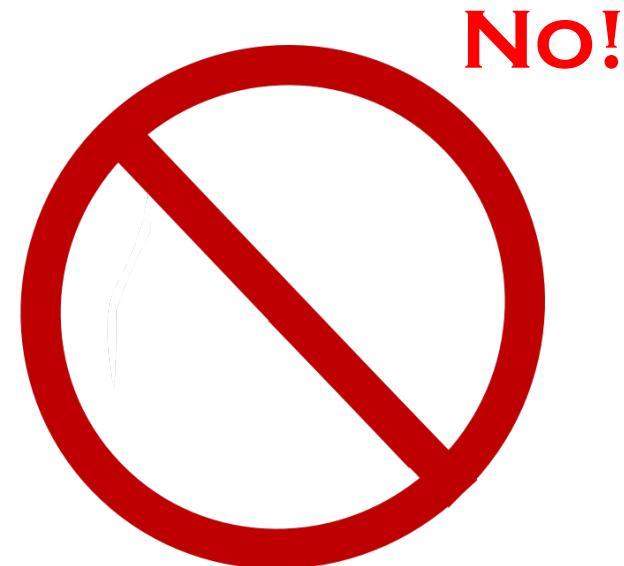
- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



What if it looks like this?

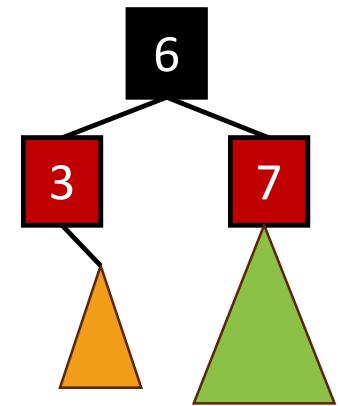


Example: insert 0

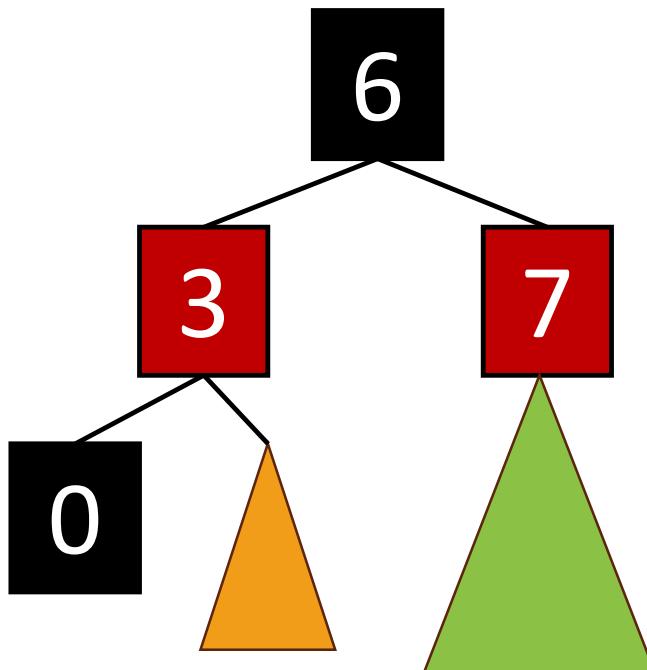


Inserting into a Red-Black Tree

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



What if it looks like this?

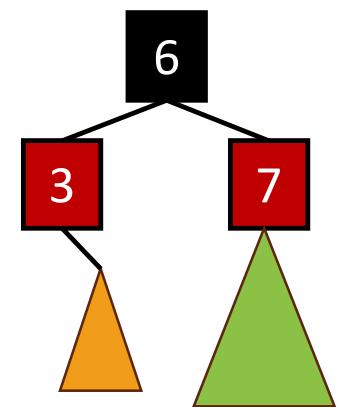


Example: insert 0

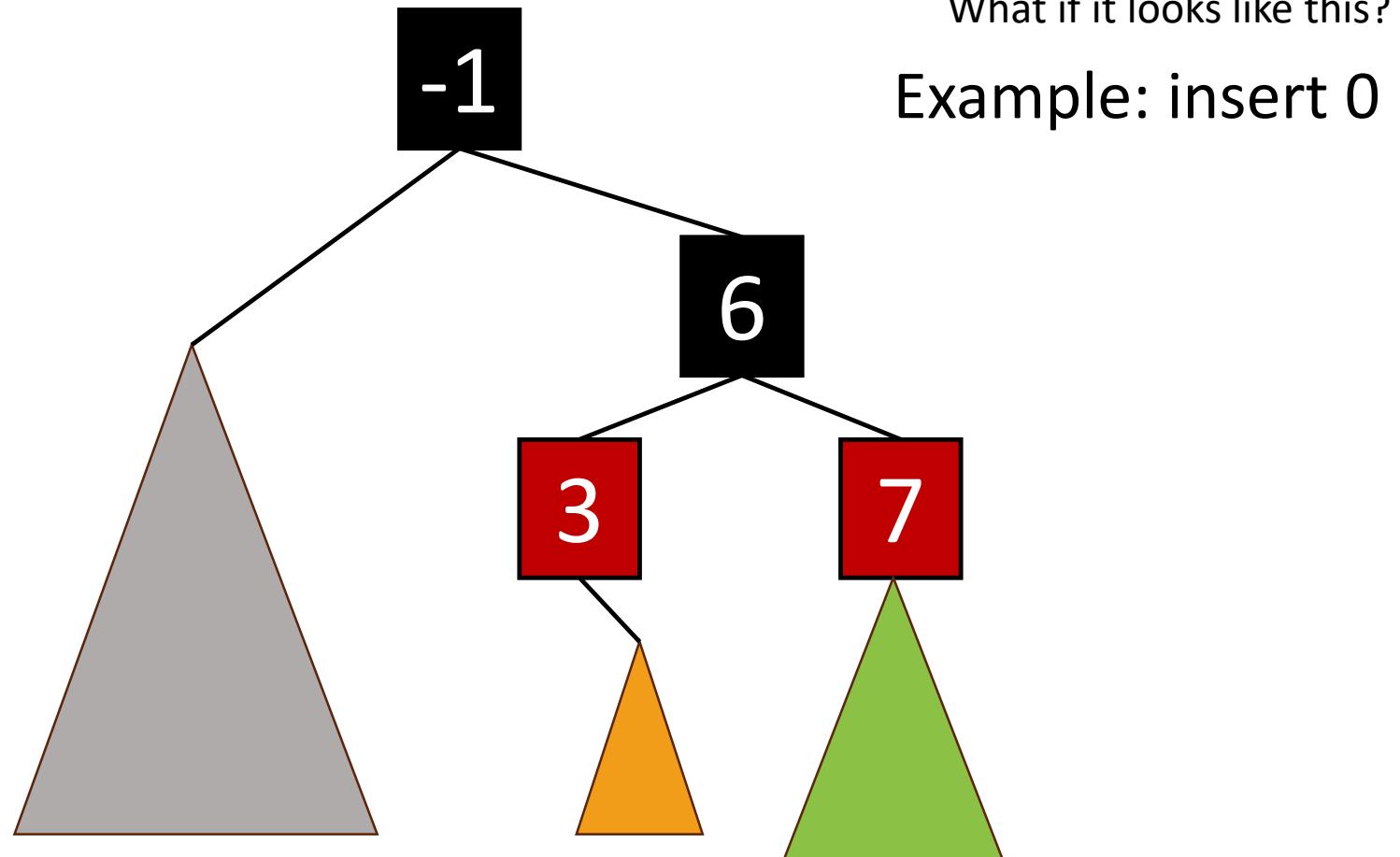
Can't we just insert 0 as a **black node**?



We need a bit more context



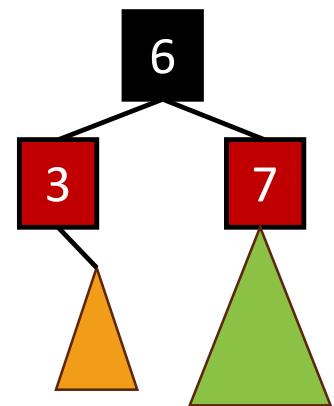
What if it looks like this?



Example: insert 0

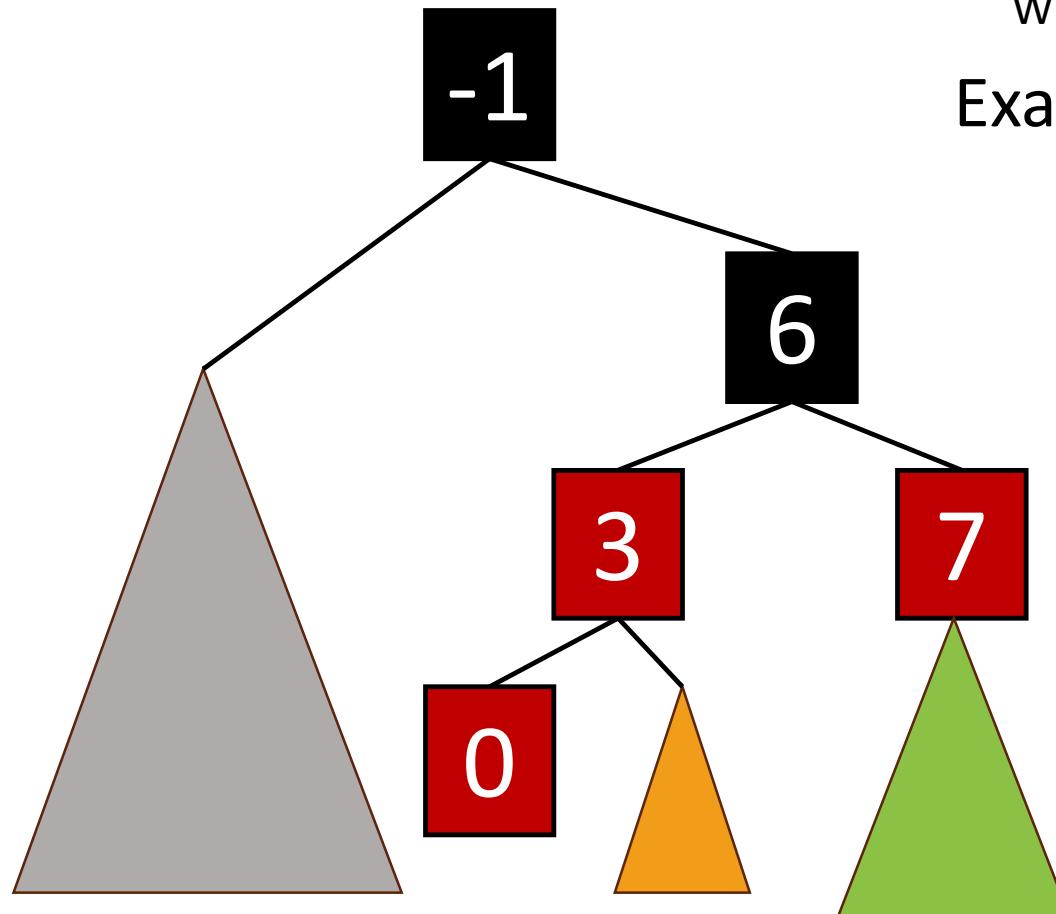
We need a bit more context

- Add 0 as a red node.



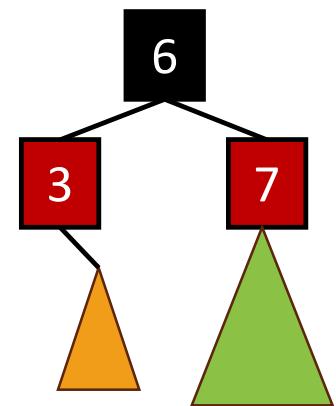
What if it looks like this?

Example: insert 0



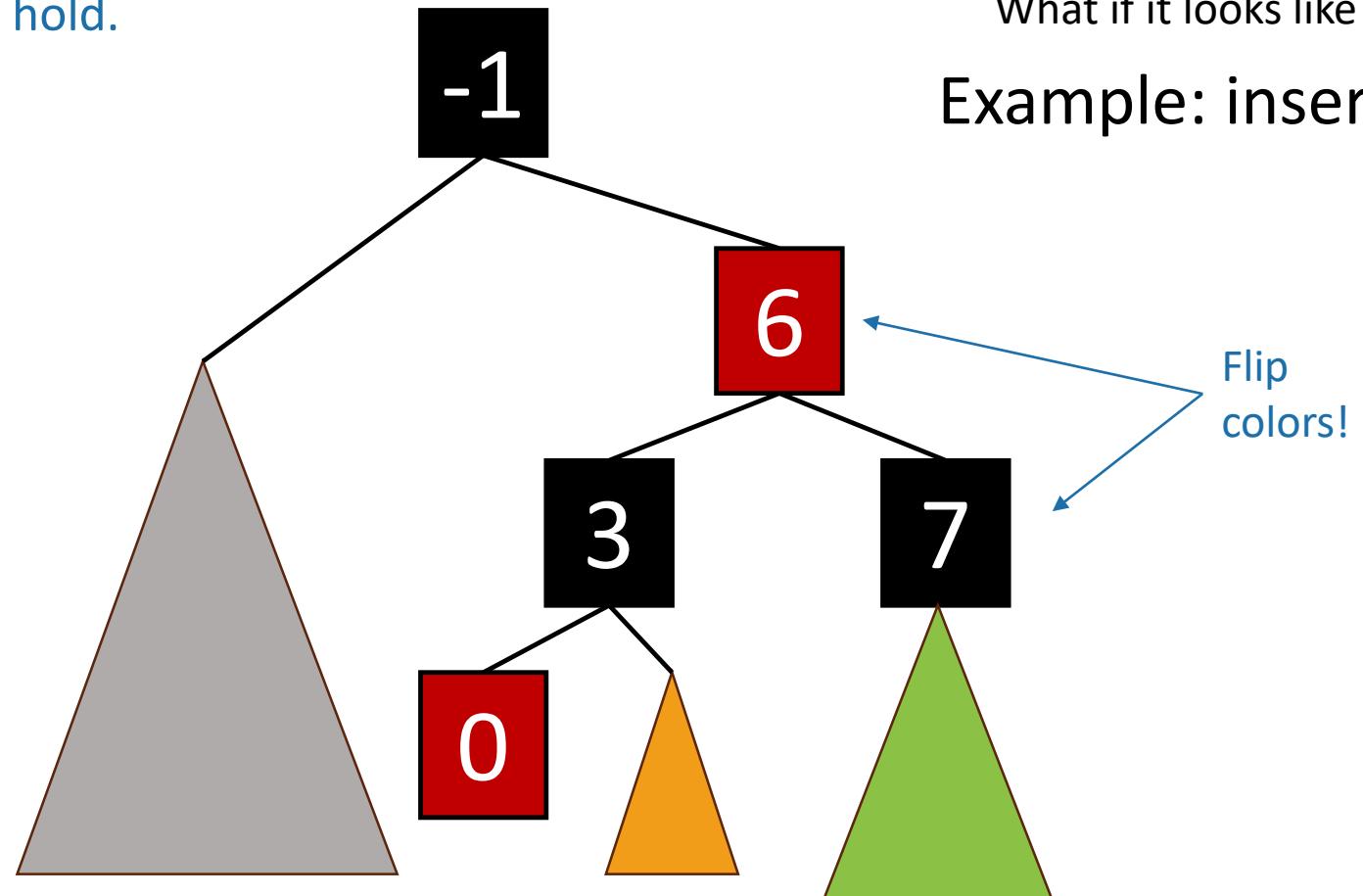
We need a bit more context

- Add 0 as a red node.
- **Claim:** RB-Tree properties still hold.

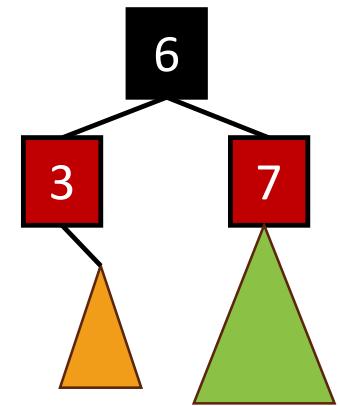
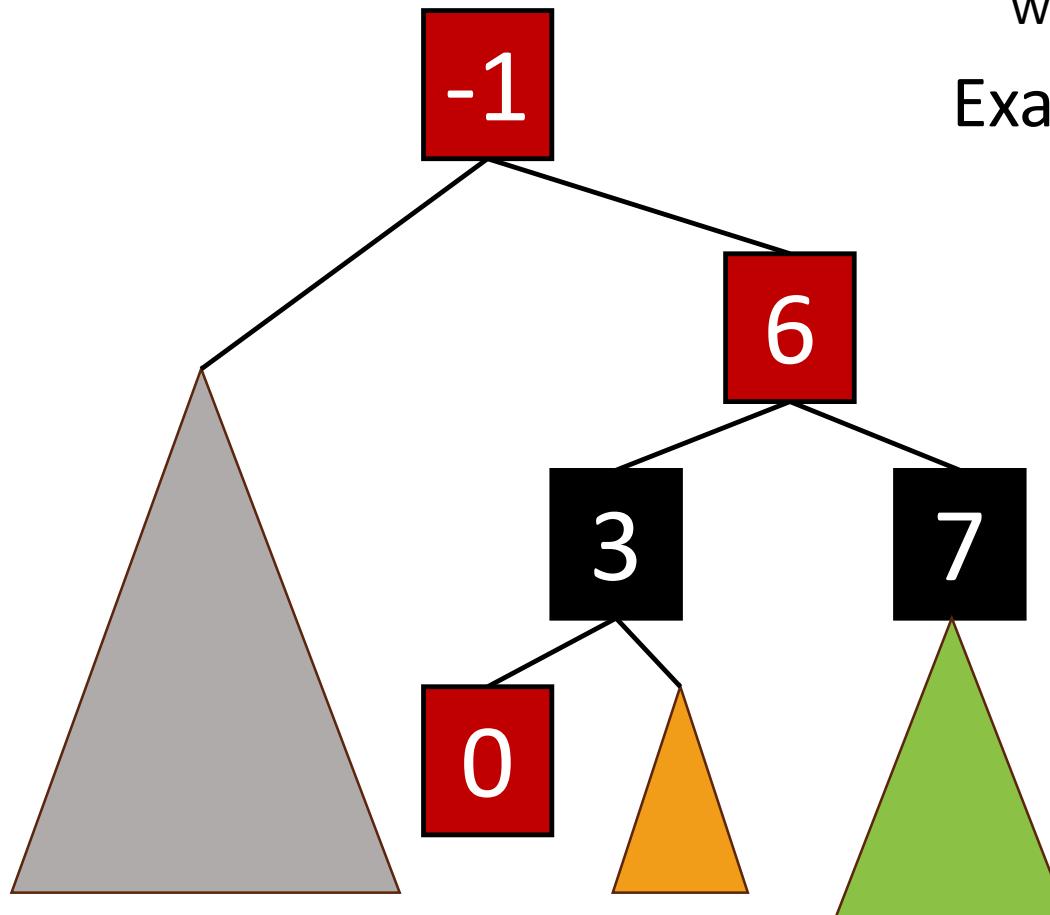


What if it looks like this?

Example: insert 0



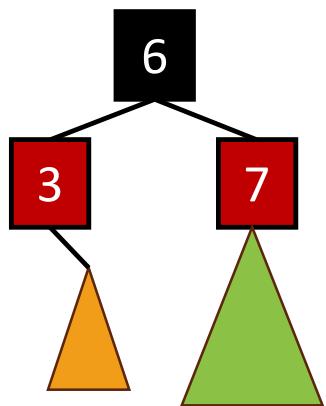
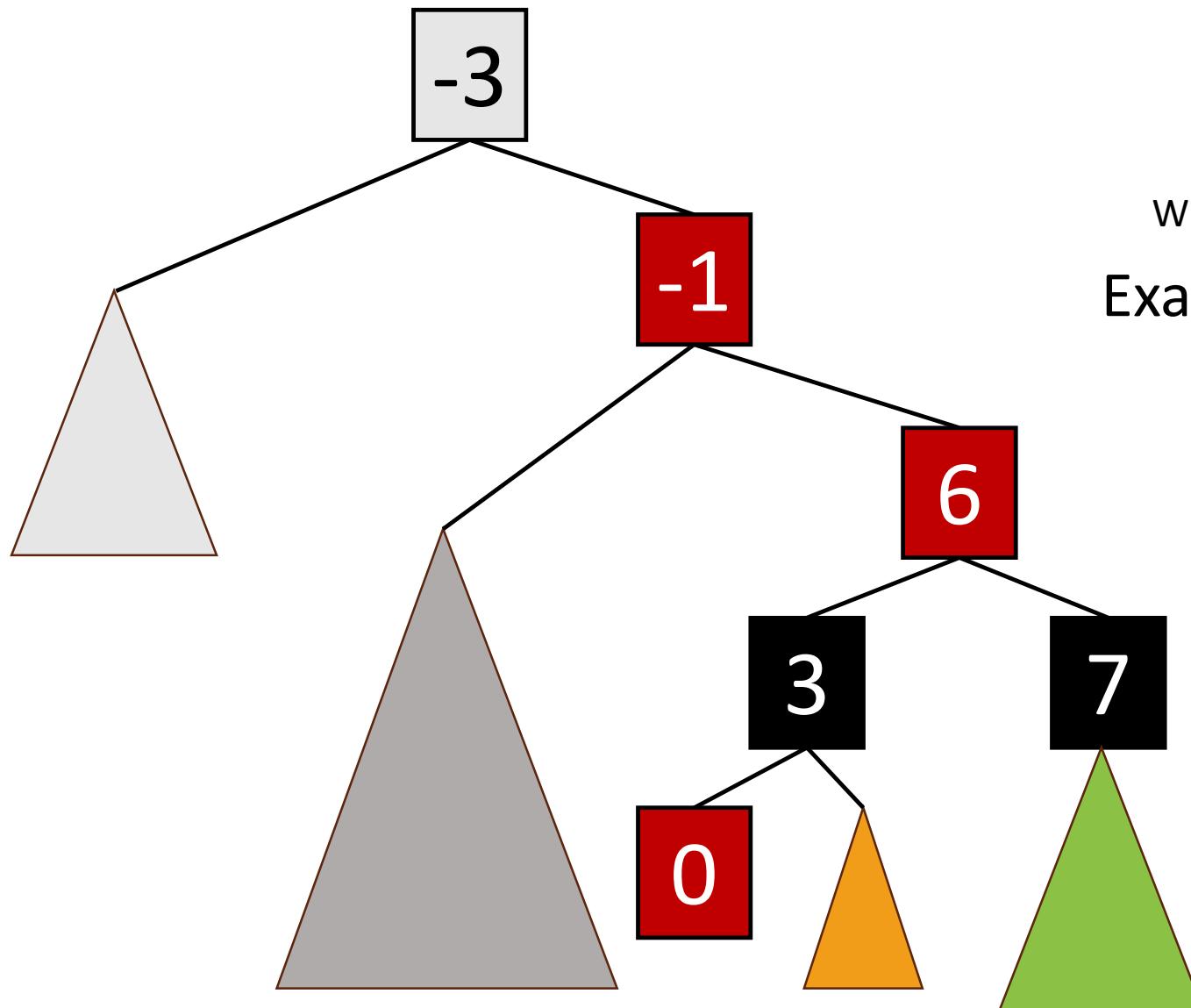
But what if **that** was red?



What if it looks like this?

Example: insert 0

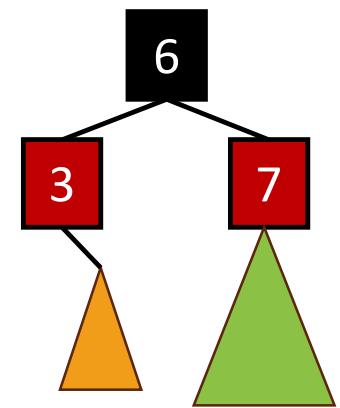
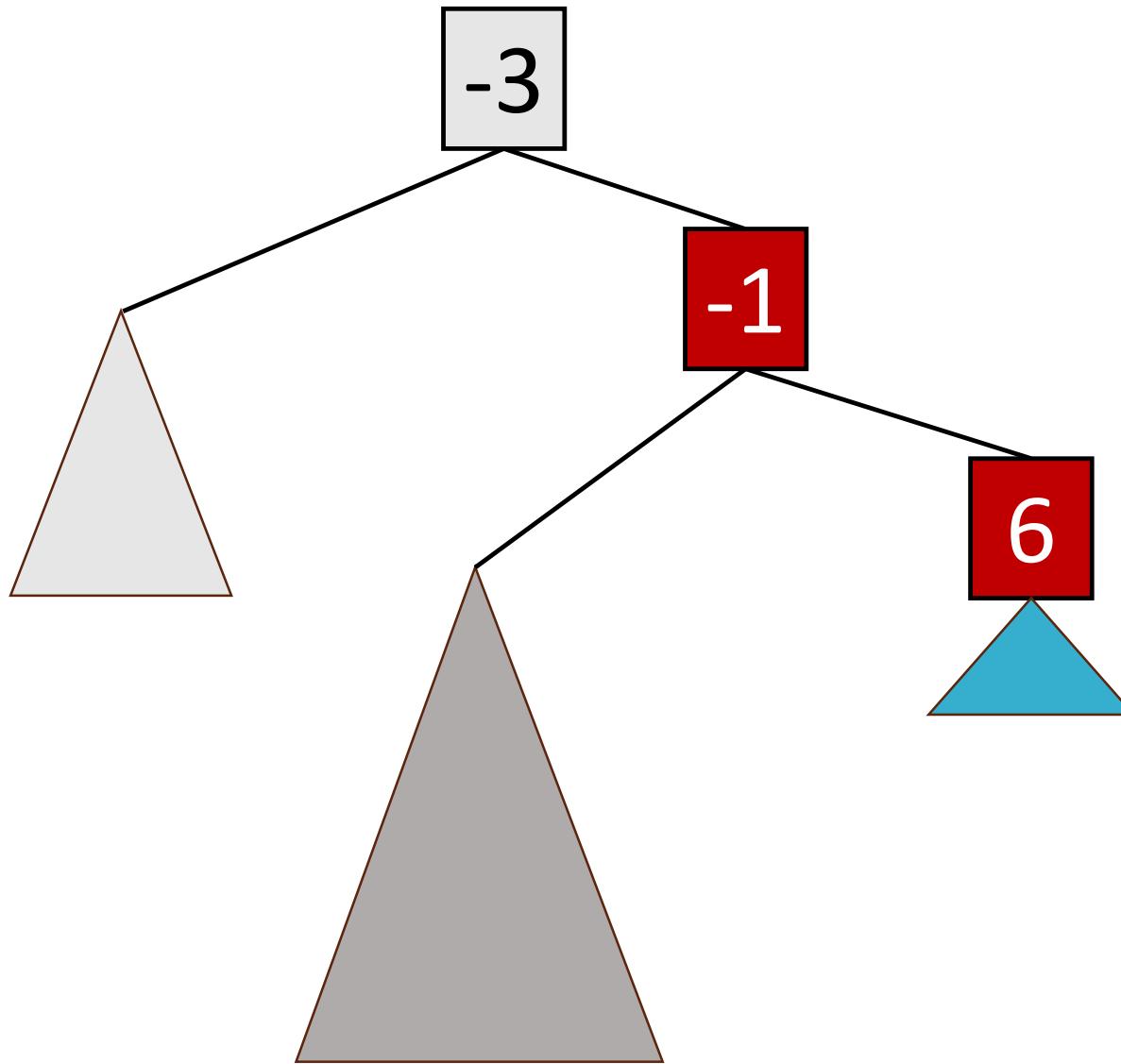
More context...



What if it looks like this?

Example: insert 0

More context...

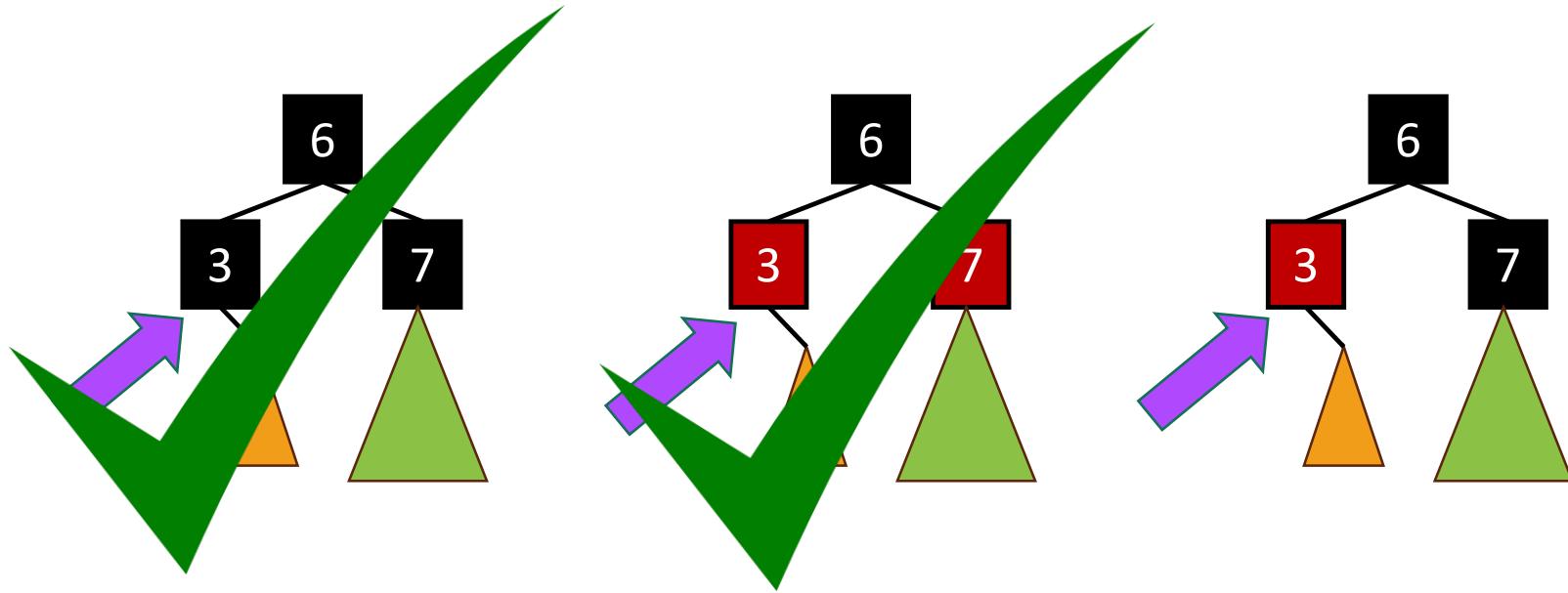


What if it looks like this?

Example: insert 0

Now we're
basically inserting
6 into some
smaller tree.
Recurse!

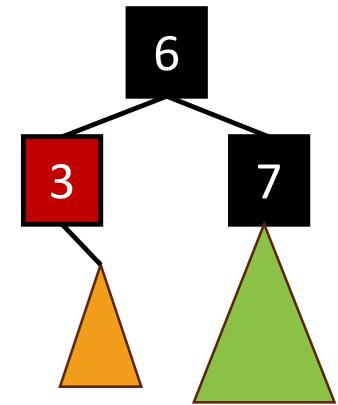
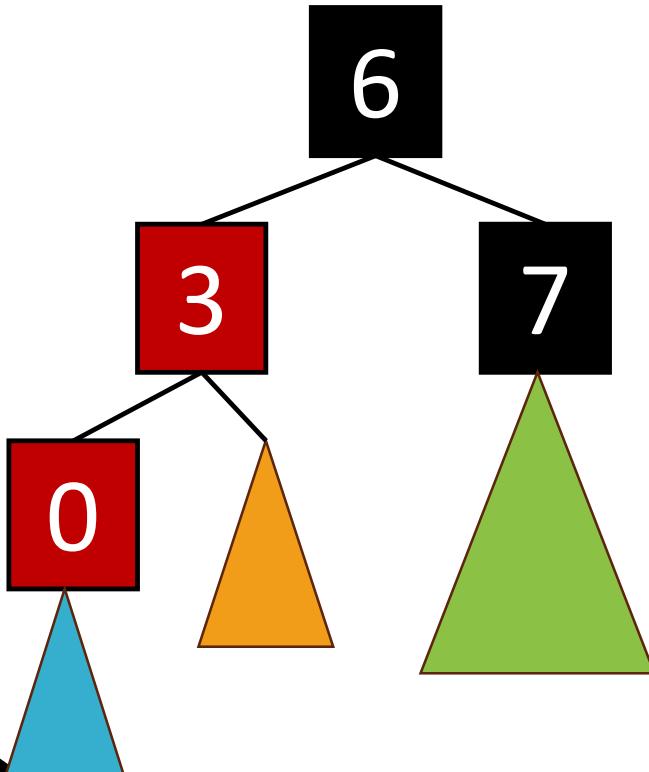
Many cases



- Suppose we want to insert **here**.
 - eg, want to insert 0.

Inserting into a Red-Black Tree

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.

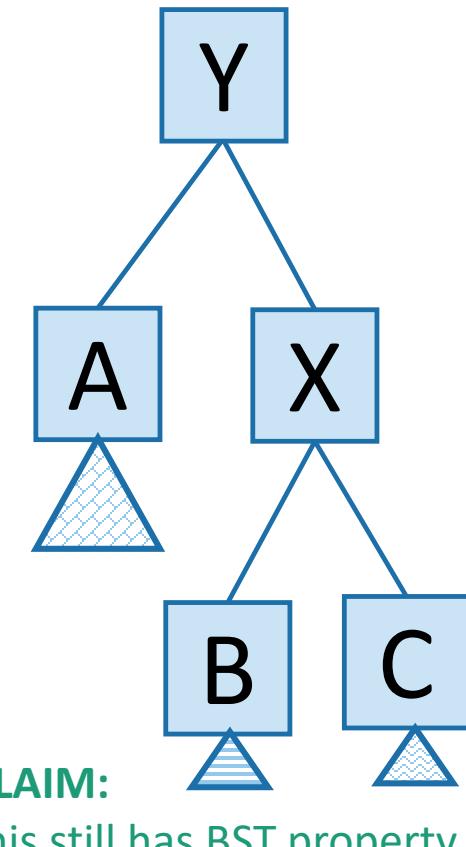
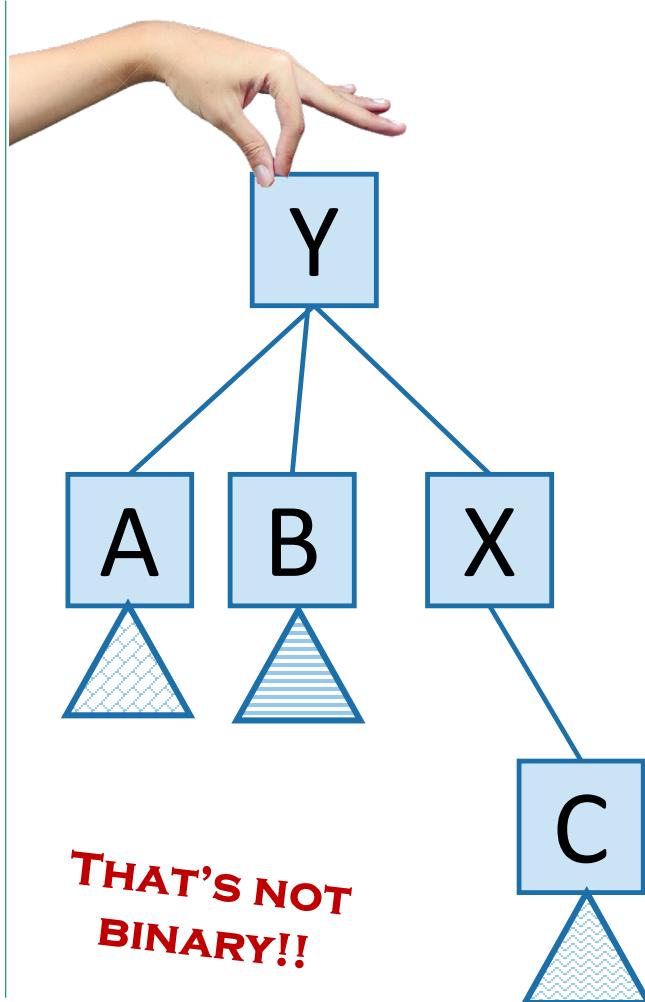
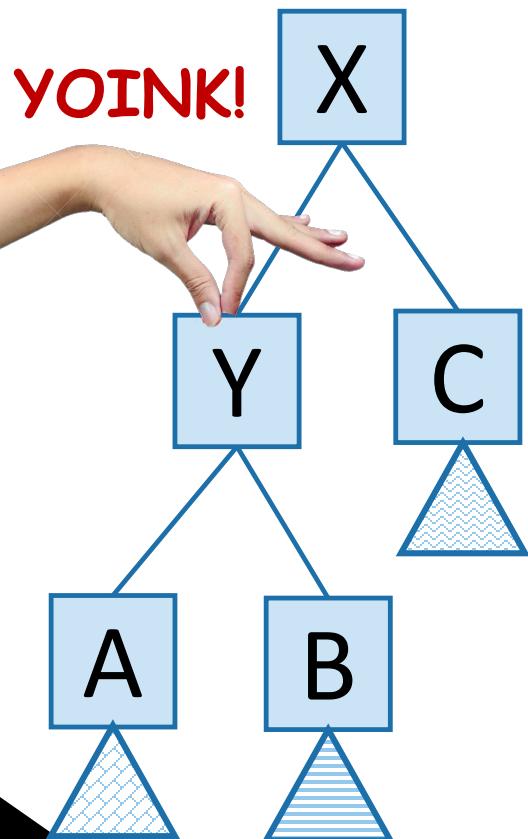


What if it looks like this?

- Example: Insert 0.
- Actually, **this can't happen?**
 - **6-3** path has one black node
 - **6-7-**... has at least two
 - It might happen that we just turned 0 red from the previous step.
 - Or it could happen if **7** is actually **NIL**.

Recall Rotations

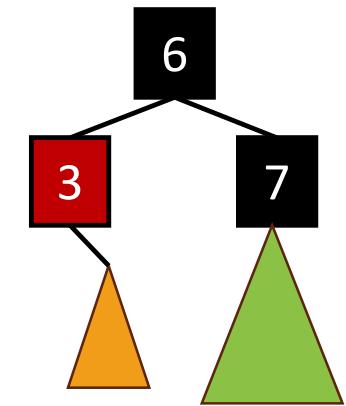
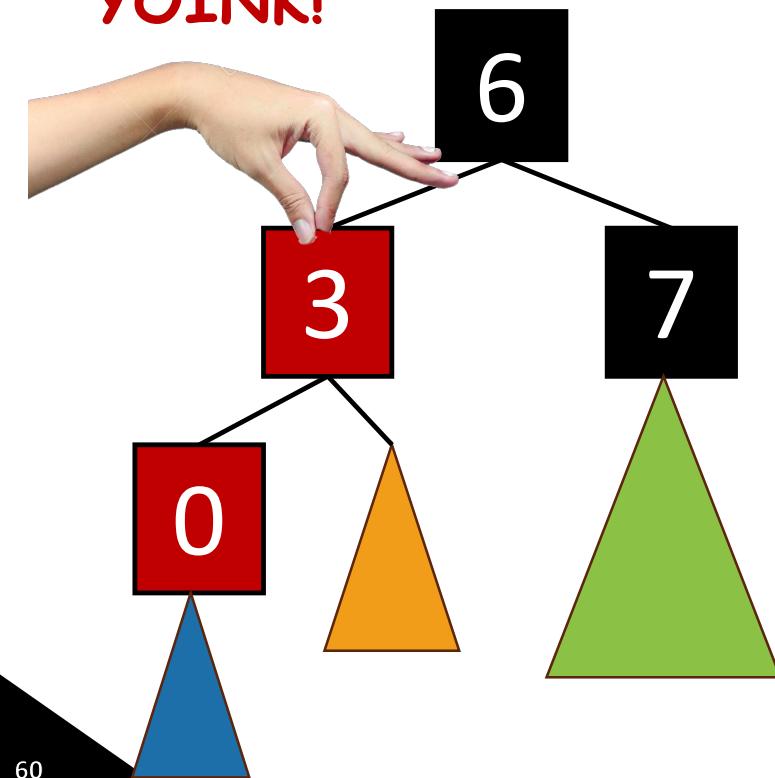
- Maintain Binary Search Tree (BST) property, while moving stuff around.



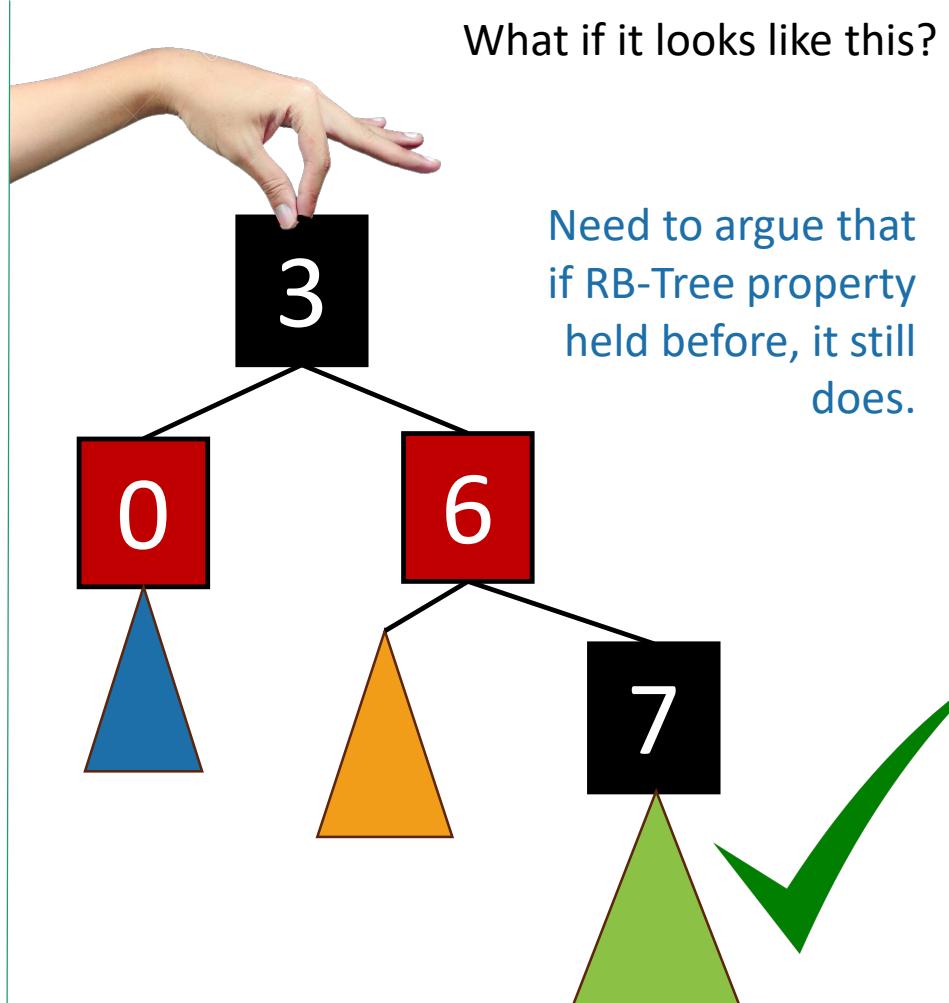
Inserting into a Red-Black Tree

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.

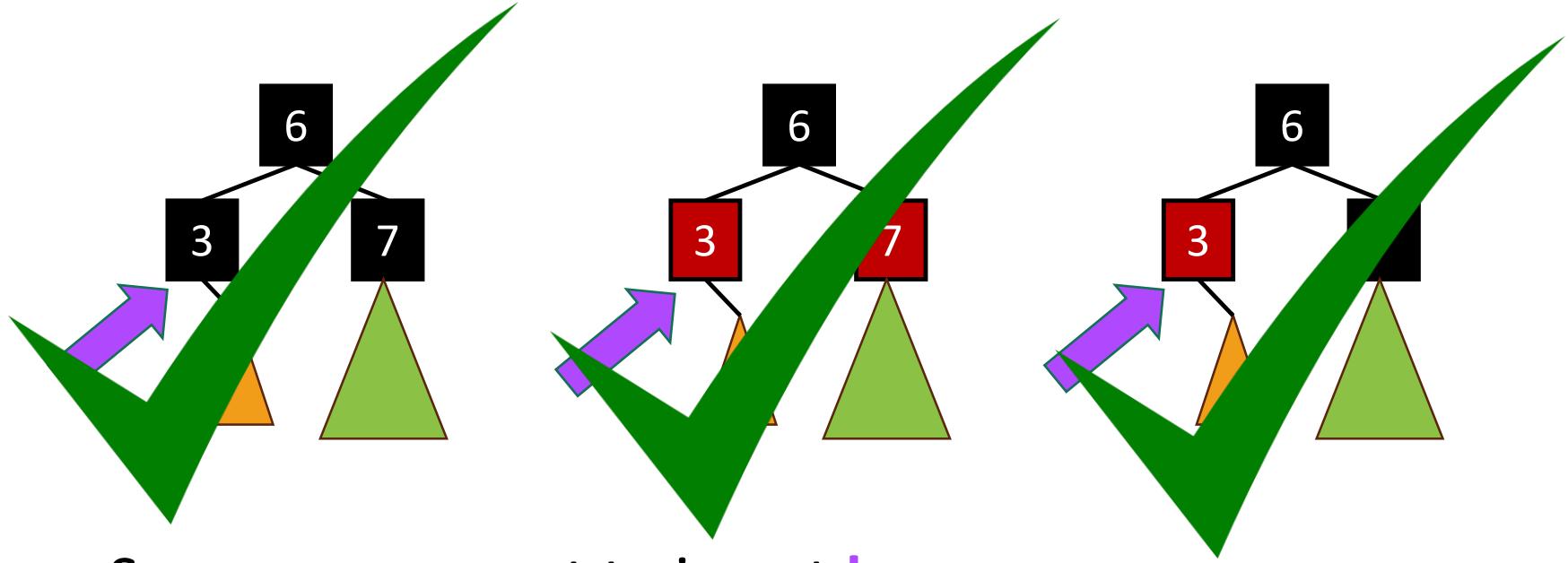
YOINK!



What if it looks like this?



Many cases



- Suppose we want to insert **here**.
 - eg, want to insert 0.

insert in Red-Black Trees

```
def rb_insert(root, key_to_insert):
    x = search(root, key_to_insert)
    v = new red vertex with key_to_insert
    if key_to_insert > x.key:
        x.right = v
        recolor(v)
    if key_to_insert < x.key:
        x.left = v
        recolor(v)
    if key_to_insert == x.key:
        return
```

Runtime: $O(\log n)$

insert in Red-Black Trees

```
def recolor(v):
    p = parent(x)
    if p.color == black:
        return
    grand_p = p.parent
    uncle = grand_p.right
    if uncle.color == red:
        p.color = black
        uncle.color = black
        grand_p.color = red
        recolor(grand_p)
    else: # uncle.color == black
        p.color = black
        grand_p.color = red
        right_rotate(grand_p) # yoink
```

recursivem
runtime $\log n$

Runtime: $O(\log n)$

Deleting from a Red-Black tree

How we do delete?

Fun exercise!



That's a lot of cases

- You are **not responsible** for the nitty-gritty details of Red-Black Trees. (For this class)
 - Though implementing them is a great exercise!
- You should know:
 - What are the properties of an RB tree?
 - And (more important) why does that guarantee that they are balanced?

What was the point again?

↘(M&O)

- Red-Black Trees **always** have height at most $\underline{2\log(n+1)}$,
- As with general **Binary Search Trees**, all operations are $O(\text{height})$
- So all operations are $O(\log(n))$.

Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Balanced Binary Search Trees
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$

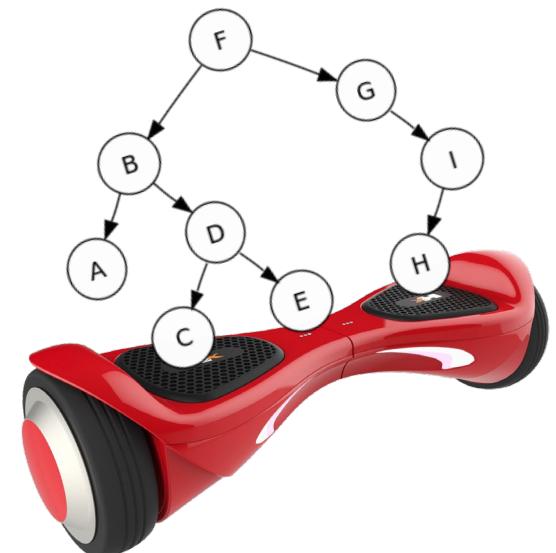
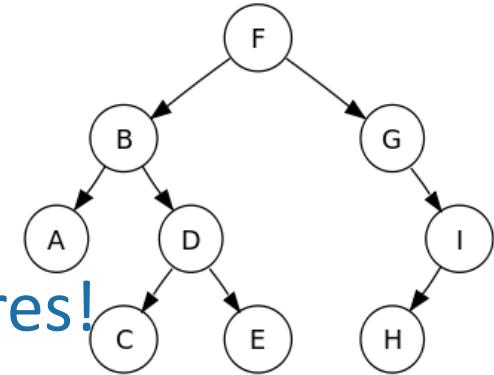
Today

- Begin a brief summary into **data structures!**
- Binary search trees
 - They are better when they're balanced.

this will lead us to...

- Self-Balancing Binary Search Trees
 - **Red-Black** trees.

Recap



Recap

- **Balanced binary trees** are the best of both worlds!
- But we need to **keep them balanced**.
- **Red-Black Trees** do that for us.
 - We get $O(\log(n))$ -time **INSERT/DELETE/SEARCH**
 - Clever idea: have a proxy for balance

