

# CPS 305

**Data Structures**

**Prof. Alex Ufkes**

**Topic 3:** Linked VS array data, searching & basic sorting

# Notice!

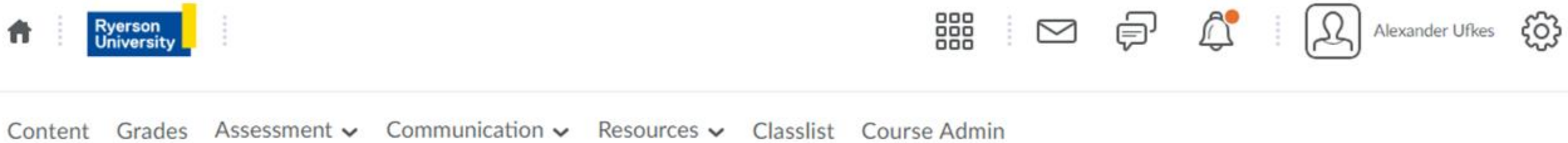
---

## **Obligatory copyright notice in the age of digital delivery and online classrooms:**

*The copyright to this original work is held by Alex Ufkes. Students registered in course CPS 305 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.*

# Course Administration

---



- Lab 2 posted
- Attend the lab to get help from your TA.

# Previously

---

## Two simple steps:

- 1) If  $f(n)$  is the sum of several terms, keep only the fastest growing.
- 2) If the remaining term is a product of several factors, remove any coefficients.

$$f(n) = 2n^2 + 4n + 9$$

- The remaining term is our Big-O complexity class.
- In this case,  $O(n^2)$

# Previously

---

But why? Doesn't this stuff matter?

$$\text{Let } f(n) = \underline{2n^2} + \underline{4n} + 9$$

**Big-O is a measure of algorithm cost as  $n$  approaches infinity:**

- As  $n$  gets ***larger***, the proportional contribution of the removed terms to the total run-time gets ***smaller***.
- As  $n$  approaches ***infinity***, the proportional contribution of the removed terms approaches ***zero***.
- High order coefficient? The ratio asymptotically approaches *that coefficient*.

# Previously

---

All log bases  
are equivalent



$O(1)$

constant

$O(\log_b(n))$

logarithmic

$O(n)$

linear

$O(n \ln(n))$

log-linear, “n-log-n”

$O(n^2)$

quadratic

$O(n^3)$

cubic

$O(n^b)$

“polynomial”

$O(b^n)$

“exponential”

$O(n!)$

factorial

**Careful!** Polynomials  
and exponentials with  
different values of **b** are  
in different classes.



# Moving On...

# Today

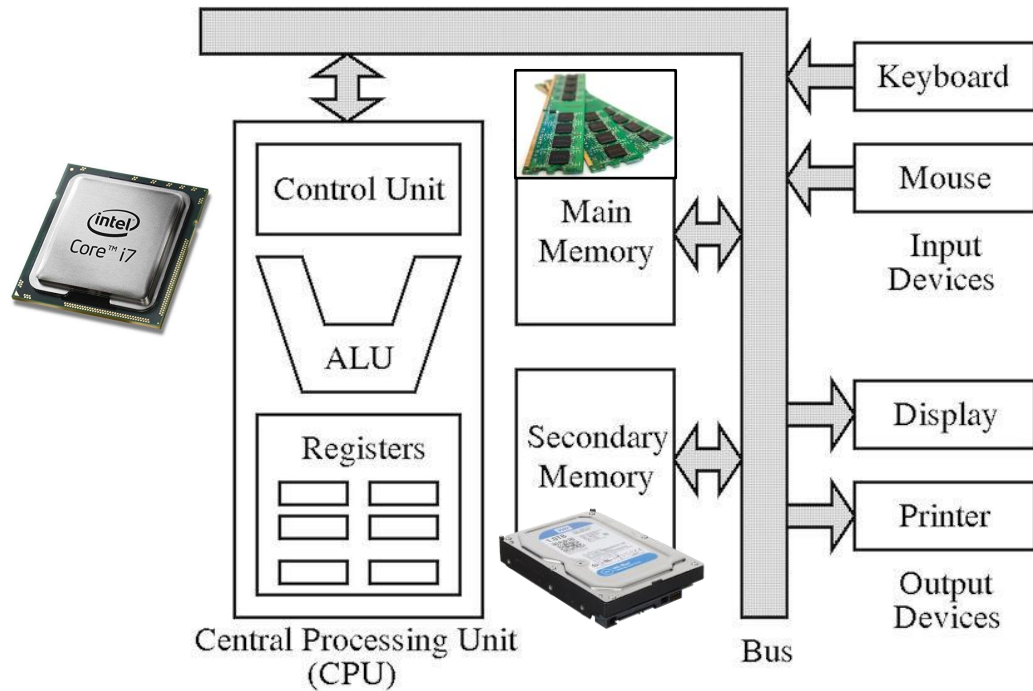
---

- Computer bottlenecks
- Linked VS Array data structures
- Searching & sorting intro



# Bottlenecks

Modern computers (Von-Neumann architecture) are full of bottlenecks

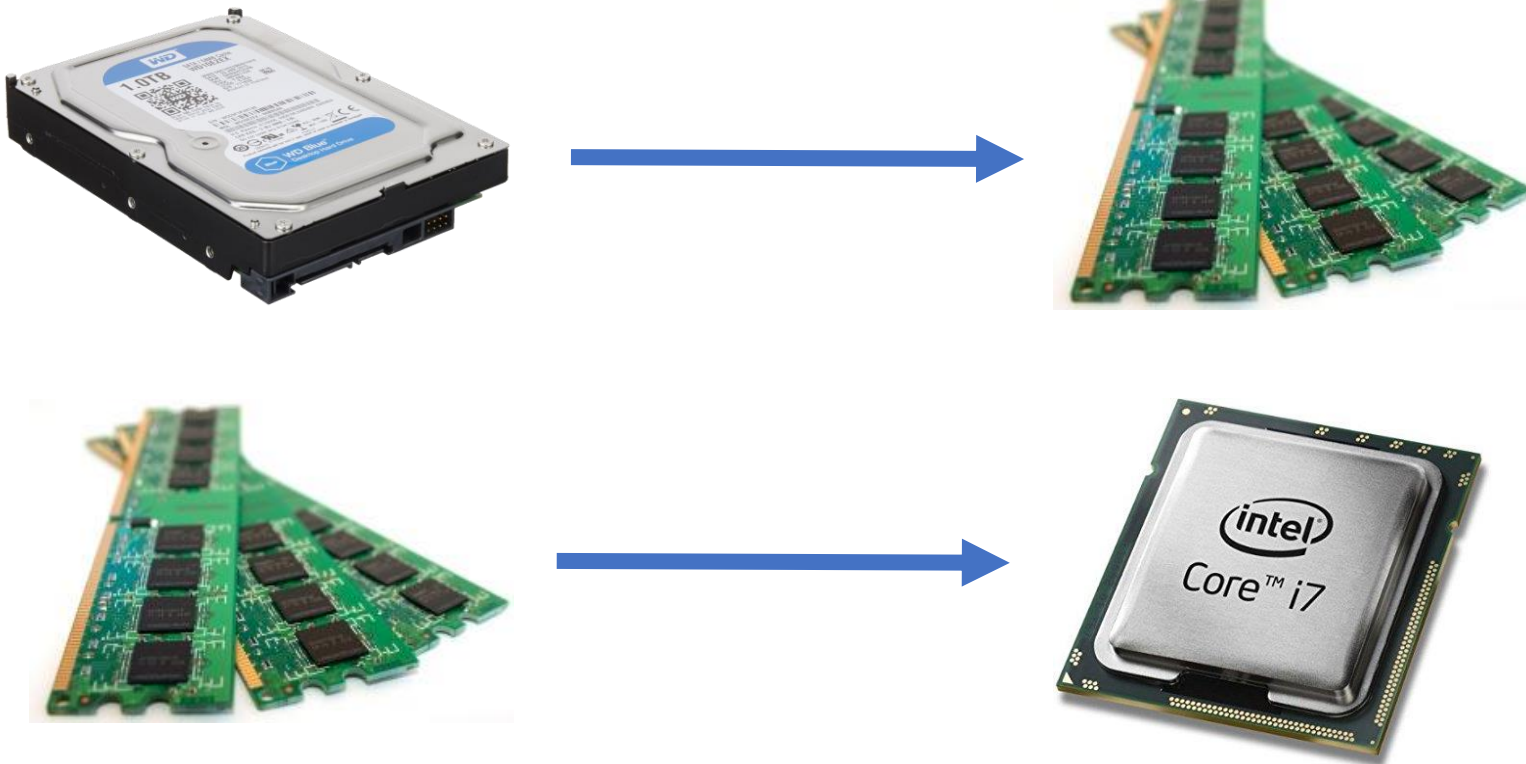


- The CPU operates on data in registers
- Data moves between secondary storage, main memory, and CPU cache/registers
- It travels along the system bus, and there are many bottlenecks along the way
- Choosing the correct data structure can mitigate these bottlenecks!
- Important even at the **hardware** level!

# Bottlenecks

---

Moving data between different levels of memory is the biggest bottleneck.

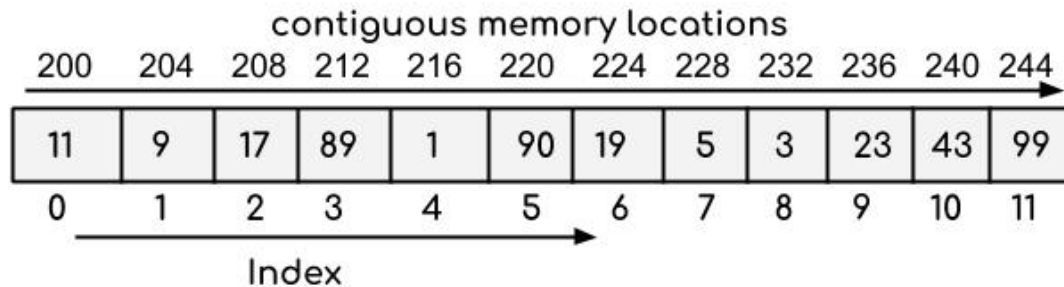


# Linked VS Array

Every Abstract Data Type (ADT) is built on one of two foundations:

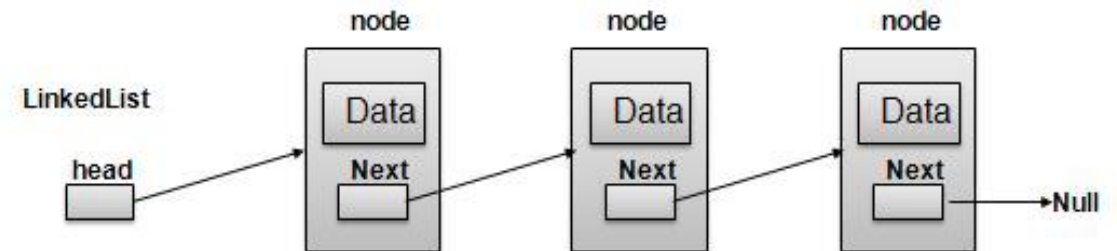
## Array:

- Elements are contiguous in memory
- $O(1)$  random access (index)
- $O(n)$  prepend, insertion

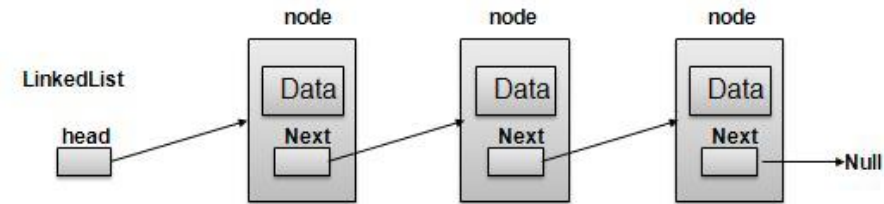
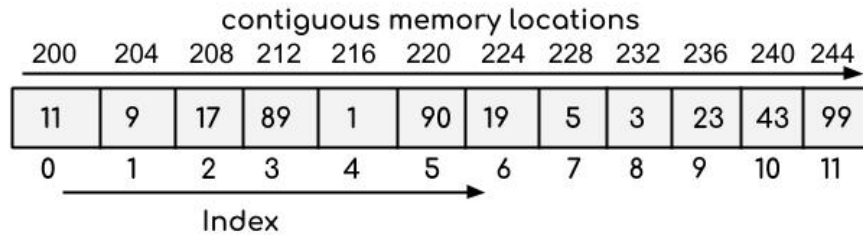


## Linked data structure:

- Elements, or nodes, are linked via pointers
- Not contiguous in memory!
- $O(1)$  prepend,  $O(n)$  indexing



# Linked VS Array

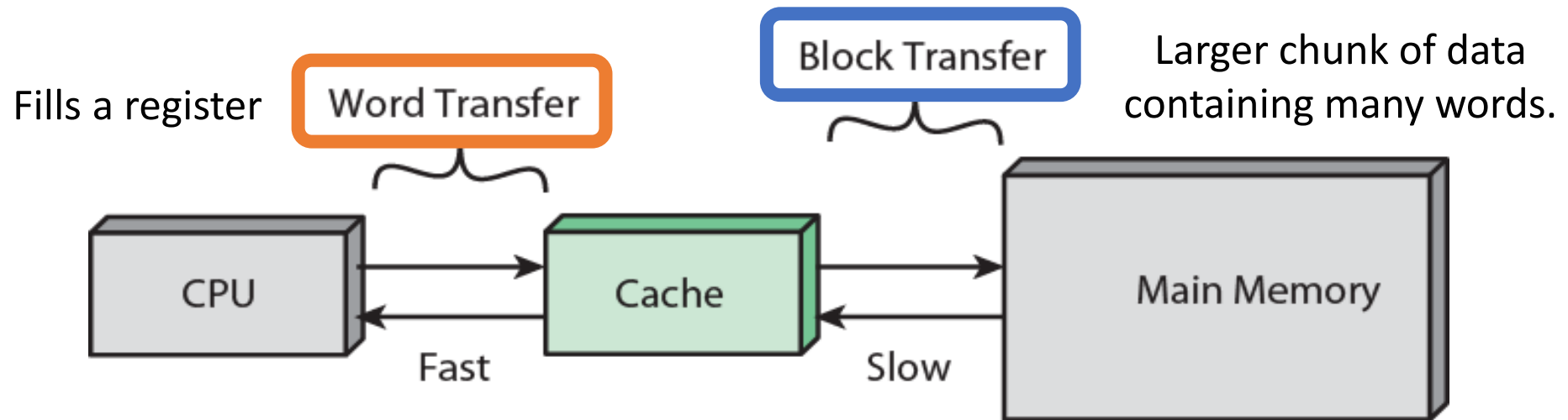


**Many ADTs can be implemented either way:**

- Binary tree? Can use an array or linked data structure.
- Expandable sequence? LinkedList or ArrayList (Java)
- Stack? Queue? Graph? Same thing.
- The properties of the data often guide the choice.

# CPS590 Sneak Peak: Cache Memory

- **Cache** memory serves as small, fast storage between registers and main memory. It is built into the CPU.
- First, CPU checks cache for what it needs. Found it? We're done!
- Not found? Check RAM, copy chunk of RAM into cache.
- The block copied into cache contains many surrounding addresses

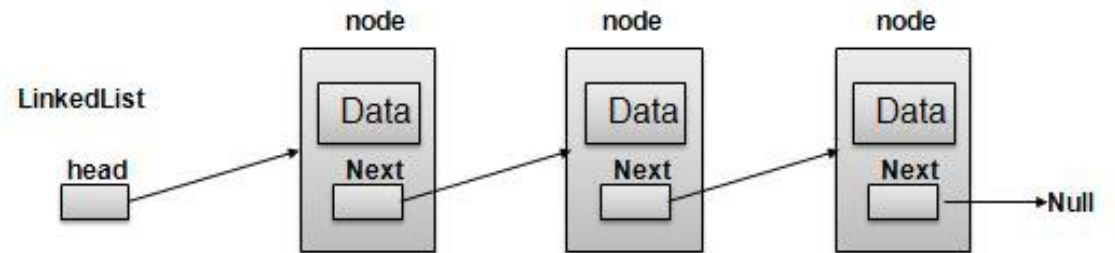
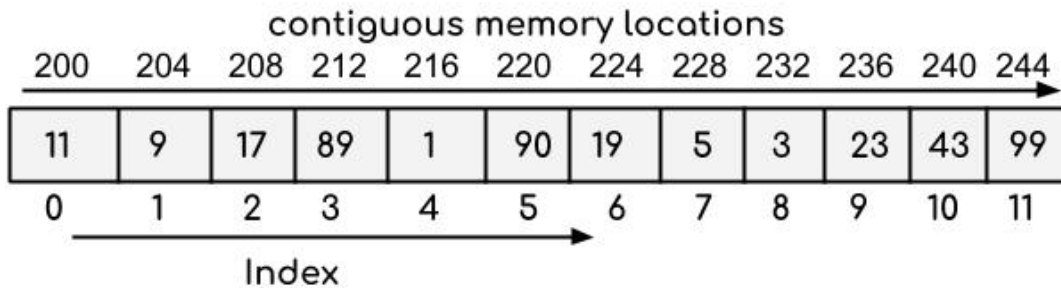


# Something to Ponder

Which data structure better lends itself to cache locality:

Linked Lists, or Arrays?

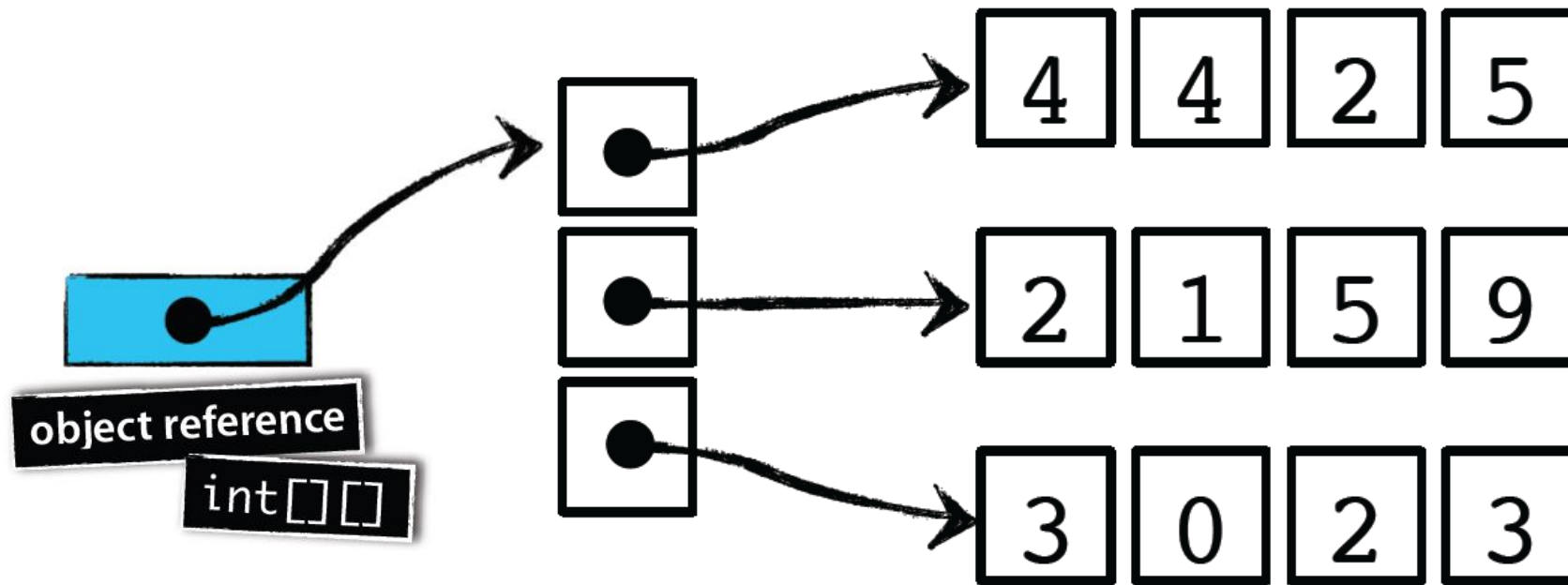
Arrays are contiguous in memory; linked lists are not.



# Talk is Cheap!

---

**Recall, of 2D arrays in Java:** A 2D array is an array of arrays, where each 1D array is a separate object.

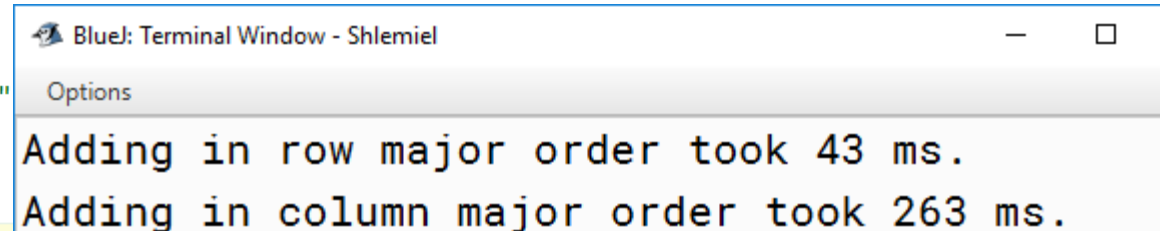


# Column VS Row Addition

```
public static void testMatrixAddOrders(int n) {  
    Random rng = new Random(12345);  
    int[][] matrix = new int[n][n];  
    for(int row = 0; row < n; row++) {  
        for(int col = 0; col < n; col++) {  
            matrix[row][col] = rng.nextInt(100);  
        }  
    }  
    long startTime = System.currentTimeMillis();  
    long r1 = addRowMajorOrder(matrix, n);  
    long endTime = System.currentTimeMillis();  
    System.out.println("Adding in row major order  
        (endTime - startTime) + " ms.");  
    startTime = System.currentTimeMillis();  
    long r2 = addColumnMajorOrder(matrix, n);  
    endTime = System.currentTimeMillis();  
    System.out.println("Adding in column major order  
        (endTime - startTime) + " ms.");  
    assert r1 == r2;  
}
```

- Create a huge 2D array
- Add elements by going across rows
- Add elements by going down columns.
- Compare!
- Going down columns means referencing a different object ***every single addition.***
- Poor cache locality (in theory)
- In practice, 5000x5000 array?

The bigger the 2D array, the more significant this gets.



Blue: Terminal Window - Shlemiel

Options

Adding in row major order took 43 ms.  
Adding in column major order took 263 ms.

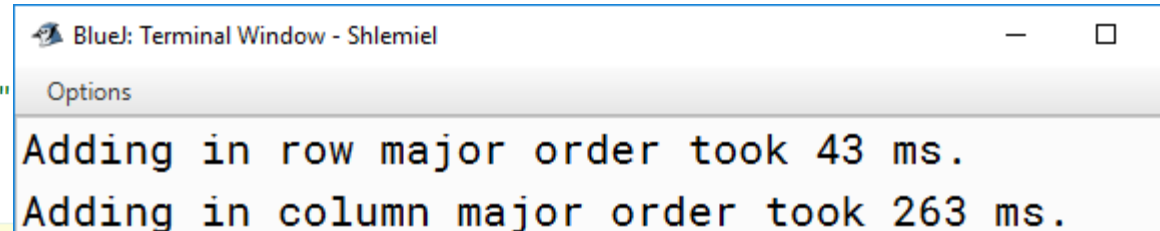


# Column VS Row Addition

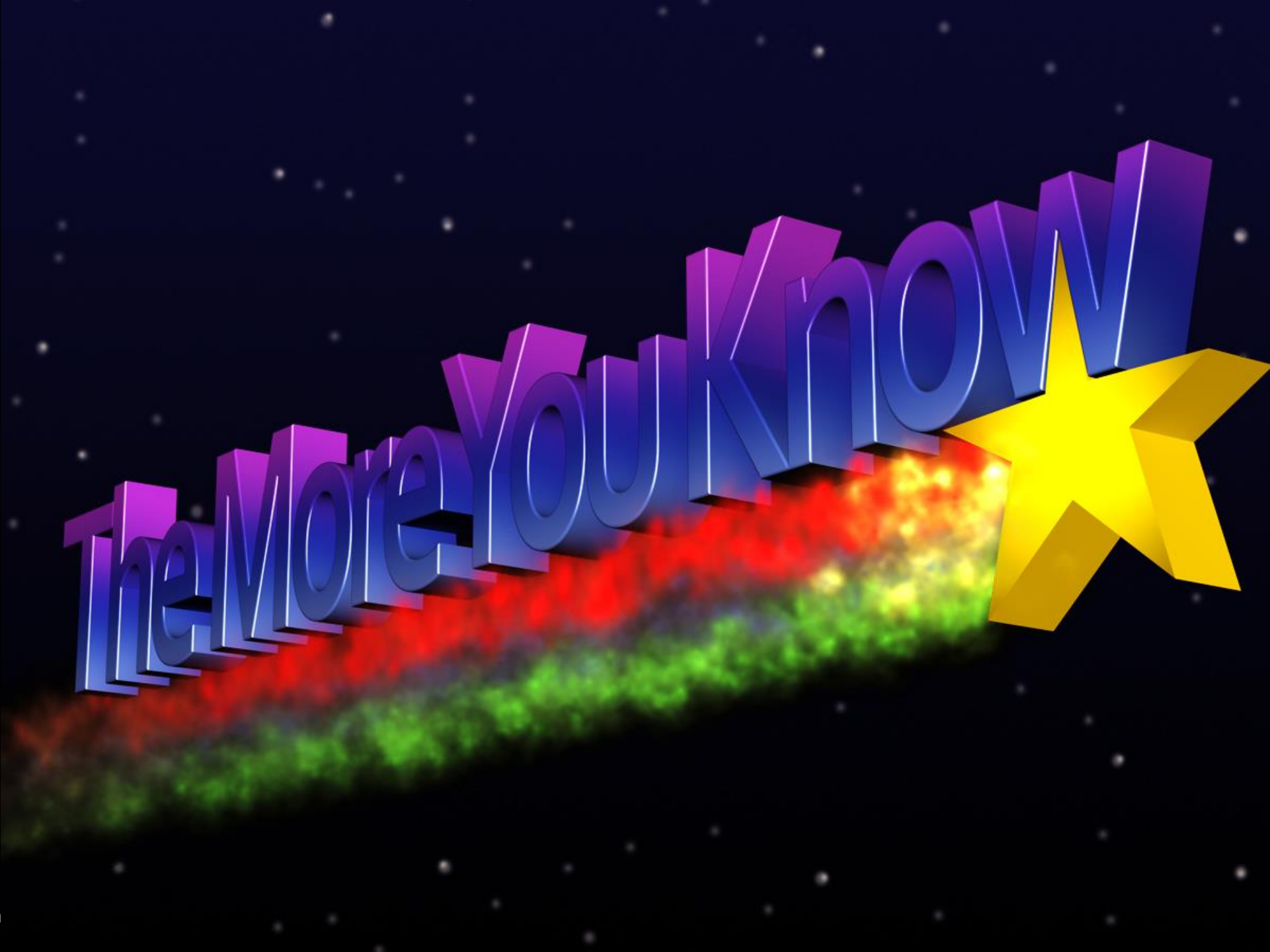
```
public static void testMatrixAddOrders(int n) {  
    Random rng = new Random(12345);  
    int[][] matrix = new int[n][n];  
    for(int row = 0; row < n; row++) {  
        for(int col = 0; col < n; col++) {  
            matrix[row][col] = rng.nextInt(100);  
        }  
    }  
    long startTime = System.currentTimeMillis();  
    long r1 = addRowMajorOrder(matrix, n);  
    long endTime = System.currentTimeMillis();  
    System.out.println("Adding in row major order took " +  
        (endTime - startTime) + " ms.");  
    startTime = System.currentTimeMillis();  
    long r2 = addColumnMajorOrder(matrix, n);  
    endTime = System.currentTimeMillis();  
    System.out.println("Adding in column major order took " +  
        (endTime - startTime) + " ms.");  
    assert r1 == r2;  
}
```

Taking a holistic view of the OS and the memory it manages can lead to surprising insights and optimizations.

The bigger the 2D array, the more significant this gets.

A screenshot of a terminal window titled "Blue: Terminal Window - Shlemiel". It shows the output of the Java program: "Adding in row major order took 43 ms." followed by "Adding in column major order took 263 ms." on the next line.

```
Blue: Terminal Window - Shlemiel  
Options  
Adding in row major order took 43 ms.  
Adding in column major order took 263 ms.
```



# Arrays in LISP

---

- This week, we will explore several searching and sorting algorithms.
- We'll be operating on arrays, not linked data.
- We'll introduce the basic contiguous array type in LISP and then discuss searching/sorting that type.
- Lists in LISP are linked, so we need something new.

# LISP Structures

---

We can define a sequence of elements (structure) in LISP using **defstruct**:

```
(defstruct movie title director year type)
```

When you create a struct, LISP creates the following for you automatically:

- Constructor: **MAKE-structureName** to create instances
- Accessors: **structureName-fieldname** to set/get fields

```
* (defvar *amovie* (make-movie))
*AMOVIE*
* *amovie*
#S(MOVIE :TITLE NIL :DIRECTOR NIL :YEAR NIL :TYPE NIL)
* (setf (movie-title *amovie*) "Lord of the Rings")
"Lord of the Rings"
* (setf (movie-director *amovie*) "Peter Jackson")
"Peter Jackson"
* (setf (movie-year *amovie*) 2001)
2001
* *amovie*
#S(MOVIE
  :TITLE "Lord of the Rings"
  :DIRECTOR "Peter Jackson"
  :YEAR 2001
  :TYPE NIL)
```

```
λ first-n-sum.lisp  λ functions.lisp  λ tuple.lisp ×
λ tuple.lisp > ...
1  (defstruct movie title director year type)
2
3  (print
4    (let ((amovie (make-movie :title "Blade Runner" :director "Ridley Scott")))
5      (setf (movie-year amovie) 1982)
6      amovie)
7  )
8
9
10
```

Refer to fields in constructor  
by name using : prefix

```
> sbcl --script tuple.lisp
#S(MOVIE :TITLE "Blade Runner" :DIRECTOR "Ridley Scott" :YEAR 1982 :TYPE NIL)
```

# LISP Arrays

---

LISP arrays are contiguous in memory:

```
* #(1 2 3 4 5)           ; Array literal form
#(1 2 3 4 5)
* (make-array 3)         ; Allocate array of size n
#(0 0 0)
* (make-array 3 :initial-element 1/2)
#(1/2 1/2 1/2)          ; Initialize elements
*
```

# LISP Arrays

---

Use **aref** to access elements, **setf** to mutate:

```
* (defvar *arr* #(1 2 3 4 5))
*ARR*
* (setf (aref *arr* 3) 42)
42
* *arr*
#(1 2 3 42 5)
```



# LISP Arrays: IMDB Example

Internet Movie DataBase (in LISP, and not on the internet)

```
(defstruct movie title director year)
(defparameter *size* 10) ;; Size of the db
(defvar *db*)
(setf *db* (make-array *size* :initial-element nil))

(defun add-movie (m)
  (dotimes (i *size*)
    (unless (aref *db* i)
      (setf (aref *db* i) m)
      (return t)
    )
  )
)
```

**defvar:** declare and initialize, doesn't reassign.

**defparameter:** declare and initialize, can also reassign.

# LISP Arrays: IMDB Example

---

Search for movie by title:

```
(defun in-db? (title)
  (dotimes (i *size*)
    (when (equal (movie-title (aref *db* i)) title)
      (return (aref *db* i))
    )
  ) )
```

```
* (load "imdb.lisp")
T
* (add-movie (make-movie :title "Blade Runner"))
T
* (add-movie (make-movie :title "Dune"))
T
* (add-movie (make-movie :title "Big Lebowski"))
T
* (in-db? "Blade Runner")
#S(MOVIE :TITLE "Blade Runner" :DIRECTOR NIL :YEAR NIL)
* (in-db? "Dune")
#S(MOVIE :TITLE "Dune" :DIRECTOR NIL :YEAR NIL)
```

# LISP Arrays: Expandable?

---

What if we don't know the size until runtime?

**LISP offers a dynamic vector:** Size expands automatically as needed

- Like an ArrayList in Java, List in Python, etc.
- Just have to create the array a little bit differently:

```
(make-array 0 :fill-pointer t :adjustable t)
```

Set initial size to zero

```
(let ((vec (make-array 0 :fill-pointer t :adjustable t)))  
  (dotimes (i 10)  
    (vector-push-extend i vec)  
    (describe vec)  
  )  
)
```

```
PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp e  
--script vector.lisp
```

```
• #(0)  
  [vector]
```

```
Element-type: T  
Fill-pointer: 1  
Size: 1  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 1) {2355BFB7}>  
#(0 1)  
[vector]
```

```
Element-type: T  
Fill-pointer: 2  
Size: 2  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 2) {235F2FDF}>  
#(0 1 2)  
[vector]
```

```
Element-type: T  
Fill-pointer: 3  
Size: 4  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 4) {235F6987}>  
#(0 1 2 3)  
[vector]
```

```
Element-type: T  
Fill-pointer: 4  
Size: 4  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 4) {235F6987}>  
#(0 1 2 3 4)
```

```
Element-type: T  
Fill-pointer: 6  
Size: 8  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 8) {235FAC37}>  
#(0 1 2 3 4 5 6)  
[vector]
```

```
Element-type: T  
Fill-pointer: 7  
Size: 8  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 8) {235FAC37}>  
#(0 1 2 3 4 5 6 7)  
[vector]
```

```
Element-type: T  
Fill-pointer: 8  
Size: 8  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 8) {235FAC37}>  
#(0 1 2 3 4 5 6 7 8)  
[vector]
```

```
Element-type: T  
Fill-pointer: 9  
Size: 16  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 16) {23603087}>  
#(0 1 2 3 4 5 6 7 8 9)  
[vector]
```

```
Element-type: T  
Fill-pointer: 10  
Size: 16  
Adjustable: yes  
Displaced: no  
Storage vector: #<(SIMPLE-VECTOR 16) {23603087}>
```

Once we can allocate arrays and mutate their elements,  
we can implement search and sort algorithms over them.

# Searching



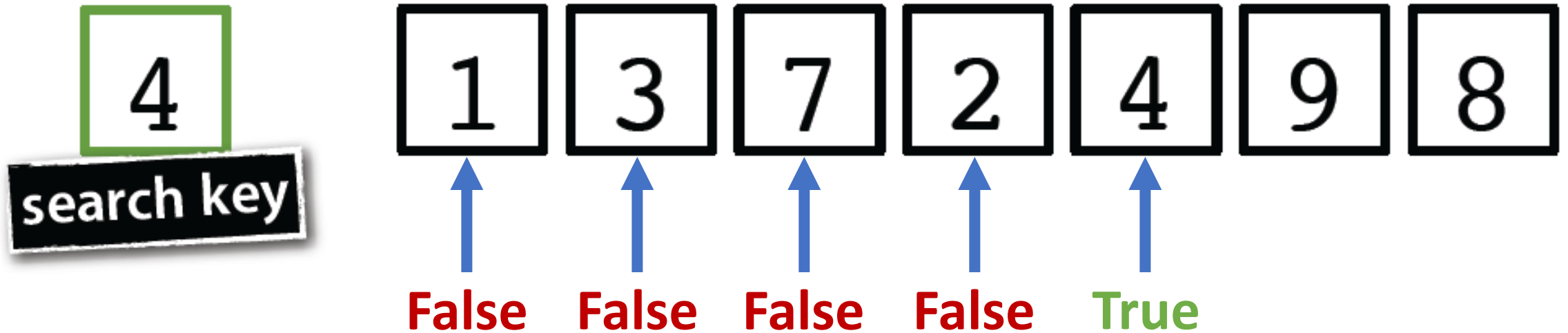
# Linear Search

---

- Have array, need index of target value.
- Used on ***unsorted*** arrays.
- Far better options exist for sorted arrays.
- There are a few different flavors, though none do better than linear time.
- Linear time? One loop, roughly speaking.



**Linear Search:** Check every element, one by one, until we find the item we're looking for or hit the end of the list.



`(= key (aref items i))`

# In LISP?

---

Easy, using  
**dotimes:**

```
(defun lin-search (arr key)
  (dotimes (i (length arr))
    (if (= (aref arr i) key) (return i) )
  ) )
```

```
* (lin-search #(1 3 5 8 6 3 6 4 9 0) 5)
2
* (lin-search #(1 3 5 8 6 3 6 4 9 0) 55)
NIL
* (lin-search #(1 3 5 8 6 3 6 4 9 0) 0)
9
* (lin-search #(1 3 5 8 6 3 6 4 9 0) 3)
1
* (lin-search #(1 3 5 8 6 3 6 4 9 0) 4)
7
```

For *linear search*, it doesn't matter what the order of the array is.

At worst, we're required to check every single element.

If we know the array is sorted, is it possible to improve the efficiency of searching?

If you're looking for a word in the dictionary (assume we're living in the mid 90s), how do you do it?

Do you start with the first word on the first page and check every single word until you find it?

**Aristotle**

**Einstein**

**Mendeleyev**

**Bohr**

**Faraday**

**Morley**

**Brahe**

**Galileo**

**Michelson**

**Cavendish**

**Galton**

**Newton**

**Copernicus**

**Hooke**

**Pauling**

**Curie**

**Laplace**

Find *Laplace* in the array

Aristotle

Einstein

Mendeleyev

Bohr

Faraday

Morley

Brahe

Galileo

Michelson

Cavendish

Galton

Newton

Copernicus

Hooke

Pauling

Curie

Laplace

Check the element in the middle. If it's less than the key, we can discard it, and ***everything that comes before it.***

	Mendeleyev
	Morley
	Michelson
Galton	Newton
Hooke	Pauling
Laplace	

Check the element in the middle. If it's greater than the key, we can discard it, and ***everything that comes after it.***

Galton

Hooke

Laplace

Repeat until we land on the key, or the array is empty.



**Laplace**

Repeat until we land on the key, or the array is empty.

# Binary Search

---



Three checks!

**Binary Search:** Progressively consider half the array until the search key is found or there are no more elements to consider.

Requires a sorted array!

# In LISP?

---

Algorithm:

If the number is smaller than the middle number:

- search in the left half

If the number is greater than the middle number:

- search in the right half

If the number is equal to the middle number:

- found, return index

Else:

- not found

# In LISP?

---

**do** form more appropriate, since index is updated manually (not incremented)

```
(defun bin-search (arr key)
  (do ( (i nil) (lb 0) (ub (1- (length arr))) )
      ((> lb ub) nil)
      (setf i (floor (+ lb ub) 2))
      (let ((e (aref arr i)))
        (cond ((< e key) (setf lb (1+ i)))
              ((> e key) (setf ub (1- i)))
              (t (return i))
        )
      )
  ) ) )
```

λ search.lisp ×

λ search.lisp > ...

```
1 (defun lin-search (arr key)
2   (dotimes (i (length arr))
3     (if (= (aref arr i) key) (return i) ))
4   )
5 )
6
7 (defun bin-search (arr key)
8   (do ( (i nil) (lb 0) (ub (1- (length arr))) )
9     ((> lb ub) nil)
10    (setf i (floor (+ lb ub) 2))
11    (let ((e (aref arr i)))
12      (cond ((< e key) (setf lb (1+ i)))
13            ((> e key) (setf ub (1- i)))
14            (t (return i)))
15    )
16  )
17 )
18 )
19
20
```

```
* (bin-search #(1 2 4 5 7 9 10 12 18) 4)
2
* (bin-search #(1 2 4 5 7 9 10 12 18) 18)
8
* (bin-search #(1 2 4 5 7 9 10 12 18) 1)
0
* (bin-search #(1 2 4 5 7 9 10 12 18) 9)
5
* (bin-search #(1 2 4 5 7 9 10 12 18) 10)
6
* (bin-search #(1 2 4 5 7 9 10 12 18) 5)
3
* (bin-search #(1 2 4 5 7 9 10 12 18) 11)
NIL
* (bin-search #(1 2 4 5 7 9 10 12 18) 6)
NIL
```

# Binary VS Linear

---

## Things to ponder:

- In the **best** case, both linear and binary searches find the key on the first comparison.
- In the **worst** case, linear search checks every element
- How many comparisons does binary search make in the worst case?
- Our intuition would say it's fewer than linear search
- Linear search reduces search space by one value
- Binary search cuts the search space **in half**

## Number of comparisons for a list of size $n$ ?

1 + number of comparisons for a list of size  $n/2$

$$T(n) = 1 + T(n/2)$$

$$T(n/2) = 1 + T(n/4)$$

$$T(n/4) = 1 + T(n/8)$$

⋮

$$T(n/\underline{2^m}) = 1 \longrightarrow \text{Continue unwrapping until } n/2^m == 1$$

- List is split  $m$  times
- Total comparisons =  $1 + m$
- How do  $m$  and  $n$  relate?



$T(\underline{n/2^m}) = 1 \longrightarrow$  Continue unwrapping until  $n/2^m == 1$

- List is subdivided  $m$  times
- Total comparisons =  $1 + m$
- How do  $m$  and  $n$  relate?

$$n/2^m = 1$$

$$n = 2^m$$

$$m = \log_2 n$$

- Binary search is  $O(\log n)$
- Linear search is  $O(n)$
- For every possible value of  $n$ ,  $\log n \leq n$

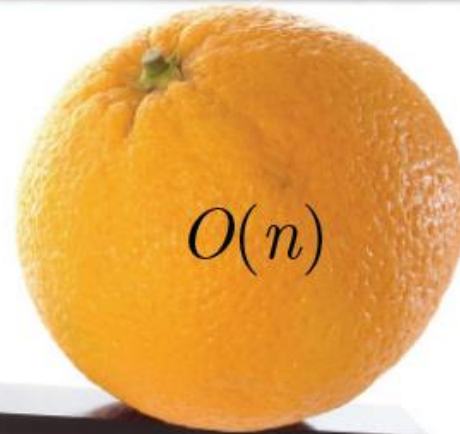
# Can we do better?

binary search



$O(\log(n))$

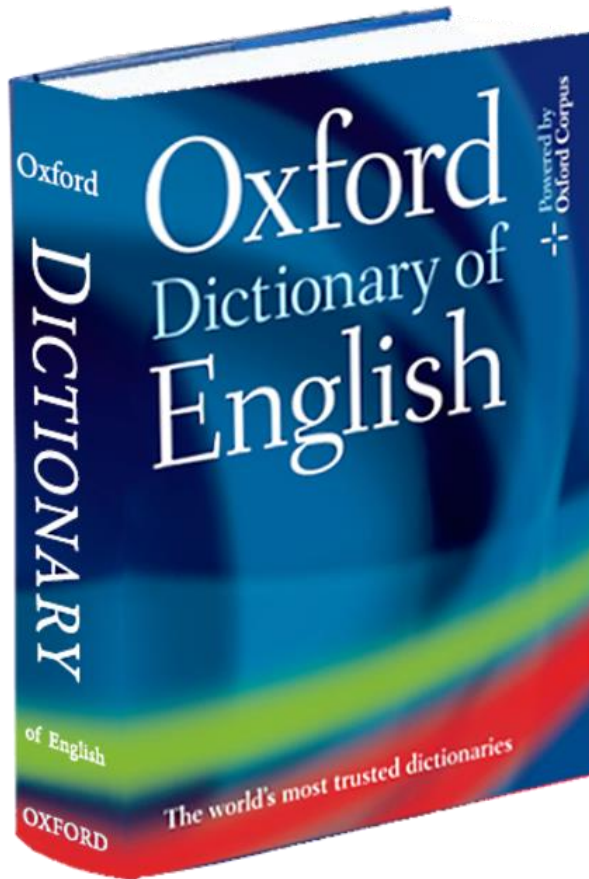
linear search



$O(n)$

# Interpolation Search

---



We don't open the dictionary at the midpoint (or at all, realistically) if our word starts with B.

We open it closer to the beginning.

This works because we know that, with a few exceptions, each letter contains a similar number of words.

# Interpolation Search

---

**Consider:** A sorted list with **min = 0** and **max = 1,000,000**

If we're searching for the value 919,856, does it make sense to split the list in half, a-la binary search?

If we *also* know the list is uniformly distributed, it doesn't. That is, the numbers are roughly evenly spread out between 0 and 1,000,000

*(As opposed to 95% of the values being below 100,000 and only 5% being above)*

# Interpolation Search

---

Thus, in these cases, we can use some form of linear interpolation to select the next index rather than picking the midpoint.

It simply involves adjusting the index update statement:

$$\text{mid} = (\text{hi} + \text{lo}) / 2;$$

*Might become:*

$$\text{mid} = \text{lo} + (\text{hi} - \text{lo}) * (\text{x} - \text{a}[\text{lo}]) / (\text{a}[\text{hi}] - \text{a}[\text{lo}])$$

# Interpolation Search

---

Allows us to cut away much larger portions of the (sorted) list at a time

However, it requires additional knowledge about the elements in the list.

Can work with other distributions as well, though computing the next index will become increasingly complicated.



# Sorting

# Sorting

---

- Sorting algorithms take an unsorted sequence of values and produce a sorted sequence of values.
- “Sorted” does not necessarily mean numerically ascending.
- Could be descending, or some (any) other criteria.
- Remember **compareTo()** in Java, sort criteria is up to you!



Why sort in the first place?

Unsorted phone books would  
be useless! Case closed.

# Sorting is something we all do:



This person is clearly a monster. Not sorted alphabetically **OR** by point value.



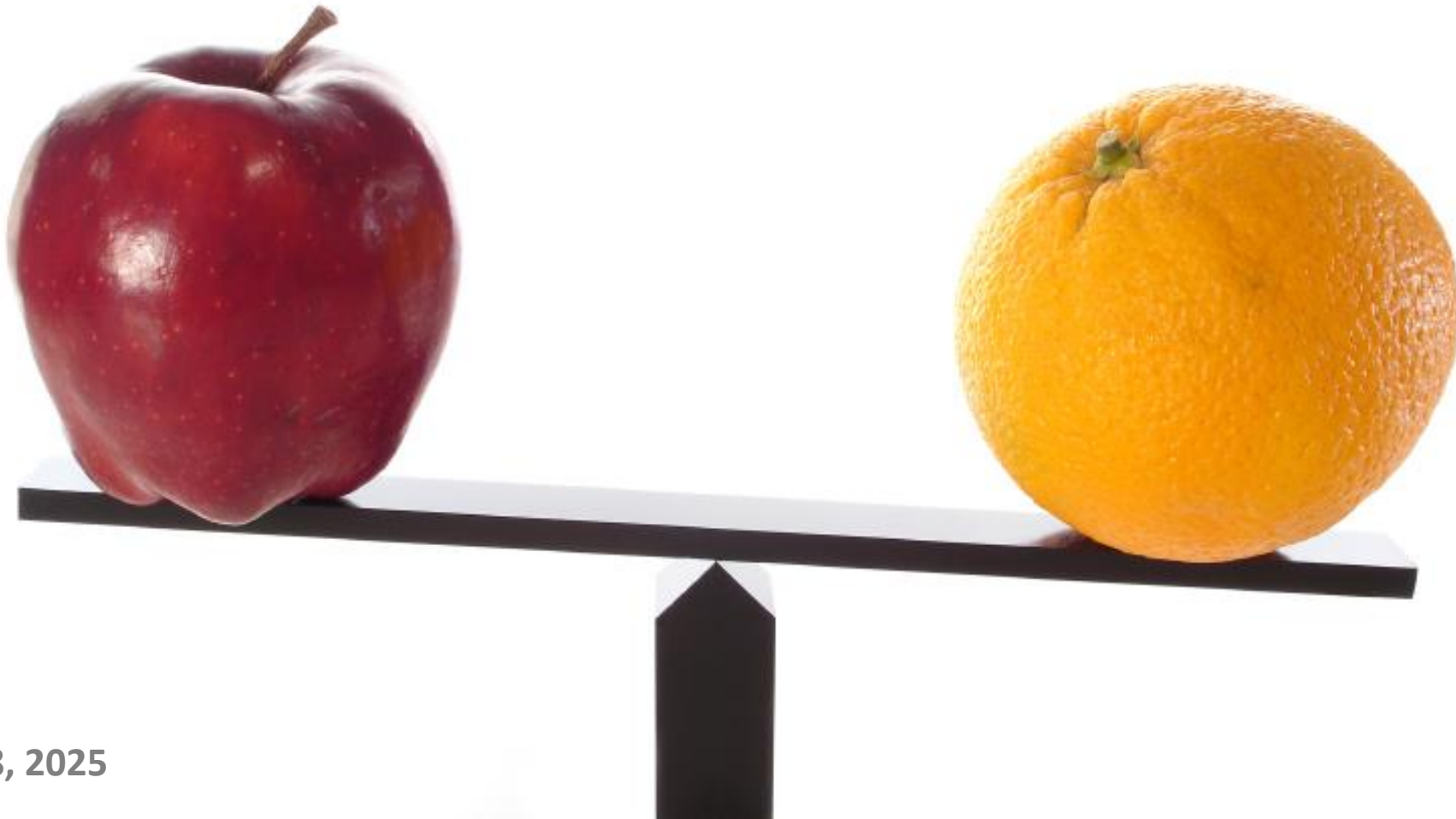
Sorting forms the foundation for faster algorithms

**Very often:**

Working on an *unsorted* list is more expensive than sorting the list first and **THEN** working on it.



# Comparison Sorting



# Comparison Sorting

---

A *comparison sort* is a sorting algorithm that uses some comparison operation (which can itself contain many individual comparisons) to determine which of two elements should come first in the sorted list.

## Requirements:

1. If  $a \leq b$ , and  $b \leq c$ , then  $a \leq c$
  2. For all  $a$  and  $b$ , either  $a \leq b$  or  $b \leq a$
- It's possible, of course, that  $a \leq b$  AND  $b \leq a$
  - In this case, it doesn't matter which element comes first
    - (with caveats – stable VS. unstable sorting).

# Comparison Sorting: A Metaphor

---



## **Balance scale + set of unsorted weights**

- With no other information, use the scale to sort the weights.
- We need not say anything meaningful about the elements being sorted, aside from how they compare to each other.
- This is the essence of comparison sorting.

## Examples [\[ edit \]](#)

Some of the most well-known comparison sorts include:

- Quicksort
- Heapsort
- Shellsort
- Merge sort
- Introsort
- Insertion sort
- Selection sort
- Bubble sort
- Odd–even sort
- Cocktail shaker sort
- Cycle sort
- Merge insertion (Ford–Johnson) sort
- Smoothsort
- Timsort

- There are many comparison sorts out there.
- We'll see 4 or 5 in total.
- Why do we need more than one in the first place?

## Performance limits and advantages of different sorting techniques [\[ edit \]](#)

# Sorting

---

## **Time complexity VS implementation complexity**

- Simple  $O(n^2)$  algorithms VS complex  $O(n \log n)$  algorithms

## **In-place VS requiring extra memory**

- Sorting *in-place* means we do not require any helper arrays

## **Stable VS unstable:**

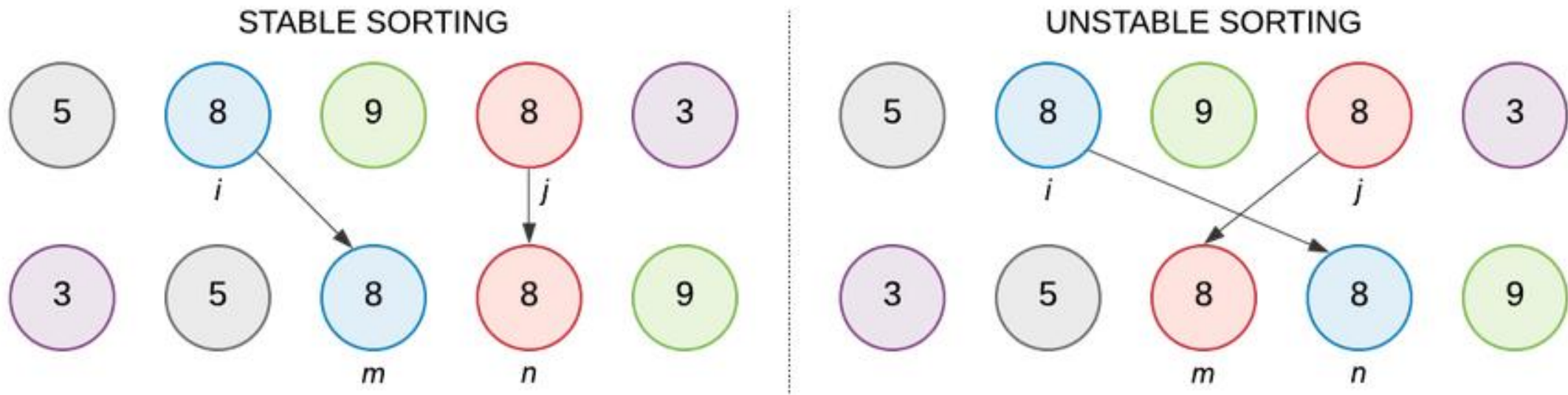
- Stable sort: Relative position of equal elements doesn't change



# Stable Sorting

## Stable VS unstable:

- Stable sort: Relative position of equal elements doesn't change



# Stable Sorting

---

2	8	1	0	5	6	5	9	7	4
---	---	---	---	---	---	---	---	---	---

- In the sorted list, these values will not have changed order.
- Useless for primitives, but VERY useful for objects. We can have multiple sub-orderings.
- Sort students based on course first, section second, last name third.
- If we begin by sorting by last name, and then re-sort based on section, the last names will still be in order *within each section*

### sorted by time

Chicago	09:00:00
Phoenix	09:00:03
Houston	09:00:13
Chicago	09:00:59
Houston	09:01:10
Chicago	09:03:13
Seattle	09:10:11
Seattle	09:10:25
Phoenix	09:14:25
Chicago	09:19:32
Chicago	09:19:46
Chicago	09:21:05
Seattle	09:22:43
Seattle	09:22:54
Chicago	09:25:52
Chicago	09:35:21
Seattle	09:36:14
Phoenix	09:37:44

### sorted by location (not stable)

Chicago	09:25:52
Chicago	09:03:13
Chicago	09:21:05
Chicago	09:19:46
Chicago	09:19:32
Chicago	09:00:00
Chicago	09:35:21
Chicago	09:00:59
Houston	09:01:10
Houston	09:00:13
Phoenix	09:37:44
Phoenix	09:00:03
Phoenix	09:14:25
Seattle	09:10:25
Seattle	09:36:14
Seattle	09:22:43
Seattle	09:10:11
Seattle	09:22:54

*no  
longer  
sorted  
by time*

### sorted by location (stable)

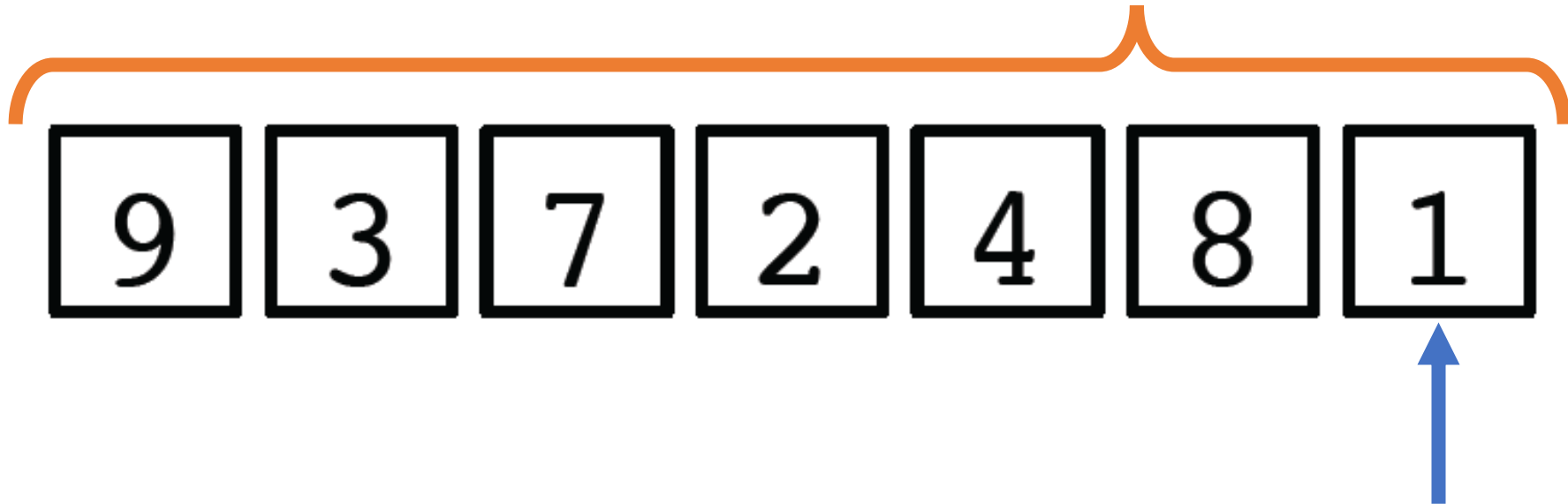
Chicago	09:00:00
Chicago	09:00:59
Chicago	09:03:13
Chicago	09:19:32
Chicago	09:19:46
Chicago	09:21:05
Chicago	09:25:52
Chicago	09:35:21
Houston	09:00:13
Houston	09:01:10
Phoenix	09:00:03
Phoenix	09:14:25
Phoenix	09:37:44
Seattle	09:10:11
Seattle	09:10:25
Seattle	09:22:43
Seattle	09:22:54
Seattle	09:36:14

*still  
sorted  
by time*

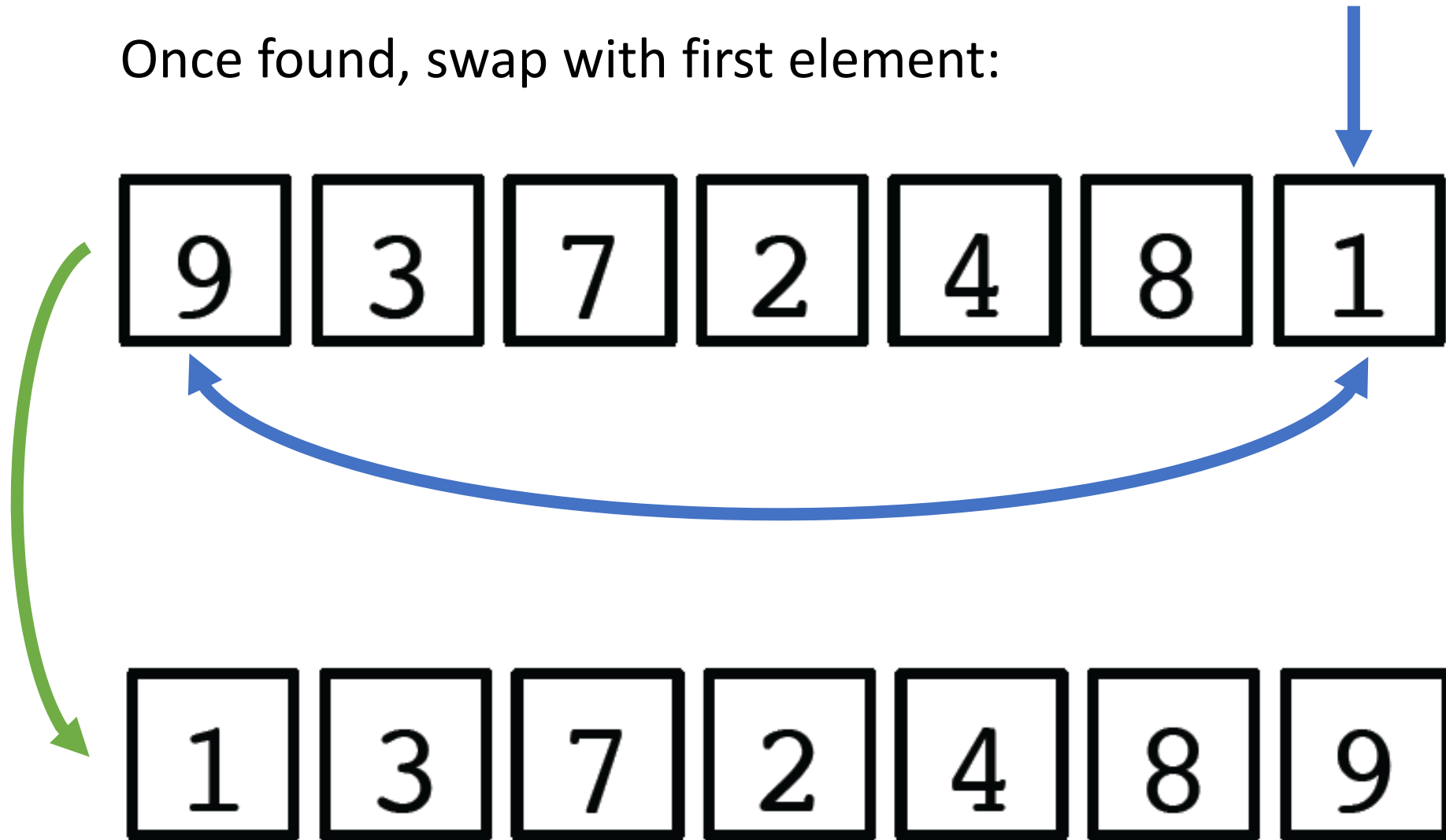
# Selection Sort

---

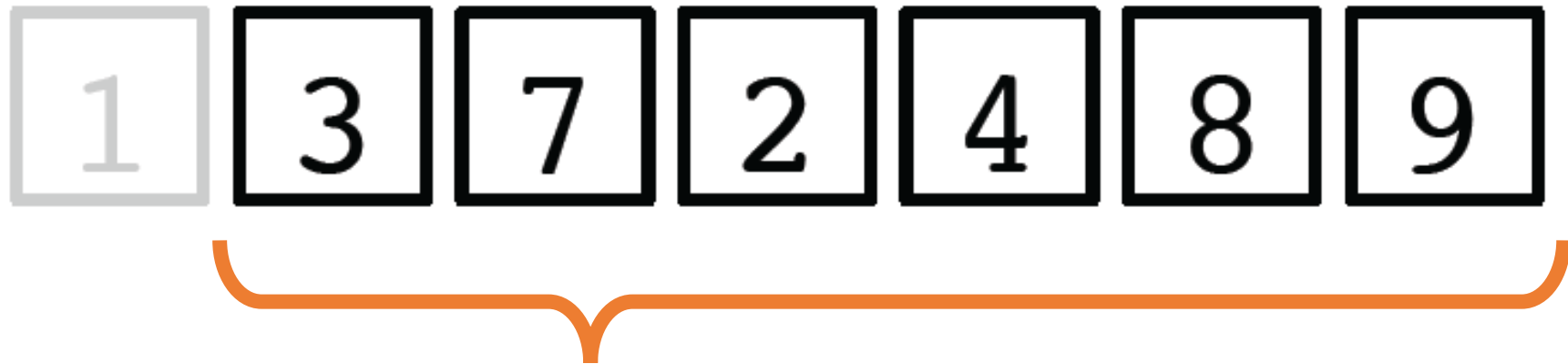
**Selection Sort:** Repeatedly find smallest element and move it to the front of the *unsorted* region



Once found, swap with first element:

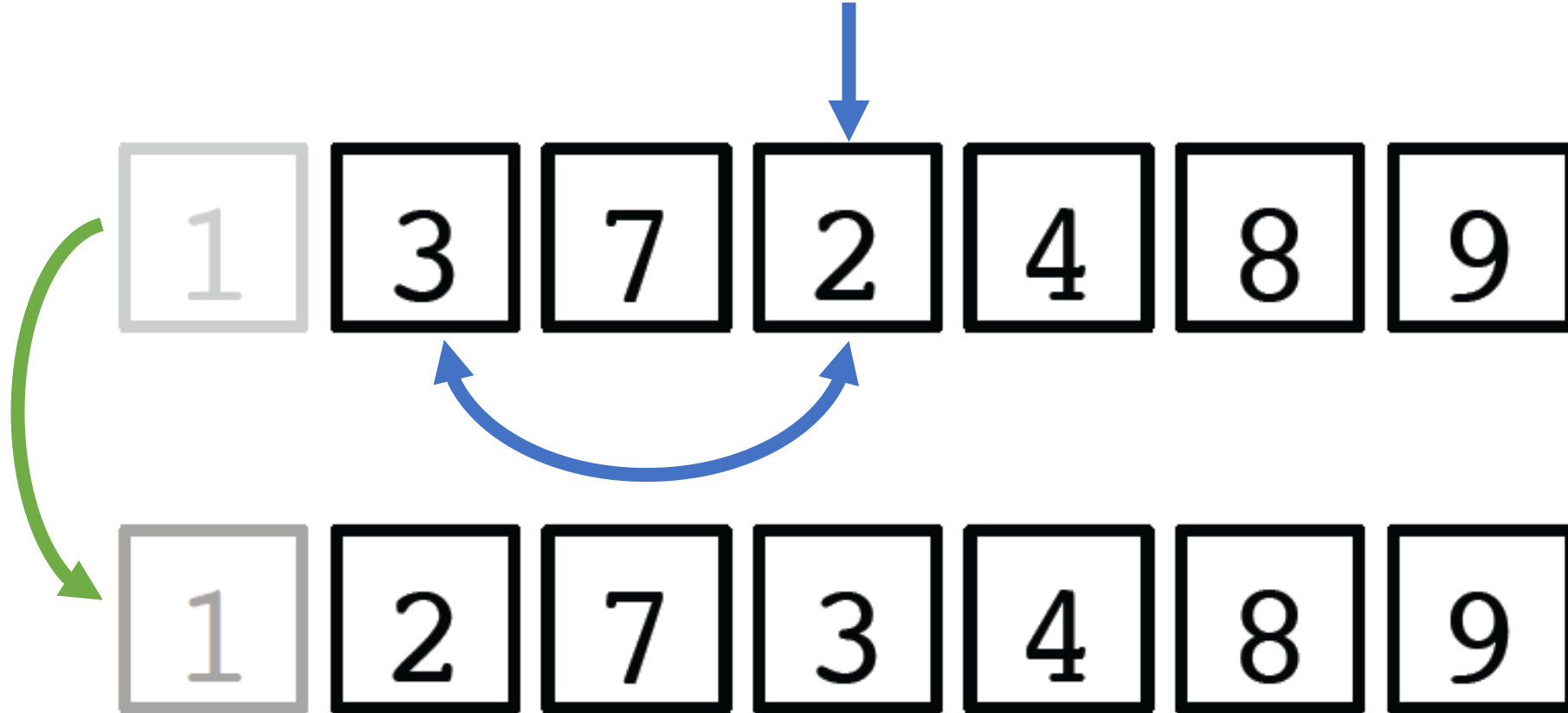


We can now be **certain** that the first element is in the correct location:

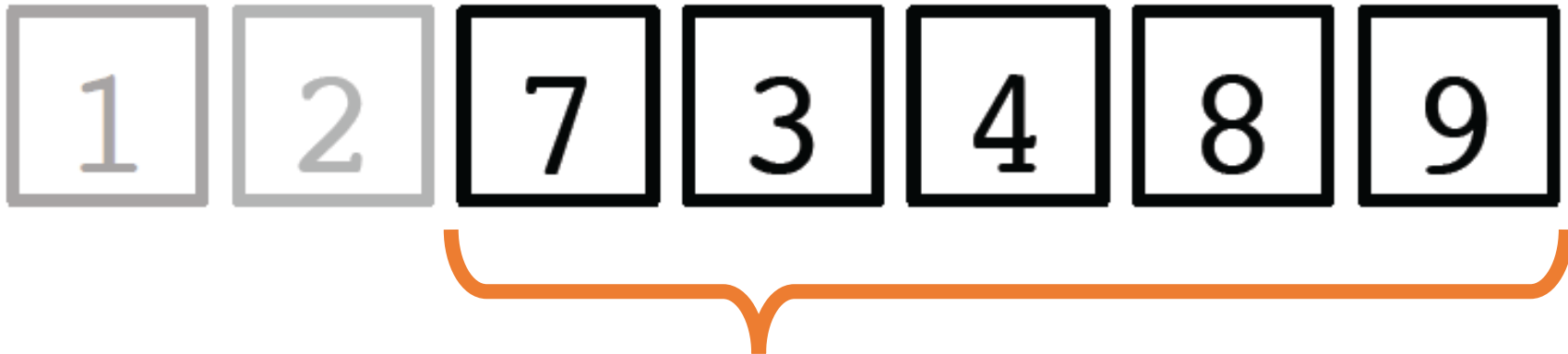


**New unsorted region!**

Once more, find the smallest element in the *unsorted* region, swap with element at the front of *unsorted* region:

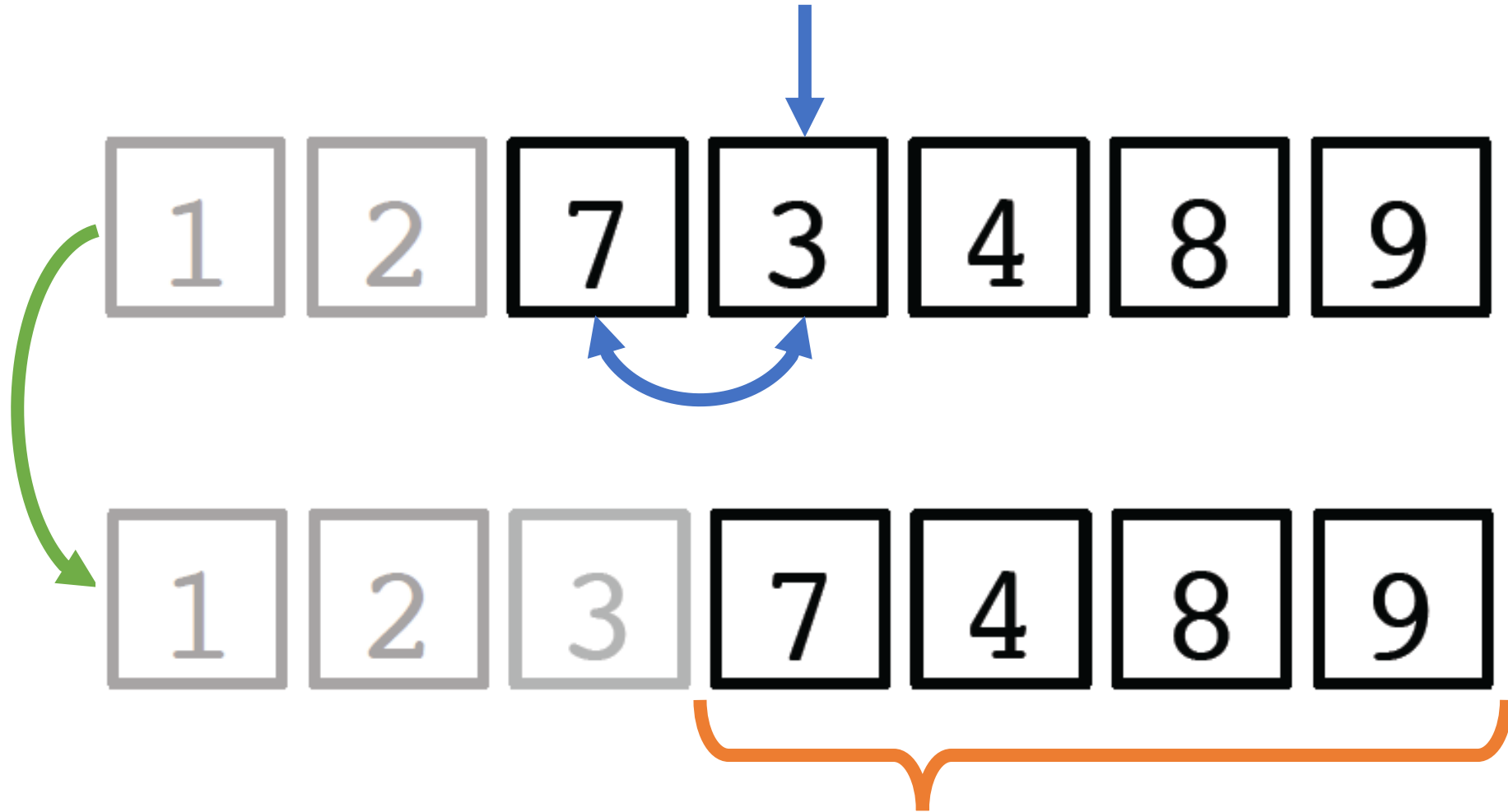


We can now be **certain** that the first TWO element are in the correct location:

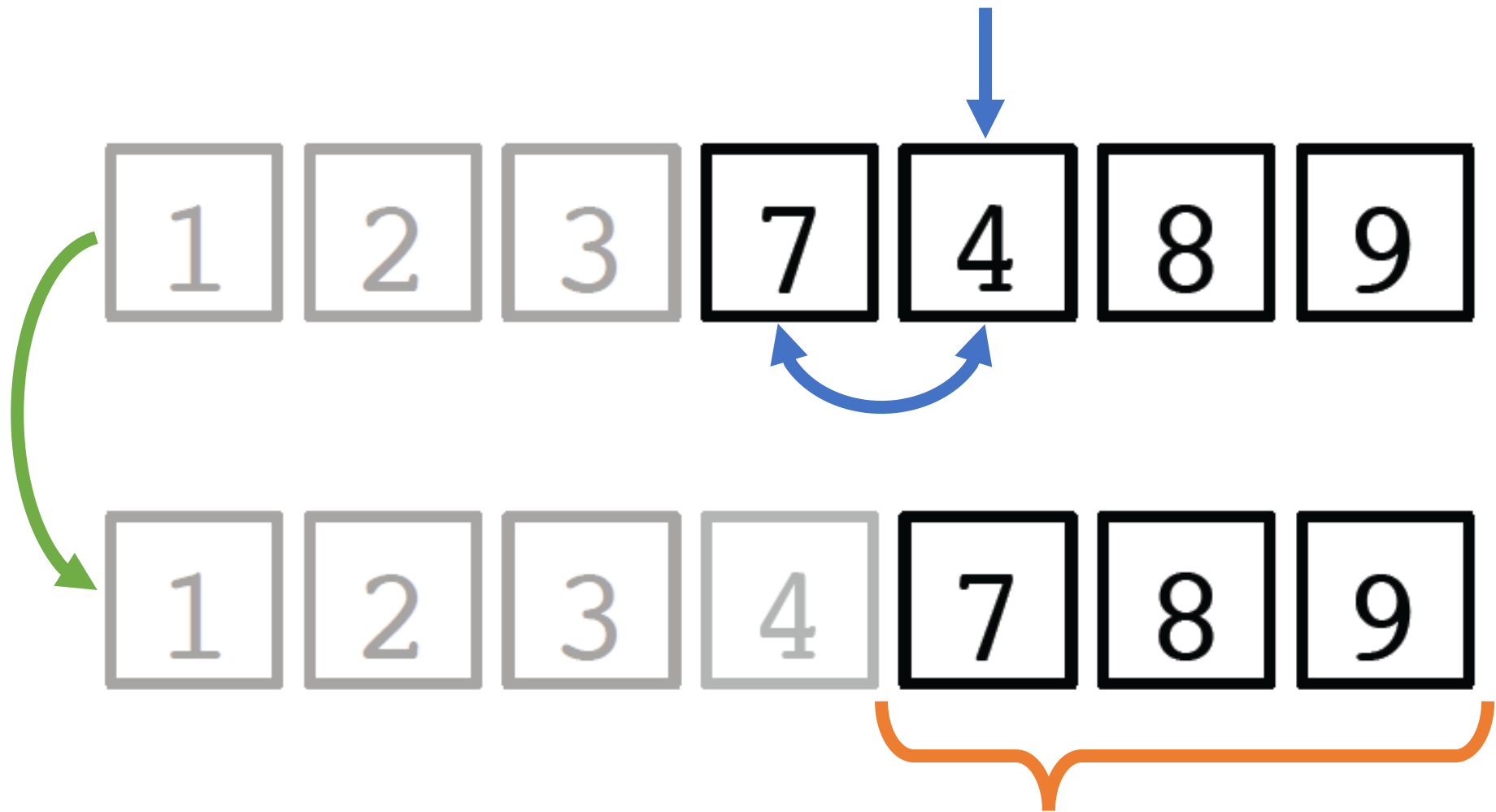


**New unsorted region!**





**New unsorted region!**

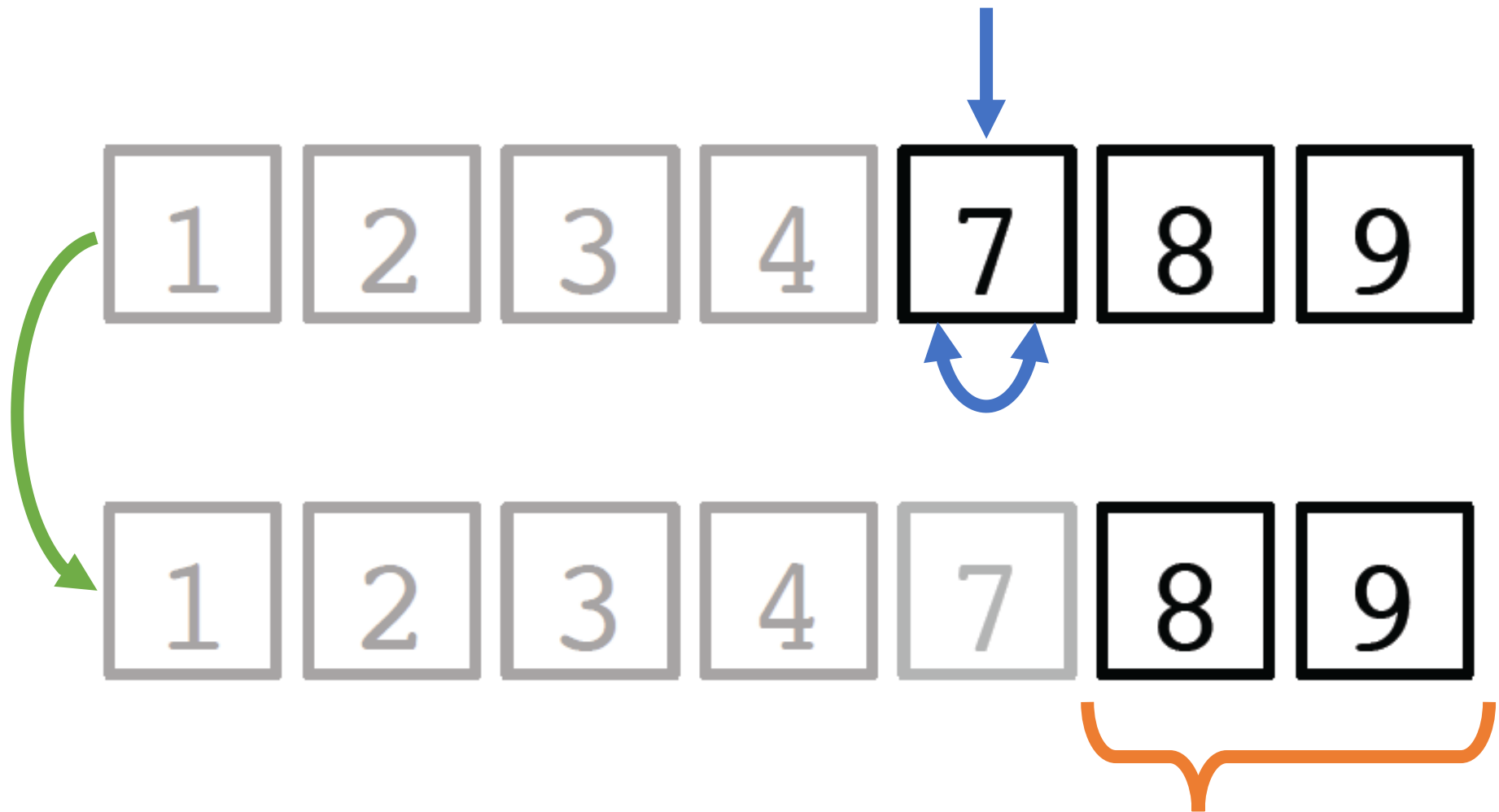


**New unsorted region!**

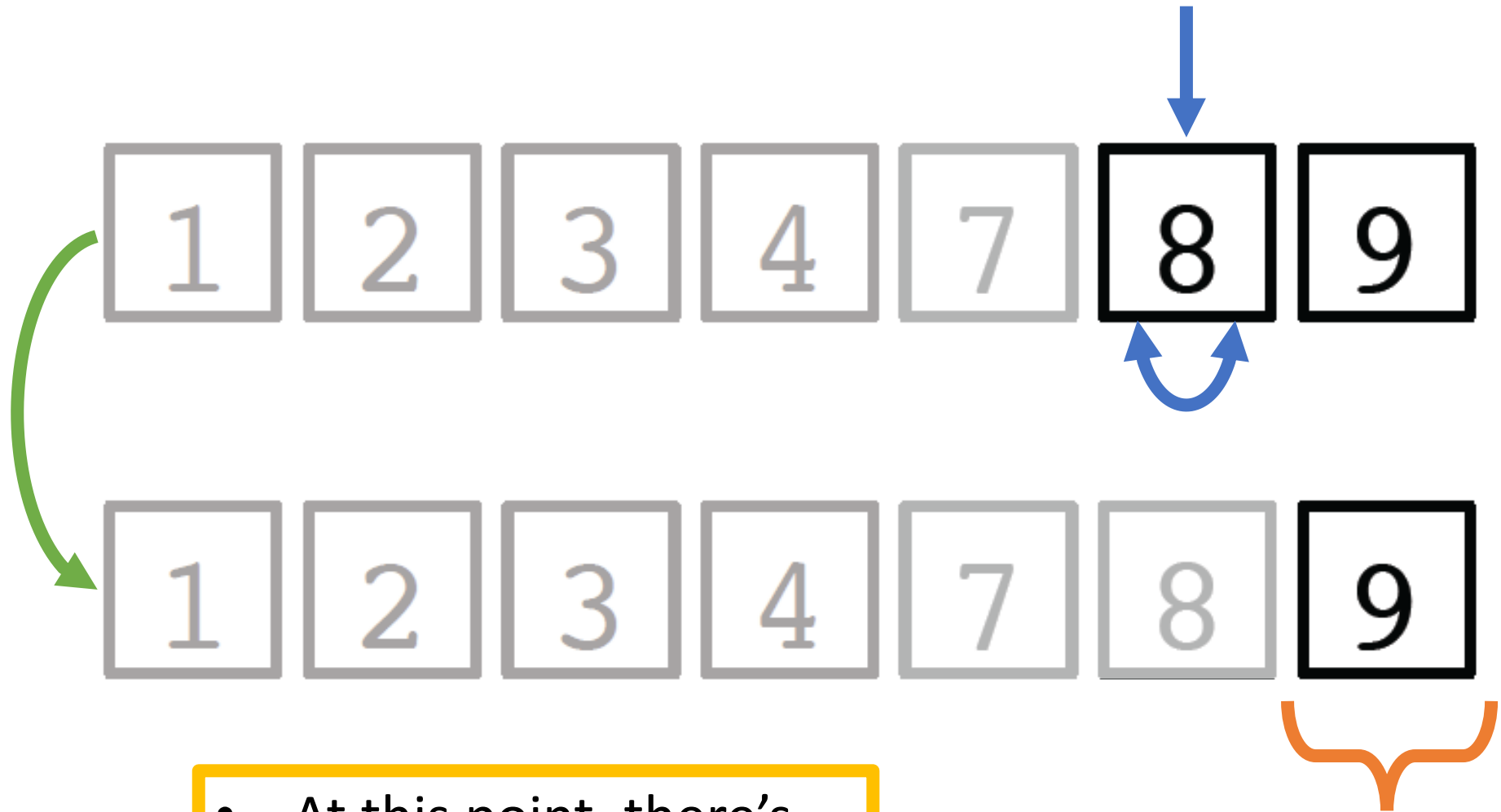


The list is sorted at this point, but the algorithm ***doesn't know that.***

We continue this process until the entire list has been compared and swapped in this manner.



**New unsorted region!**



- At this point, there's one element left.
- We know we're done.

**New unsorted region!**

# Selection Sort

---

It's **BAD**

- It's good as "*my first sorting algorithm*"
- Bad for sorting in an efficient manner
- Performance is identical in best-case and worst-case scenarios.
- Even if the list is ***already sorted***, selection sort takes just as long to perform.
- **Why?** Finding minimum value is ***always***  $O(n)$

```

(defun selection-sort (vec comp)
  (dotimes (cur (1- (length vec)))
    (let ((best (aref vec cur))
          (idx cur))
      (do ((j (1+ cur) (1+ j)))
          ((> j (1- (length vec))))
        (when (funcall comp (aref vec j) best)
          (setf best (aref vec j)
                idx j)
        )
      )
      (rotatef (aref vec cur) (aref vec idx))
    )
  )
  vec
)

```

Iterate through every element except the last

Variables to store index of "best" element and current index

Find the index of "best" element. Note relationship between **j** and **cur**!

Once found, swap "best" element with front of *unsorted* region

Return vec (now sorted) at the end

```
* (defvar a (make-array 6 :initial-contents (list 3 1 0 7 8 2)))
```

A

```
* a
```

```
#(3 1 0 7 8 2)
```

```
* (defvar b (make-array 6 :initial-contents '(3 1 0 7 8 2)))
```

B

```
* b
```

```
#(3 1 0 7 8 2)
```

```
* (selection-sort a #'<)
```

```
#(0 1 2 3 7 8)
```

```
* (selection-sort b #'>)
```

```
#(8 7 3 2 1 0)
```

```
*
```

```
(defun selection-sort (vec comp)
  (dotimes (cur (1- (length vec)))
    (let ((best (aref vec cur))
          (idx cur))
      (do ((j (1+ cur) (1+ j)))
          ((> j (1- (length vec))))
        (when (funcall comp (aref vec j) best)
          (setf best (aref vec j)
                idx j)
          )
        )
      (rotatef (aref vec cur) (aref vec idx))
      )
    )
  )
  vec
)
```



# Selection Sort: Time Complexity

---

**Selection Sort:** Repeatedly find smallest element and move it to the front of the *unsorted* region

Finding the smallest element in a list of size  $n$  requires  $n-1$  comparisons:



With selection sort, we must find the minimum many times.



N-1 comparisons



N-2 comparisons

⋮

⋮



2 comparisons



1 comparison

**Selection Sort:** Repeatedly find smallest element and move it to the front of the *unsorted* region

**Total comparisons:**  $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$

**This is an arithmetic series!**

$$1 + 2 + 3 + 4 + \dots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}$$

**We're only going up to n-1:**

$$1 + 2 + 3 + 4 + 5 + \dots + (n-2) + (n-1) = \frac{(n-1)(n)}{2}$$

**We're only going up to n-1:**

$$\begin{aligned} 1 + 2 + 3 + 4 + 5 + \dots + (n-2) + (n-1) &= \frac{(n-1)(n)}{2} \\ &= \frac{n^2 - n}{2} \end{aligned}$$

**Big-O notation:**

- Locate fastest growing term
- Ignore constant coefficient

$$= \frac{1}{2}n^2 - \frac{1}{2}n$$

### Big-O notation:

- • Locate fastest growing term
- • Ignore constant coefficient

$$\frac{1}{2}n^2 - \frac{1}{2}n$$

Selection sort is Big-O  $n^2$

Or just  $O(n^2)$

# Selection Sort

---

It's **BAD**

- Even among  $O(n^2)$  sorting algorithms, it's bad.
- We'll see a much more attractive  $O(n^2)$  algorithm next class, and then move on to  $O(n \log n)$  sorting algorithms.

# In Summary

---

## **Searching & Sorting intro:**

- Linear search, binary search
- Comparison sorting
- Selection sort

