Introduction to Algorithms

L1. Algorithmic analysis. I

Instructor : Kilho Lee

Last time

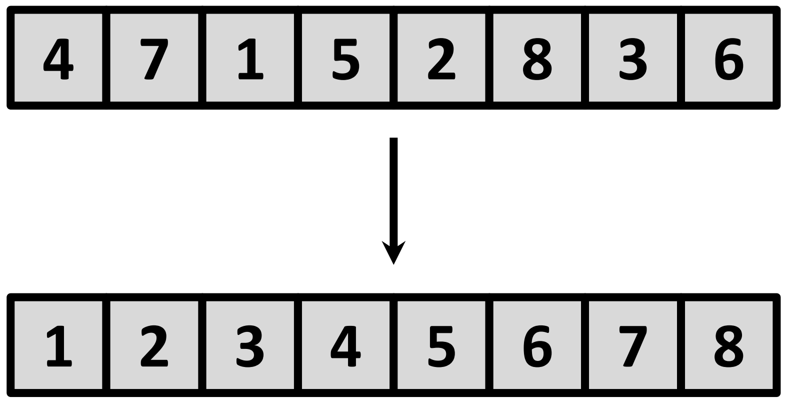
- Course Goals
  - What is an algorithm?
  - Why we study algorithms?
    - Fundamental, useful, fun
  - Goal: design and analysis of algorithms
  - Algorithm designer's questions

Today: 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

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

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…

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

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?**

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

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?**

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

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?**

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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

4	3	1	5	2

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

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).

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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

3	4	1	5	2

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

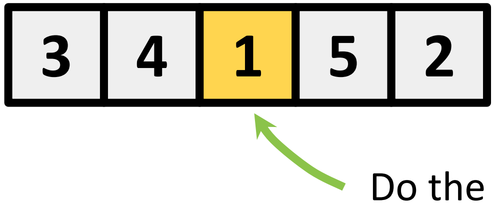
1. **Does this actually work?** Let’s see an example!

3	4	1	5	2

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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



Do the same thing for **`A[2]`**.

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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

1	3	4	5	2

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

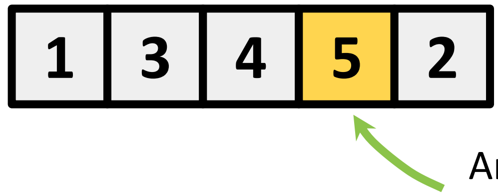
1. **Does this actually work?** Let's see an example!

1	3	4	5	2

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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



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

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

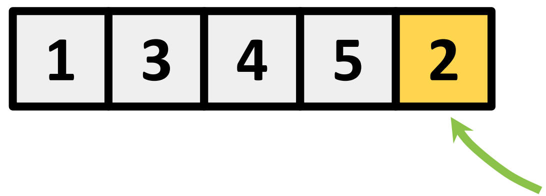
1. **Does this actually work?** Let's see an example!

1	3	4	5	2

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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



And lastly for **`A[4]`**.

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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

1	2	3	4	5

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Insertion sort

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

1	2	3	4	5

Then we're done!

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

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.

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

2. **Is it fast?**

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.

- (모든 경우에 대해서 작동한다는 것을 보이기 위해, 변치 않는 특성이 무엇인지 관찰해보자)

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, $\boxed{1}\boxed{3}\boxed{4}\boxed{5}$, and another element $\boxed{2}$.

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

Insertion sort

Suppose you have a sorted list, $\boxed{1}\boxed{3}\boxed{4}\boxed{5}$, and another element $\boxed{2}$.

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

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.**

4	3	1	5	2

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

Insertion sort

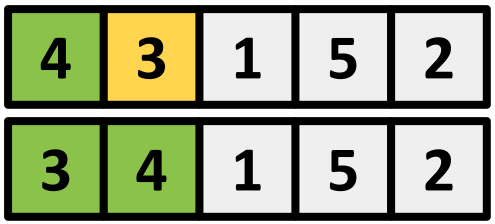
- **We can apply this logic at every step.**

4	3	1	5	2
3	4	1	5	2

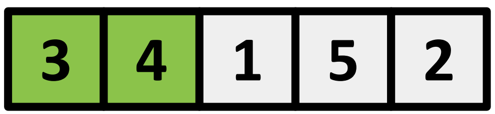
The first element, [4], is a sorted list. 3 is our other element. **Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.**

Insertion sort

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



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.

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

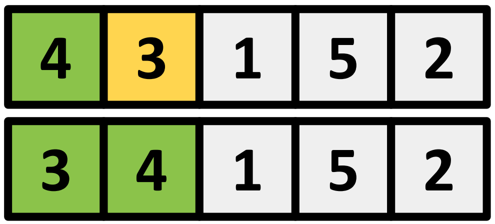
The first two elements, [3, 4], are a sorted list. 1 is our other element. **Correctly inserting 1 into the sorted list [3, 4] produces another sorted list [1, 3, 4] that’s longer by one element.**

1	3	4	5	2

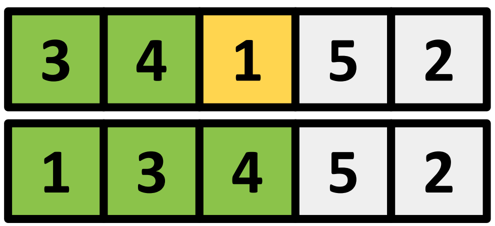
The first three elements, [1, 3, 4], are a sorted list.

Insertion sort

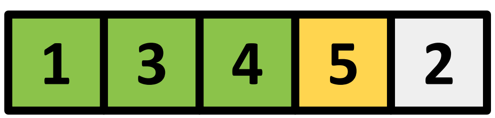
- **We can apply this logic at every step.**



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.**

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

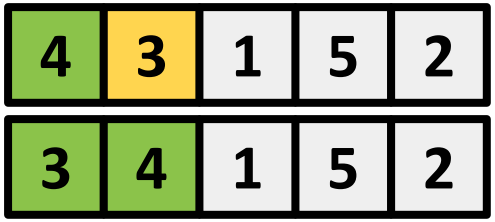
The first three elements, [1, 3, 4], are a sorted list. 5 is our other element. **Correctly inserting 5 into the sorted list [1, 3, 4] produces another sorted list [1, 3, 4, 5] that's longer by one element.**

1	3	4	5	2

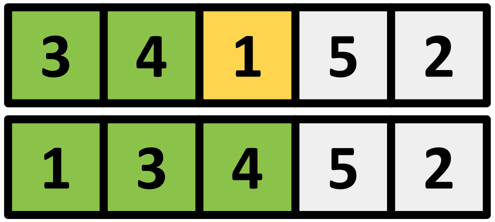
The first four elements, [1, 3, 4, 5], are a sorted list. 2 is our other element.

Insertion sort

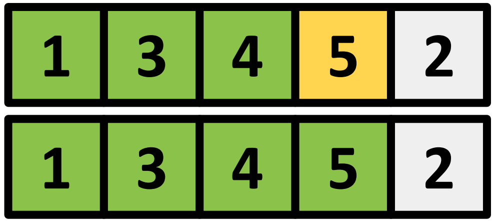
- **We can apply this logic at every step.**



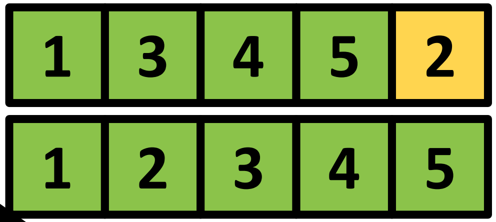
The first element, [4], is a sorted list. 3 is our other element. **Correctly inserting 3 into the sorted list [4] produces another sorted list [3, 4] that's longer by one element.**



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**. (루프 불변성)

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 end of iteration i (the iteration where we try to insert element **`A[i]`** 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.
- Formally, for insertion sort...

Proving Correctness

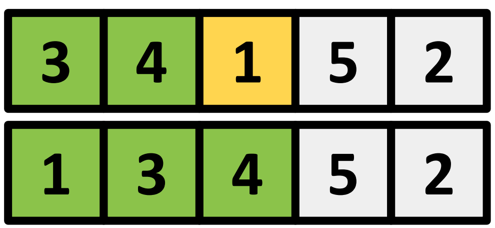
- Loop invariant(i): **`A[:i+1]`** is sorted.
- Formally, for insertion sort...
  - **Inductive Hypothesis** The loop invariant(i) holds at the end of iteration i of the outer loop (i.e. **`A[:i+1]`** is sorted).

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

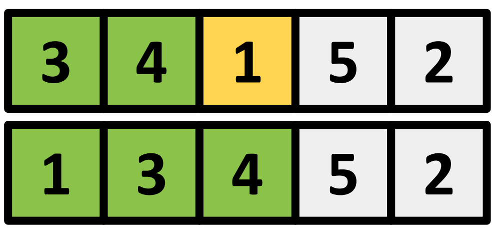
- Loop invariant(i): **`A[:i+1]`** is sorted.
- Formally, for insertion sort...
  - **Inductive Hypothesis** The loop invariant(i) holds at the end of iteration i of the outer loop i.e. **`A[:i+1]`** is sorted.
  - **Base case** The loop invariant(i) holds before the algorithm starts when i = 0 i.e.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



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.



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 modifies the list.

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

Proving Correctness

```
while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
```

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 before the i-th iteration, then it holds after the i-th 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?

Analyzing Runtime

```
void sort (int l[], int N)
{
     for (int i=1; i<N; i++)
     {
          int key = l[i];
          int j = i-1;
          while (j >= 0 && l[j] > key)
          {
               l[j+1] = l[j];
               j--;
          }
          l[j+1] = key;
     }
```

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

Post any question
On the Q&A board.