# CS 161

Design and Analysis of Algorithms

Summer 2018

# Outline

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

# Course Info

- Our website is live at cs161-sum18.github.io.
  - I'm trying to redirect cs161.stanford.edu to our website. Stay tuned.
  - Please read the Course Info page for our course policies, logistics, description.
  - Please read the Resources page for LaTeX, Python, and Homework resources.

# Course Info

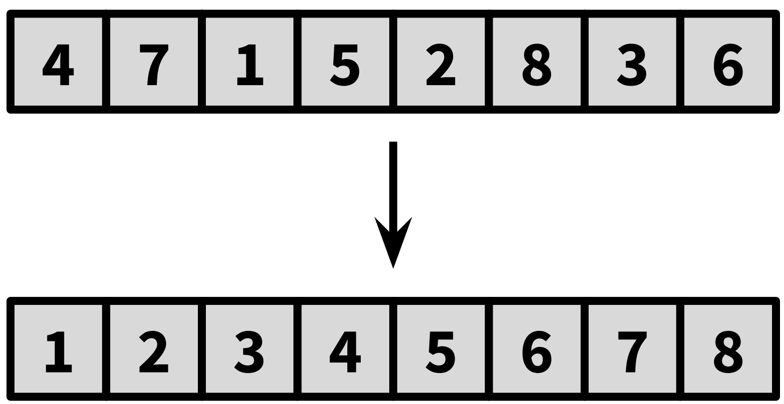
- Our website is live at cs161-sum18.github.io.
  - I'm trying to redirect cs161.stanford.edu to our website. Stay tuned.
  - Please read the Course Info page for our course policies, logistics, description.
  - Please read the Resources page for LaTeX, Python, and Homework resources.
- Homework 0 is live!
  - This assignment is "due" next Tuesday at 5 p.m.
  - No submission is required; it's worth 0% of your grade.
  - Use it as an opportunity (1) for self-assessment and (2) for style advice for future Homework submissions.

# Algorithmic Analysis

Summer 2018 • Lecture 06/26

# Sorting

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



# Insertion sort

```
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…

```
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?**

```python
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?**

```
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?**

```
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!

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

```python
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!

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

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

```python
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!

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 1 | 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

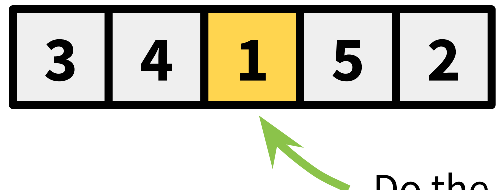
1. **Does this actually work?** Let’s see an example!

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 1 | 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!




Do the same thing for **`A[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!

| | | | | |
|---|---|---|---|---|
| 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

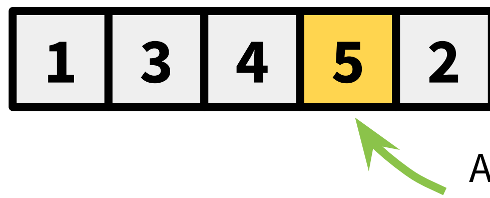
1. **Does this actually work?** Let's see an example!

| | | | | |
|---|---|---|---|---|
| 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!

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

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!

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

```
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!

```
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, well duh… 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, well duh… 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.

# Words of Wisdom

- 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?

# Words of Wisdom

- 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.

# 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, | 1 | 3 | 4 | 5 | , and another element | 2 | .

Inserting | 2 | immediately to the right of the largest element from the original list that's smaller than | 2 | (i.e. right of | 1 | ) produces another sorted list.

# Insertion sort

**Suppose you have a sorted list, | 1 | 3 | 4 | 5 | , and another element | 2 | .**

**Inserting | 2 | immediately to the right of the largest element from the original list that's smaller than | 2 | (i.e. right of | 1 | ) produces another sorted list. Notice that this new list is longer than the original one by one element: | 1 | 2 | 3 | 4 | 5 | .**

# Insertion sort

- **We can apply this logic at every step.**

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

# Insertion sort

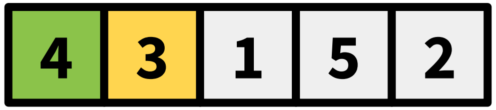
- **We can apply this logic at every step.**

| | | | | |
|---|---|---|---|---|
| **4** | **3** | **1** | **5** | **2** |

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

# Insertion sort

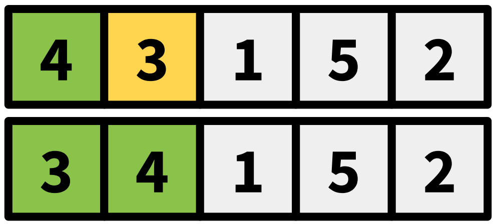
- **We can apply this logic at every step.**




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

# Insertion sort

- **We can apply this logic at every step.**




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.**

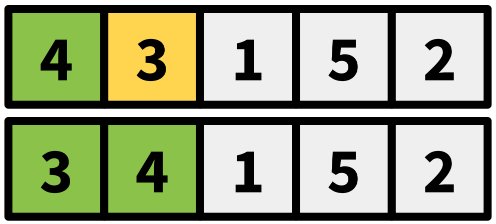
---

| | | | | |
|---|---|---|---|---|
| 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.**

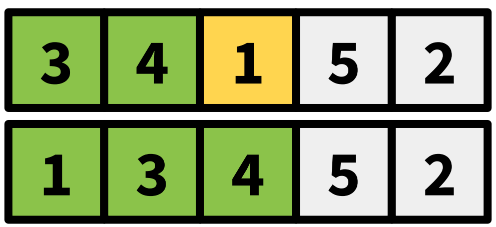
# Insertion sort

- **We can apply this logic at every step.**



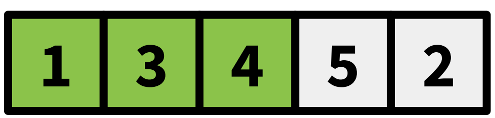

The first element, [4], is a sorted list. 3 is our other element.
**Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.**

---




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.**

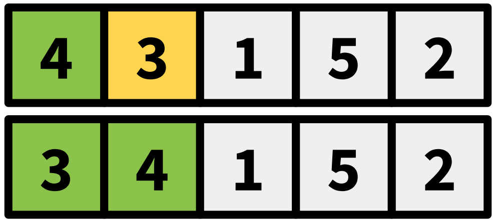
---



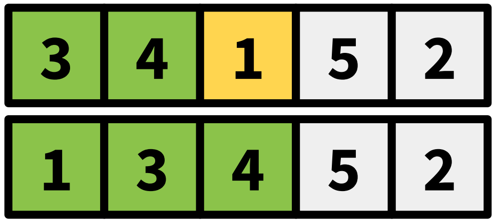

The first three elements, [1, 3, 4], are a sorted list.

# Insertion sort

- **We can apply this logic at every step.**




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.**

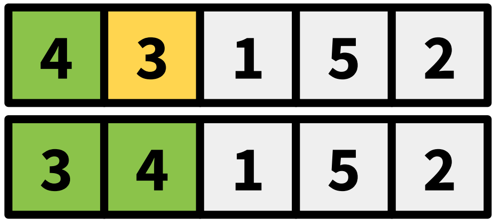



The first two elements, [3, 4], are a sorted list. 1 is our other element. **Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.**

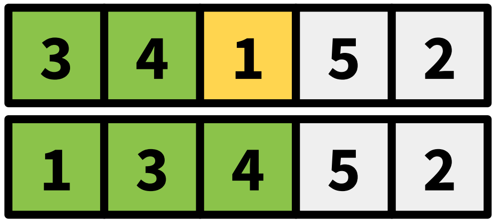

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element.

# Insertion sort

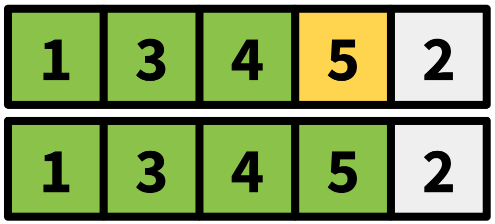
- **We can apply this logic at every step.**




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.**




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.**




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.**

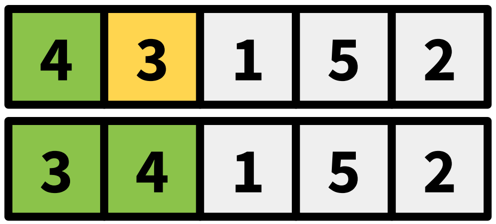
---

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

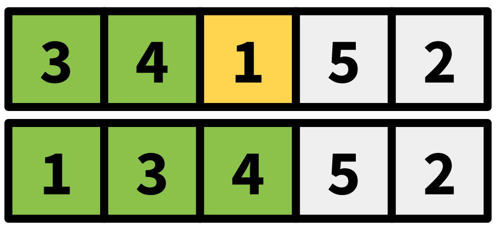
The first four elements, [1, 3, 4, 5], are a sorted list.

# Insertion sort

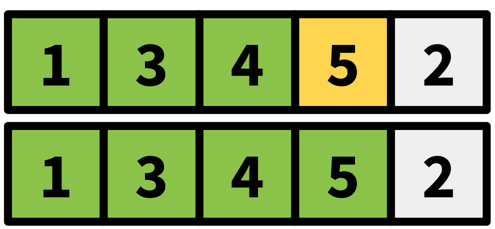
- **We can apply this logic at every step.**



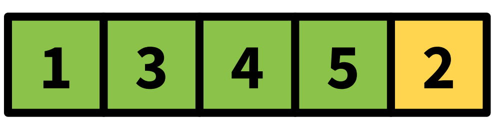

The first element, [4], is a sorted list. 3 is our other element. **Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.**




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.**



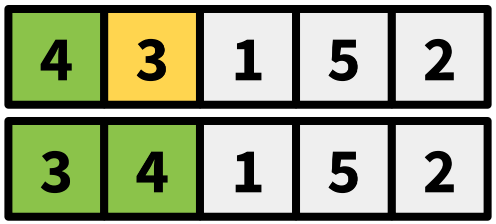

The first three elements, [1, 3, 4], are a sorted list. 5 is our other element. **Correctly inserting 5 into the sorted list [1, 3, 4] produces another sorted list [1, 3, 4, 5] that's longer by one element.**



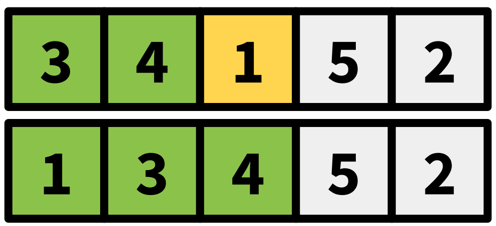

The first four elements, [1, 3, 4, 5], are a sorted list. 2 is our other element.

# Insertion sort

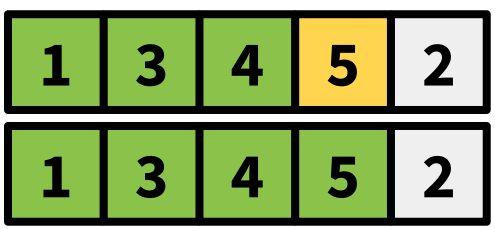
- **We can apply this logic at every step.**



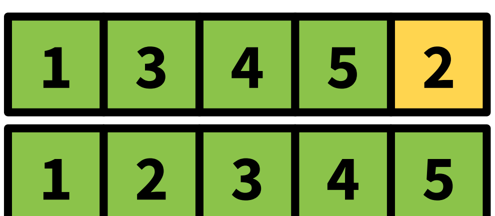

The first element, [4], is a sorted list. 3 is our other element.
**Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.**




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.**




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.**




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**.

# 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.

# Proving Correctness

- Recall, there are four components in a proof by induction.
  - **Inductive Hypothesis** The loop invariant holds after the ith iteration.
  - **Base case** The loop invariant holds before the first iteration.
  - **Inductive step** If the loop invariant holds after the ith 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!

# Proving Correctness

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

# Proving Correctness

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

I will use 0-index and right-exclusive (same as Python) array-notation.

# Proving Correctness

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

I will use 0-index and right-exclusive (same as Python) array-notation.

# 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.

I will use 0-index and right-exclusive (same as Python) array-notation.

# Proving Correctness

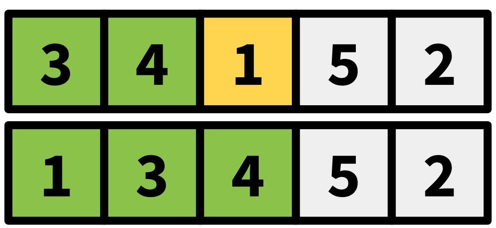
- Loop invariant(i): **`A[:i+1]`** is sorted.
- Formally, for insertion sort...
  - **Inductive Hypothesis** The loop invariant(i) holds at the end of iteration i of the outer loop i.e. **`A[:i+1]`** is sorted.
  - **Base case** The loop invariant(i) holds before the algorithm starts when i = 0 i.e.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

I will use 0-index and right-exclusive (same as Python) array-notation.

# 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 (see Lecture Notes for details).

I will use 0-index and right-exclusive (same as Python) array-notation.



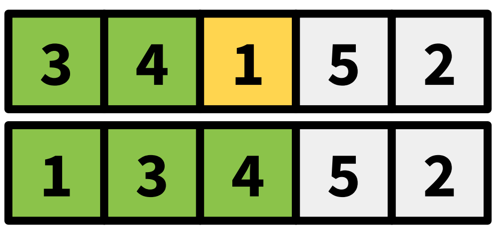

The first two elements, [3, 4], are a sorted list. 1 is our other element. **Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that's longer by one element.**

# Proving Correctness

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

I will use 0-index and right-exclusive (same as Python) array-notation.

  - **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 (see Lecture Notes for details).




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

- It turns out proving the logic from the animation requires another proof by induction and involves another loop invariant!
    - Recall, there's a inner `while` loop that mutates the list.

```
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
```

# Proving Correctness

- It turns out proving the logic from the animation requires another proof by induction and involves another loop invariant!
  - Recall, there's a inner **`while`** loop that mutates the list.
  - To whet your appetite (yum!)... Loop invariant(j): **`A[0:j,j+1:i+1]`** contains the same elements as the original sublist **`A[0:i]`**, still sorted, such that all of the values in the right sublist **`A[j+1:i+1]`** are greater than **`cur_value`**.

```
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
```

# Proving Correctness

- Another way to think of proofs by induction for iterative algorithms…
  - **Inductive Hypothesis** The loop invariant holds after the ith iteration.
  - **Base case** The loop invariant holds before the first iteration.
    - **“Initialization”**
  - **Inductive step** If the loop invariant holds after the ith 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, well duh… 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, well duh… 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, well duh… 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, well duh… 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?

# 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**

# 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**

Let's develop notation so we don't need to be so verbose.

# 5-min Break

# Outline

- ~~Course Info~~ **Done!**
- 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

# 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?
    - 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?

# Big-O Notation

- What does it mean to measure “runtime” of an algorithm?
  - Engineers probably care most about the “wall time”: how long does the algorithm take in seconds, minutes, hours, days, etc.?
  - This heavily depends on computer hardware, programming language, etc.
  - 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.

# Big-O Notation

- **Key insight** Focus on how the runtime scales with n (the input size).
  - **Pros** (1) It controls for computer hardware, programming language, and other considerations. (2) Less of a trial-and-error process.
  - **Cons** It only makes sense if n is large compared to the constant factors.

Should 9,999,999,999,999n be "better than" $n^2$???

# Big-O Means Upper-Bound

- Big-O notation is a mathematical notation for upper-bounding a function’s rate of growth.
    - Informally, it can be determined by ignoring constants and non-dominant growth terms.

# 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,

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

$$\textbf{iff}$$

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

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

# Big-O Notation

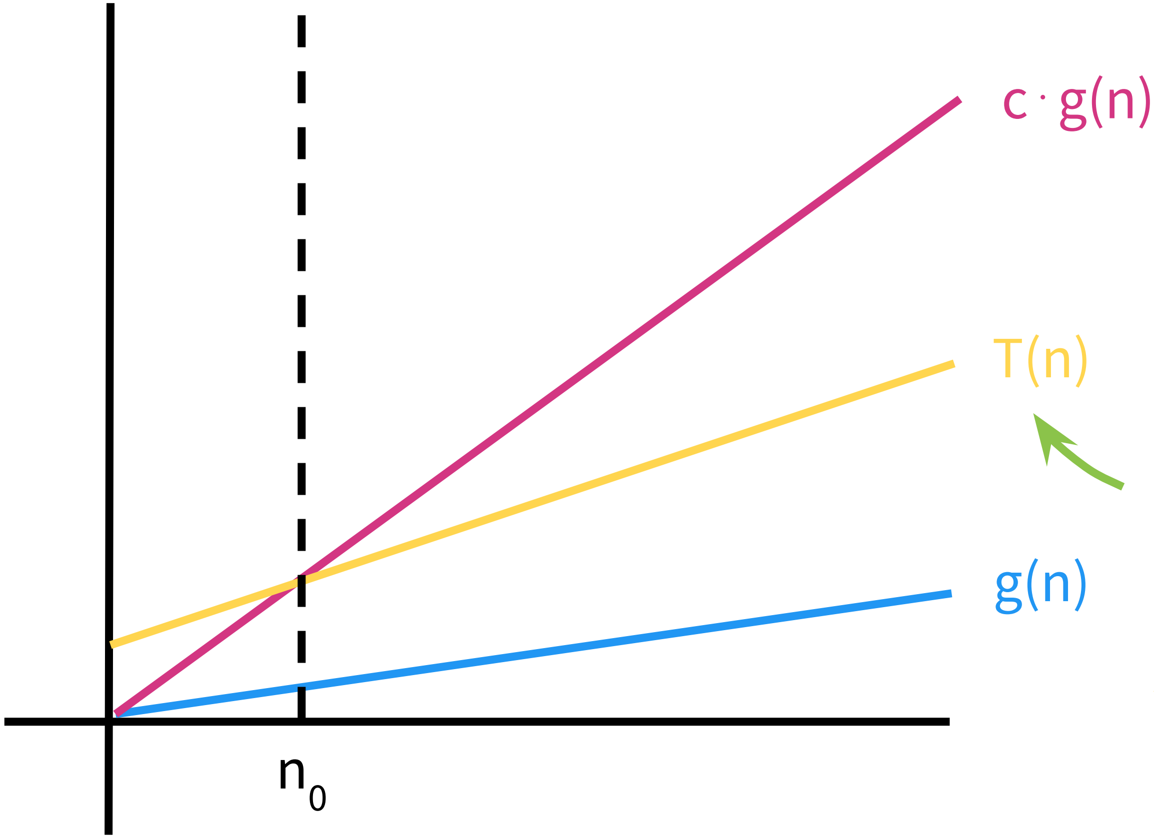
$T(n) = O(g(n))$

**iff**

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

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

- 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

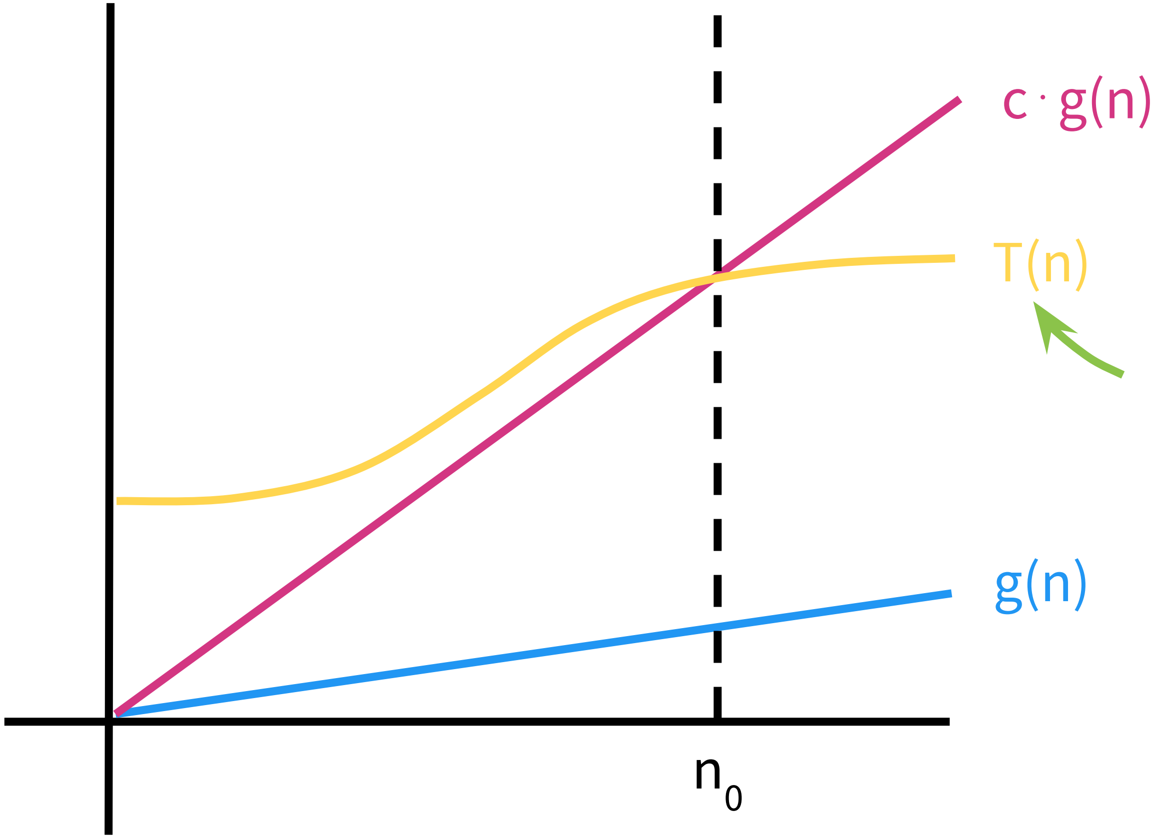
$T(n) = O(g(n))$

iff

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

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

- Graphically,




T(n) doesn't always have to be linear.

# Big-O Notation

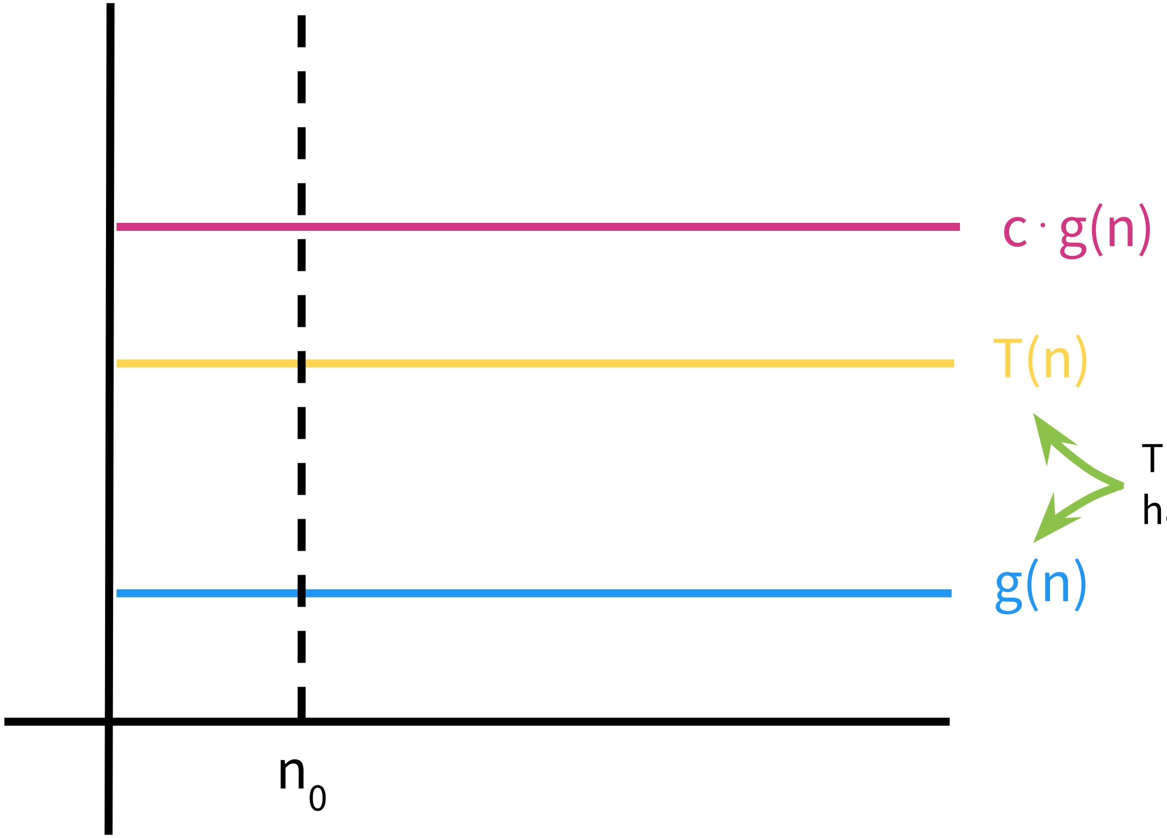
$T(n) = O(g(n))$

iff

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

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

- Graphically,




T(n) and g(n) do not always have to be increasing.

# Big-O Notation

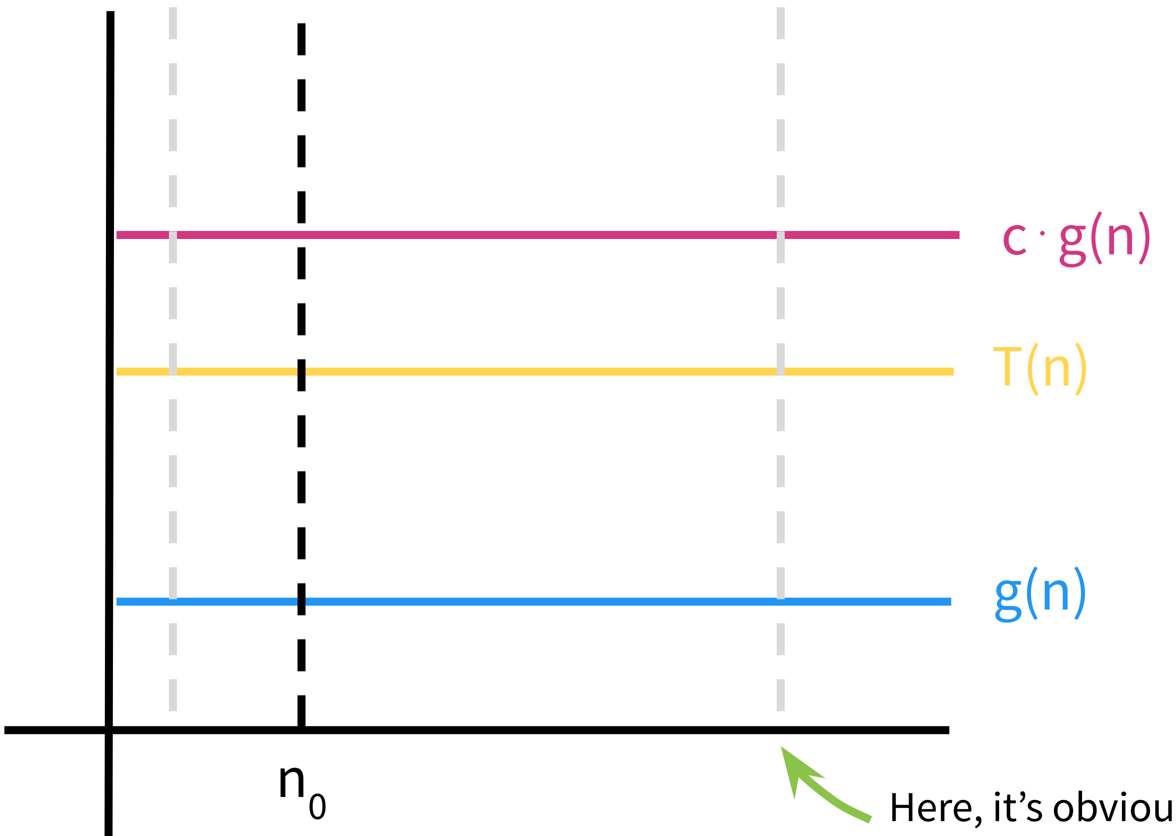
$T(n) = O(g(n))$

iff

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

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

- Graphically,




Here, it's obvious that there isn't one "correct" choice for $n_0$.

# Big-O Notation

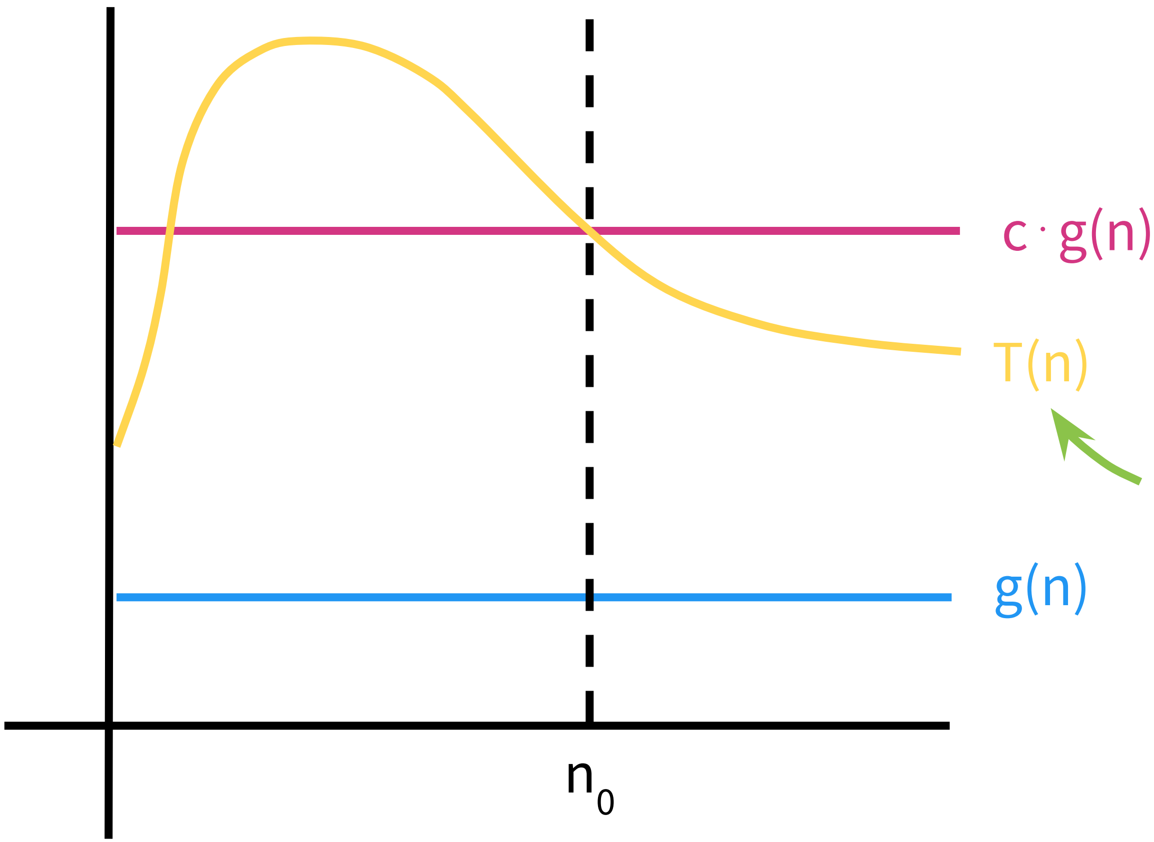
$T(n) = O(g(n))$

iff

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

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

- Graphically,




T(n) looks weird, but it's still O(g(n)).

# Big-O Notation

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

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

iff

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

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

# Big-O Notation

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

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.
  - **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 $n \geq 2$.

# Big-O Notation

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

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.
    - **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.

# Big-O Notation

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

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.
  - **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 $n^2 > c \cdot n$.

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

iff

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

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

# Big-O Notation

- 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-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 Ω(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!

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

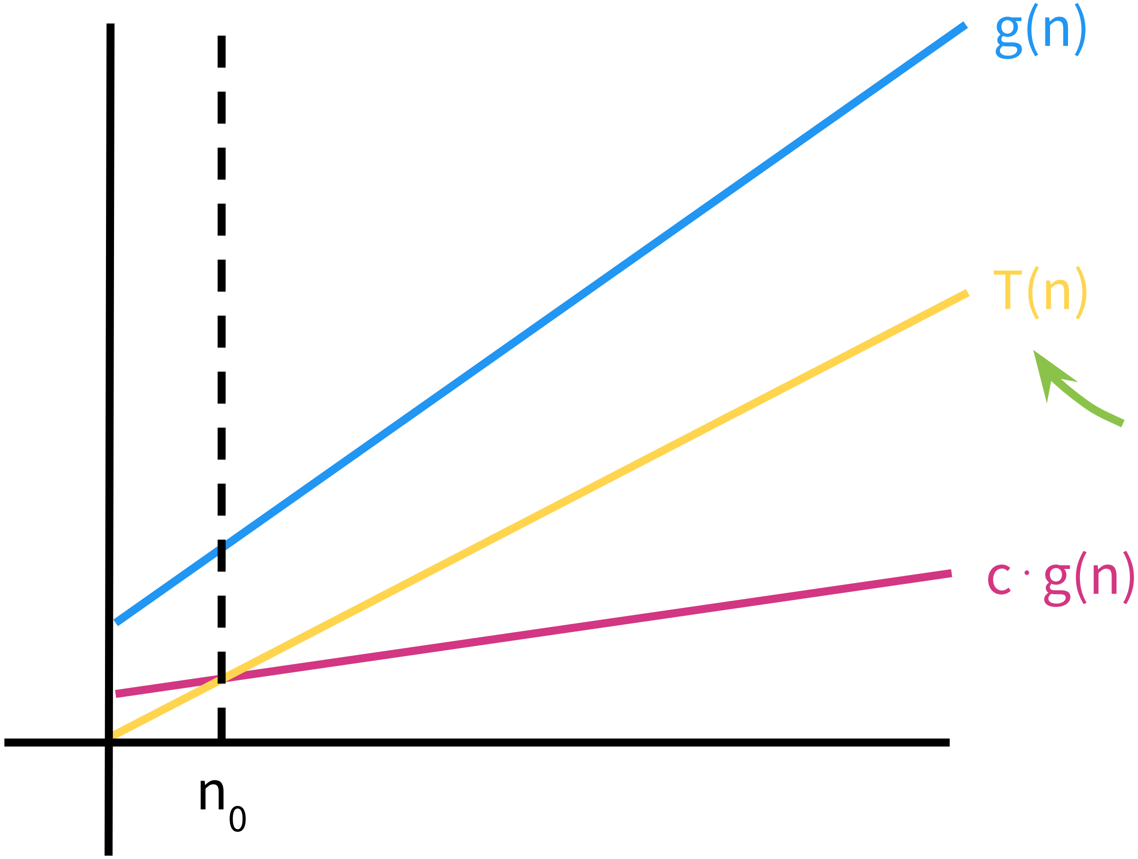
iff

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

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

# Big-Ω Notation

- Graphically,




Big-**Ω** defines **"T(n) = Ω(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 Θ(g(n))"** iff

**T(n) = O(g(n))**

**AND**

**T(n) is Ω(g(n))**

# 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?

| | | | | | | |
|---|---|---|---|---|---|---|
| n | ... | 5 | 4 | 3 | 2 | 1 |

Reverse order

```
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?

| | | | | | | |
|---|---|---|---|---|---|---|
| **1** | **2** | **3** | **4** | **5** | **...** | **n** |

Sorted order

```
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 $\mathbf{\Theta(n^2)}$.
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
    - The best-case runtime of insertion sort is $\mathbf{\Theta(n)}$.
  - **Average-case analysis** What is the runtime of the algorithm on the average input?

# 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 **$\Theta(n^2)$**.
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
    - The best-case runtime of insertion sort is **$\Theta(n)$**.
  - **Average-case analysis** What is the runtime of the algorithm on the average input?

**A common confusion** Why isn't this $O(n^2)$? It would also be correct to say the worst-case runtime of insertion sort is $O(n^2)$ since every function that's $\Theta(n^2)$ must also be $O(n^2)$. In fact, worst-case runtimes are usually reported with Big-O since we really only care about upper-bounding the worst-case. However, I reported the runtime with Big-Θ to emphasize the point that "worst-case runtime" describes the runtime of an algorithm on a specific input (namely, a worst-case input), and that we *infer* from this specific runtime that all possible other inputs are faster.

# 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 **$\Theta(n^2)$**.
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
    - The best-case runtime of insertion sort is **$\Theta(n)$**.
  - **Average-case analysis** What is the runtime of the algorithm on the average input?
    - We'll worry about this type of analysis when we cover Randomized Algorithms!

# 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 **$\Theta(n^2)$**.
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
    - The best-case runtime of insertion sort is **$\Theta(n)$**.
  - **Average-case analysis** What is the runtime of the algorithm on the average input?
    - We'll worry about this type of analysis when we cover Randomized Algorithms!
- When someone asks "What is the runtime of this algorithm?" it's implied to mean "What is the upper-bound for the worst-case runtime of this algorithm?" but, technically, it's ambiguous.

# 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 <u>worst-case</u> runtime $O(n^2)$**

# Analyzing Runtime

```python
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 <u>best-case</u> runtime $\Omega(n)$**

# Outline

- ~~Course Info~~ **Done!**
- 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

# Outline

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