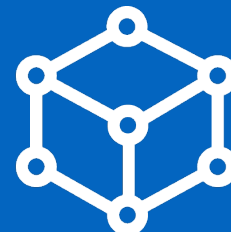
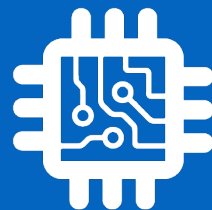
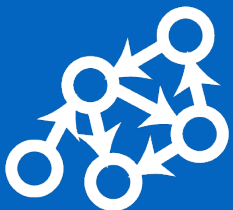


CSI 34 Lecture:

Measuring Efficiency



Announcements & Logistics

- **HW 10** due tonight @ 10 pm
 - (___str___ from class should have been included for Q11)
- **Lab 9 Boggle (Parts 1 & 2)** due Wed/Thurs at 10 pm
 - Make sure your completed game satisfies all of the expected behavior mentioned in handout
 - Test your game thoroughly!
 - Not just "normal game behavior"
 - Stress test it with unexpected clicks, etc

Do You Have Any Questions?

Last Time: Linked Lists

- More discussion about linked lists
- Made linked list type iterable (we can use it in our for-each loops)
 - implemented **`__iter__`** and **`__next__`** methods
- Talked briefly about efficiency in linked lists and arrays
 - prepend (add-to-front)
 - insert (add-to-middle)
 - append (add-to-end)

Today and Next Class

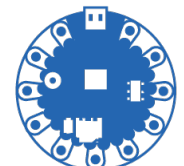
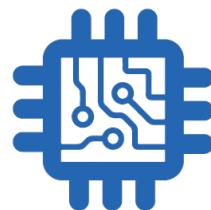
- Start a more principled discussion of efficiency in computer science
 - What is it we are trying to capture?
 - Explore some classic trade-offs surrounding certain operations in data structure and algorithm design
- Discuss efficiency of some classic algorithms
 - Linear search
 - Binary search



Linked List Efficiency

- How can we compare the efficiency of the following **LinkedList** operations?
 - **append** an item at the end of a LinkedList
 - **prepend** an item to the beginning of a LinkedList
- Any thoughts on which is "faster" (without defining efficiency formally)
 - **append** needs to traverse the entire list to find last item
 - "number of steps" is proportional to number of items
 - **prepend** just needs to change self._rest of newly inserted item
 - this is independent of how many items are in the **LinkedList**
- This is intuitively why **prepend** is more efficient than **append**
- For more formal discussion: need to figure out what we want to measure

Measuring Efficiency



Measuring Efficiency



- How do we measure the efficiency of our program?
 - We want programs that run "fast"
 - How should we measure this?
- One idea: use a stopwatch to see how long it takes
 - "Wall clock time" seems like a *reasonable* proxy for efficiency
 - But, what is it *actually* measuring?
- Suppose I run the exact same program on a really slow/old computer vs a really powerful supercomputer
 - Stopwatch will measure different times!
 - **Question:** Are we measuring how fast our program is or how fast the computer executes it?

Measuring Efficiency



- How do we measure the efficiency of our program?
 - We want programs that run "fast"
 - How should we measure this?
- One idea: use a stopwatch to see how long it takes
 - Reality: stop-watch approach measures how long a piece of code takes **on this machine on this particular input**
 - Both machine and input dependent
- We want to isolate our ***program's efficiency***
 - How well does it **scale** to larger inputs?
 - How does it **compare to other solutions to the same** problem: we want a concrete tool to say whether one solution is better

Efficiency Metric: Goals

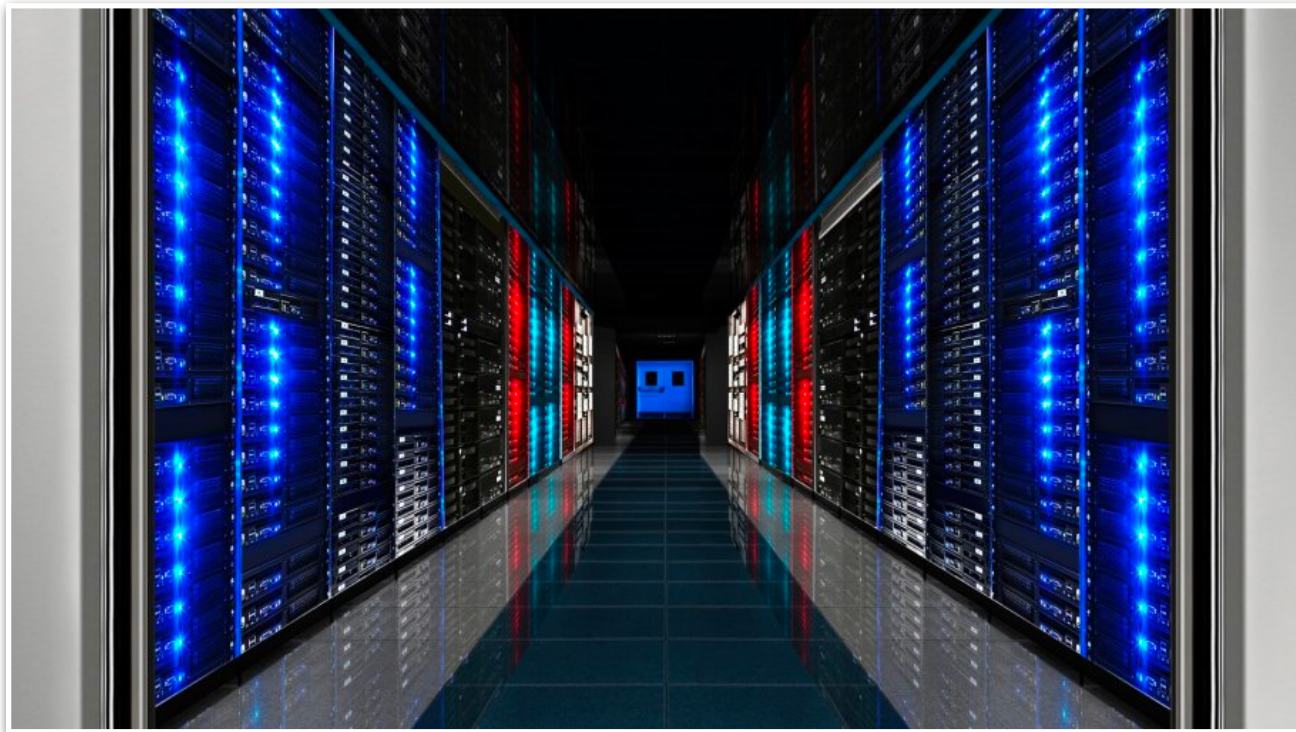
We want a method to evaluate efficiency that:

- **Is machine and language independent**
 - Analyzes the **algorithm's** (i.e., the problem-solving approach) efficiency
- **Provides guarantees that hold for different types of inputs**
 - Some values may be "easy" to work with while others are not
- **Captures the dependence on input size**
 - Determine how the performance "scales" when the input gets bigger
- **Captures the right level of specificity**
 - We don't want to be too specific (cumbersome)
 - Measure things that matter, ignore what doesn't

Platform/Language Independent

Machine and language independence

- We want to evaluate how good the algorithm is, rather than how good the machine or implementation is
- Basic idea: Count the **number of steps** taken by the algorithm
- Sometimes referred to as the "running time"

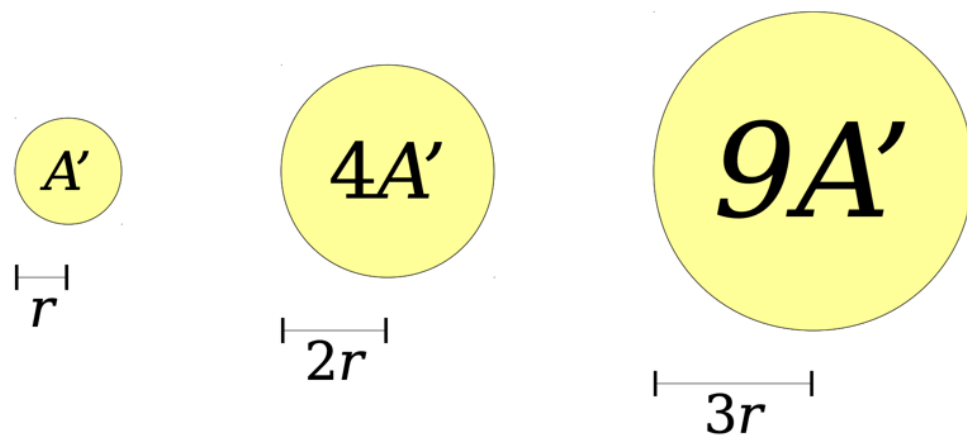


Worst-Case Analysis

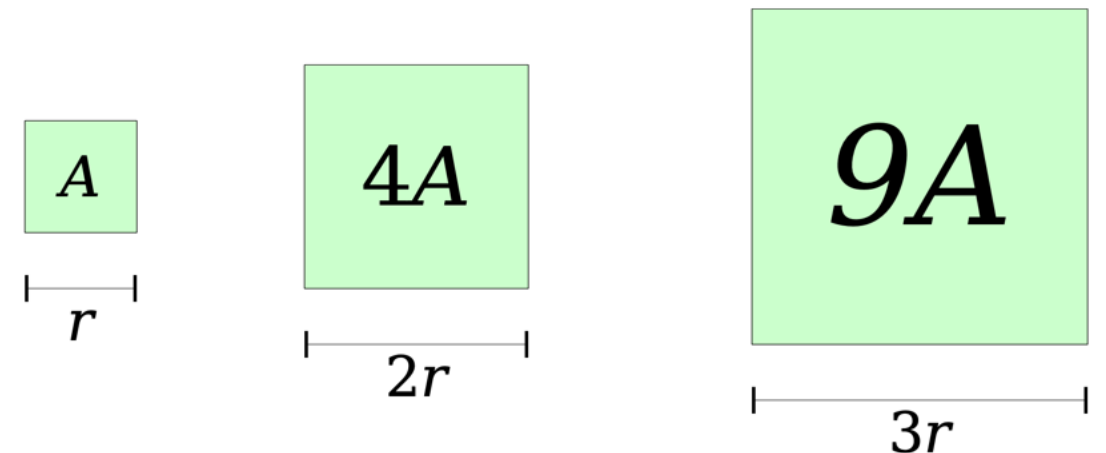
- We can't just analyze our algorithm on a few inputs and declare victory. Which of the following is the scenario we should be measuring?
- **Best case.** Minimum number of steps taken over all possible inputs of a given size
- **Average case.** Average number of steps taken over all possible inputs of a given size
- **Worst case.** Maximum number of steps taken over all possible inputs of a given size.
- Benefits of worst case analysis:
 - Regardless of input size, we can conclude that the algorithm always does *at least as well as* the pessimistic analysis

Dependence on Input Size

- We generally don't care about performance on "small inputs"
- Instead we care about "the rate at which the completion time grows" with respect to the input size
- For example, consider the area of a square or circle: while the formula for each is different, they both grow at the same rate wrt radius
 - doubling radius increases area by 4x, tripling increases by 9x



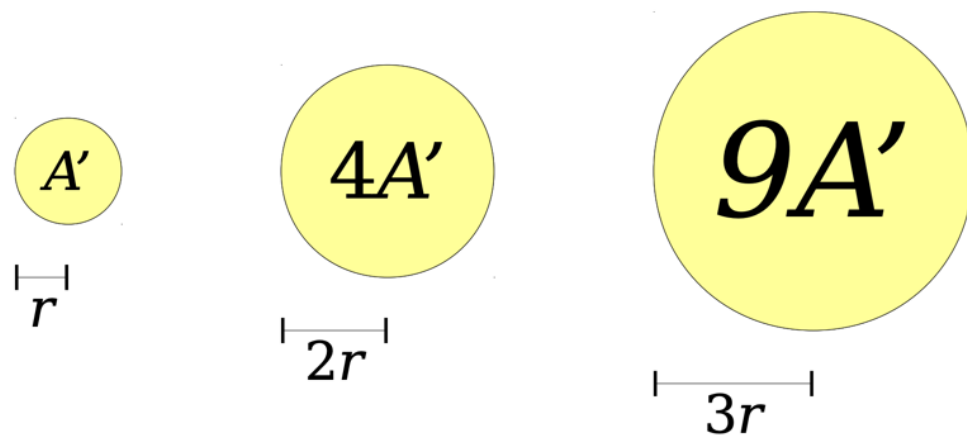
Doubling r increases area 4x.
Tripling r increases area 9x.



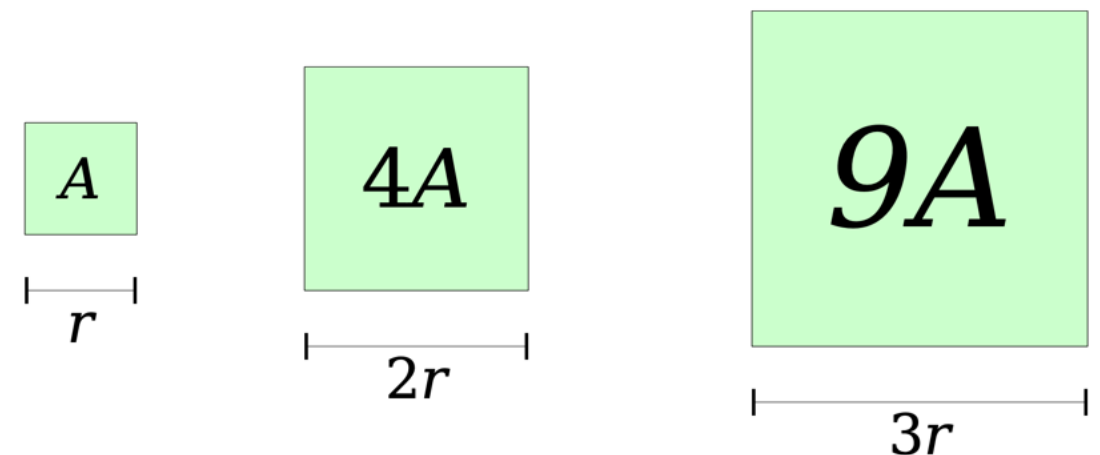
Doubling r increases area 4x.
Tripling r increases area 9x.

Dependence on Input Size: Big-O

- Big-O notation in Computer Science is a way of quantifying (in fact, upper bounding) the growth rate of algorithms/functions wrt input size
- For example:
 - A square of side length r has area $O(r^2)$.
 - A circle of radius r has area $O(r^2)$.



Doubling r increases area 4×.
Tripling r increases area 9×.



Doubling r increases area 4×.
Tripling r increases area 9×.

Dependence on Input Size: Big-O

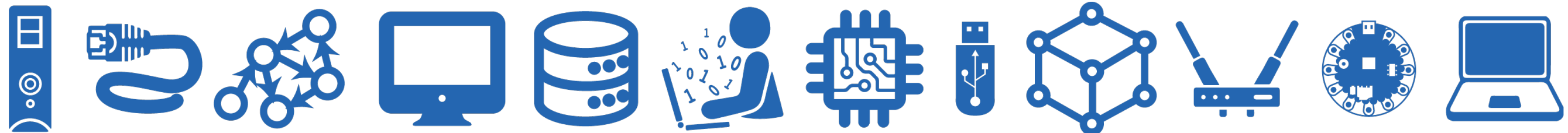
- Big-O notation captures the **rate** at which the **number of steps taken** by the algorithm **grows** wrt size of input n , "as n gets large"
- Not precise by design, it ignores information about:
 - Constants (that do not depend on input size n), e.g. $100n = O(n)$
 - Lower-order terms: terms that contribute to the growth but are not dominant: $O(n^2 + n + 10) = O(n^2)$
- Powerful tool for predicting performance behavior: focuses on what matters, ignores the rest
- Separates fundamental improvements from smaller optimizations
- Won't study this notion too formally: covered in CS136 and CS256!

Append vs Prepend: Big Oh

- Let's revisit append vs prepend efficiency more formally
- How does the cost of `append` grow with number of items in `LinkedList`?
 - Need to traverse `len(LinkedList)` items at least
 - Grows proportionally to input size
 - We call this $O(n)$ or linear time
- How does the cost of `prepend` grow with number of items in `LinkedList`?
 - Same number of steps, independent of input size!
 - We call this $O(1)$ or constant time:
 - Essentially saying does not grow as input size gets large

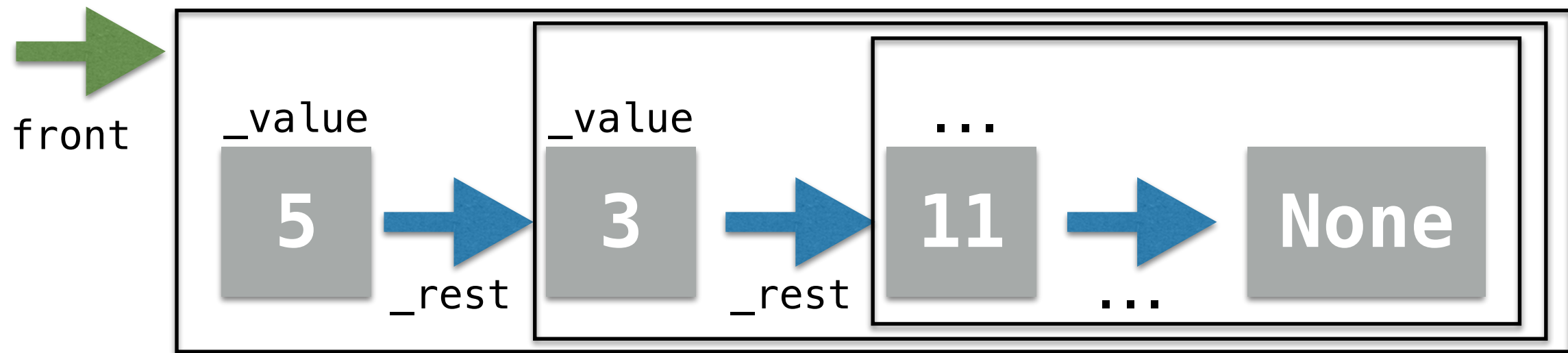
Lists (Arrays) vs. Linked Lists

(Review) Efficiency Trade Offs

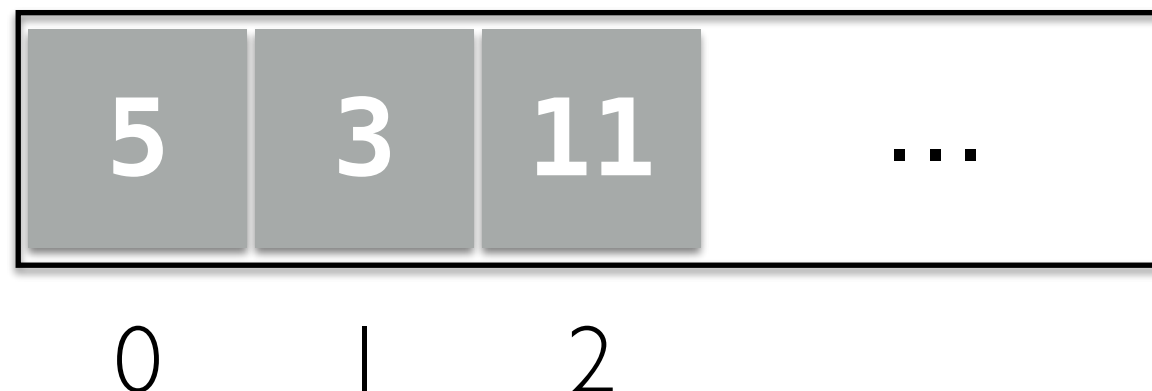


Lists vs Linked Lists

- **Linked Lists:** “pointer-based” data structure, items need not be contiguous in memory

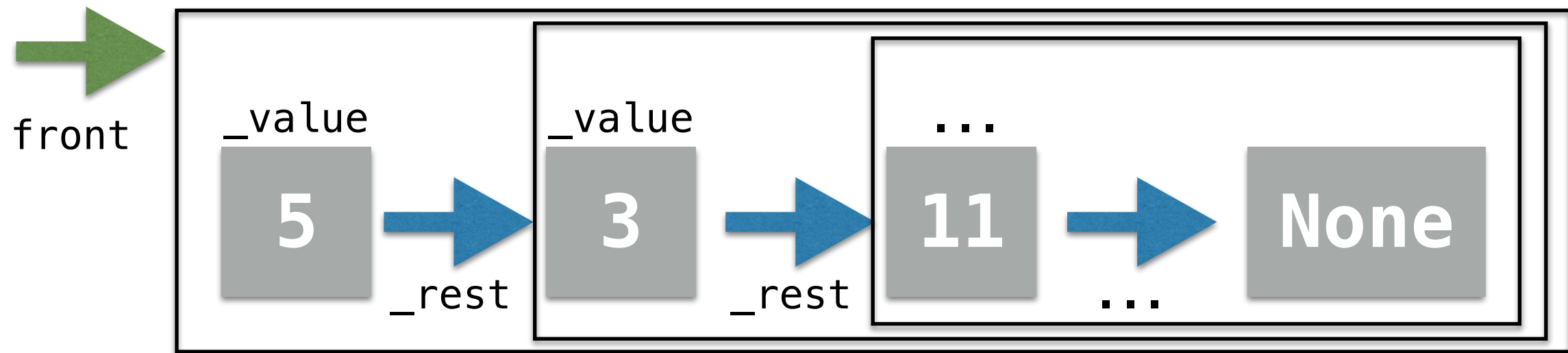


- **Arrays:** index-based data structure items are always stored contiguously in memory (think of a Python built-in list as an array)

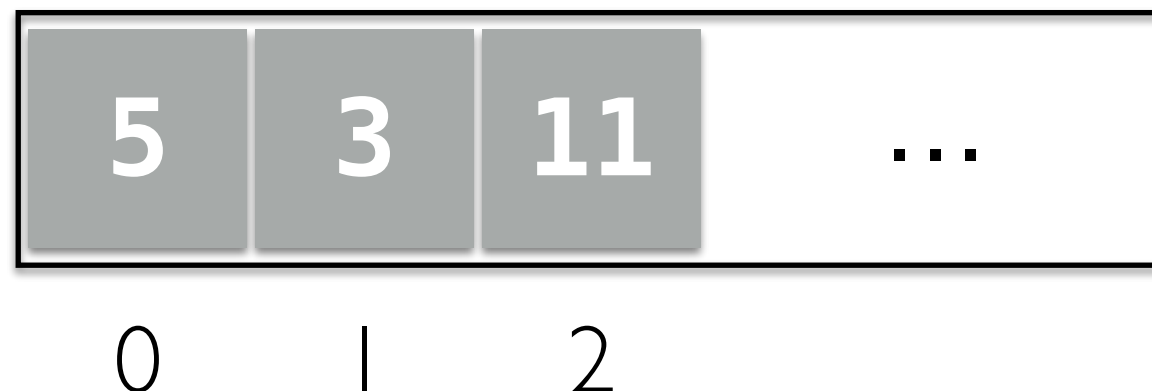


Lists vs Linked Lists

- **Linked Lists:** Can grow and shrink on the fly: do not need to know size at the time of creation (therefore no wasted space!)

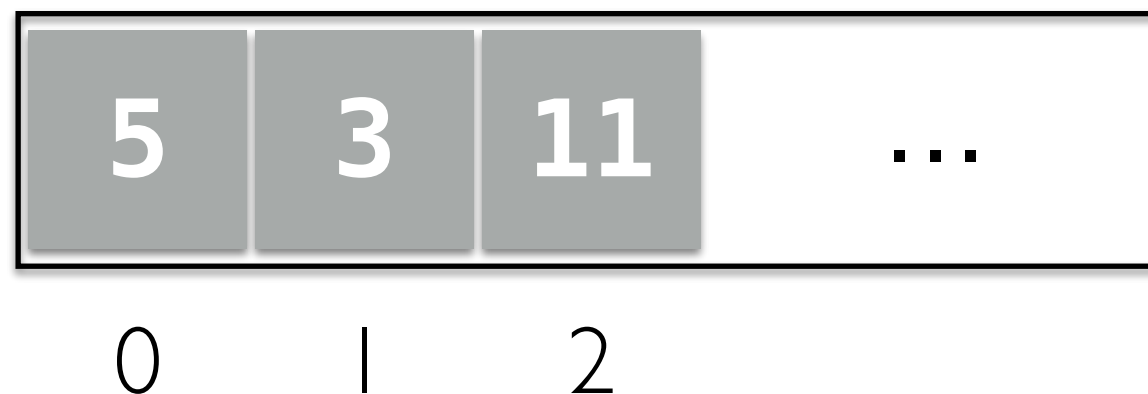
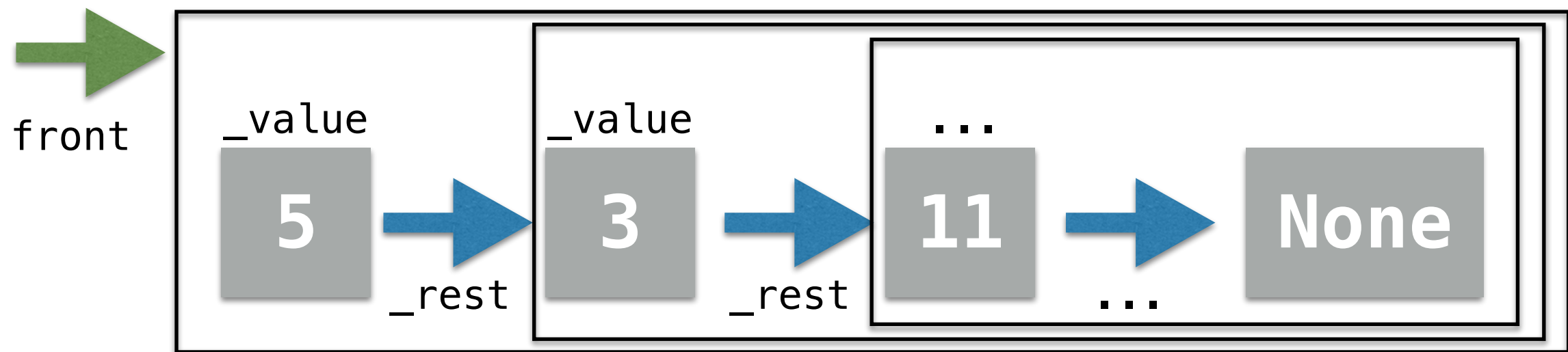


- **Arrays:** must know size in advance to create enough space for each contiguous element (either over-estimate or copy when need to grow)



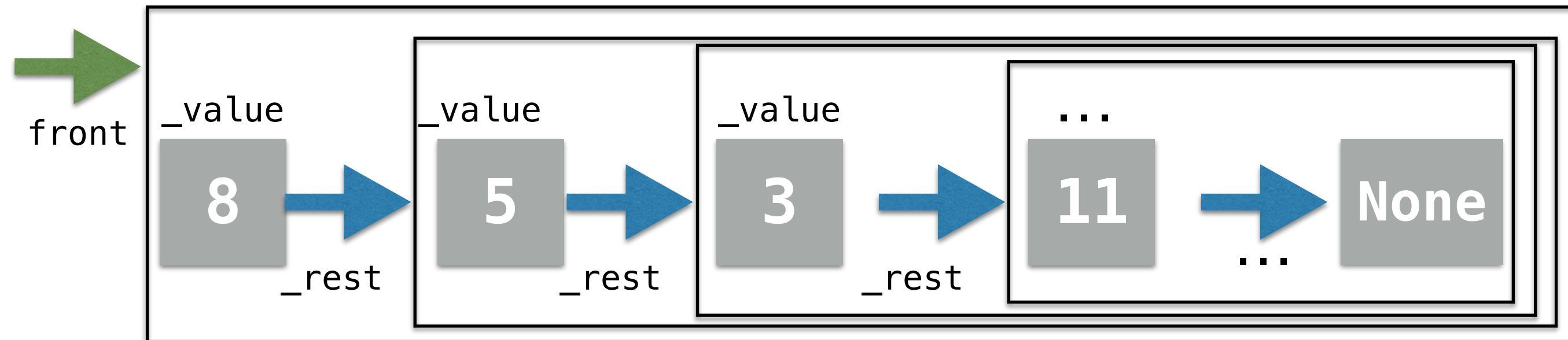
Array vs Linked Lists

- Inserts at the beginning: which one is better?



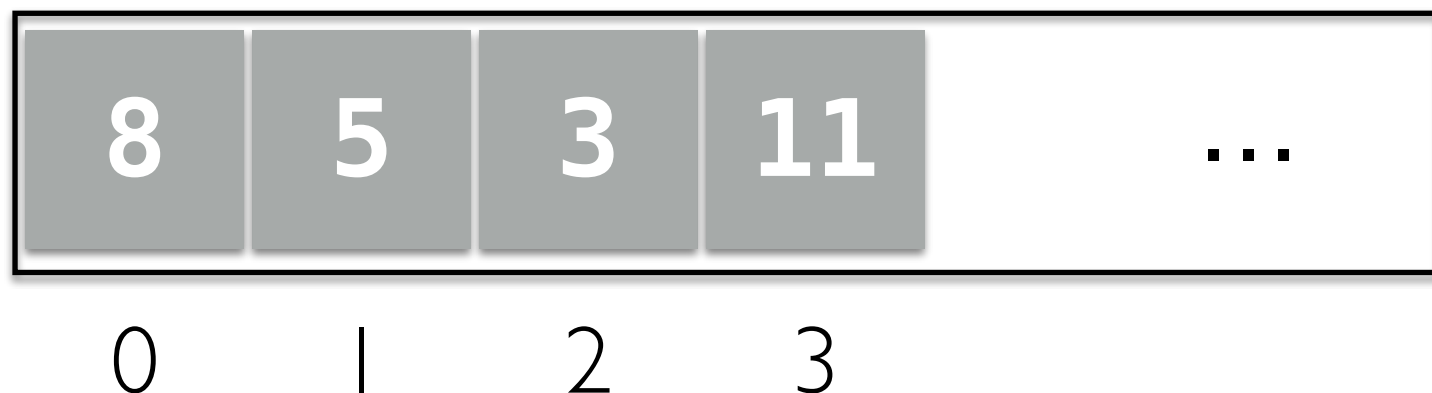
Array vs Linked Lists

- LinkedList prepend is fast! LinkedList steps:
 - Point front to new element
 - Point rest of new element to old list
- Number and sequence of steps don't depend on size of list
 - Therefore, run-time is **constant**, that is, $O(1)$ time



Array vs Linked Lists

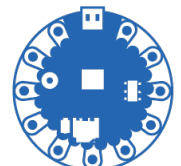
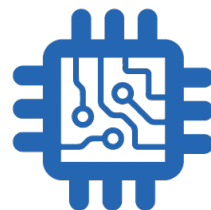
- Now consider an array-based list
- To insert at index 0, we need to shift every element over by one spot
 - This takes time proportional to the size: linear time or $O(n)$
- So when are arrays more efficient?
 - When **indexing** elements: they give **direct access** $O(1)$
 - Linked list: we need to traverse the list to get to the element $O(n)$



So Which is Better?

- It depends!
- Think about what operations are a priority in your program!
 - We should choose the way to represent/organize our data according to how we plan to use the data
- Let's take an example of an application where one of the data structures is way more efficient than the other

Searching in a Sequence



Search

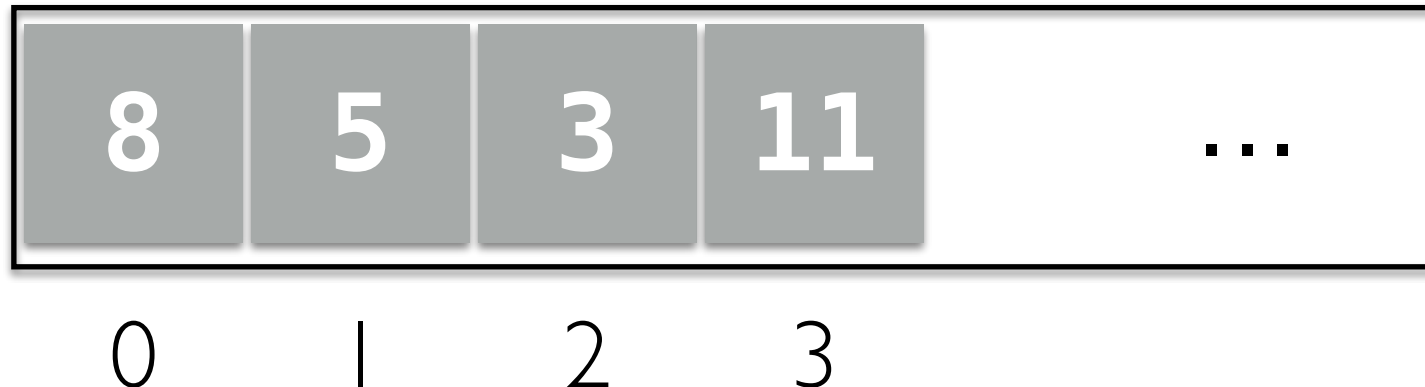
- **Search.** Given an input sequence **seq**, search to see if a given **item** is in the sequence.
 - For example, if some name is in a sequence of student names
- **Input:** a sequence of n items and a query item
 - For now suppose this sequence is in a *random order*
- **Output:** True if query item is in sequence, else False
- Can use **in** operator to do this (calls **__contains__**)
 - But without knowing how it works, can't analyze efficiency
- Let's figure out a direct way to solve this problem

Searching in a Sequence

- First algorithm: iterate through the items in sequence and compare each item to query

```
def linear_search(item, seq):  
    for elem in seq:  
        if elem == item:  
            return True  
    return False
```

Might return early if item is first elem in seq, but we are interested in the **worst case analysis**; this happens if item is not in seq at all



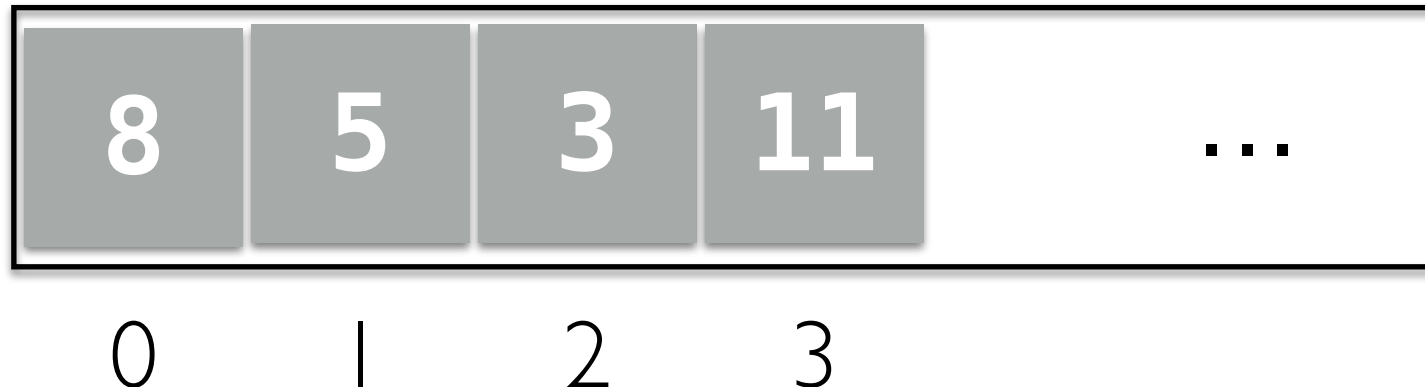
Searching in a Sequence

- In the worst case, we have to walk through the entire sequence
- Overall, the number of steps is linear in n : we write $O(n)$ in Big Oh

```
def linear_search(item, seq):  
    for elem in seq:  
        if elem == item:  
            return True  
    return False
```

Loop runs n items
in worst case

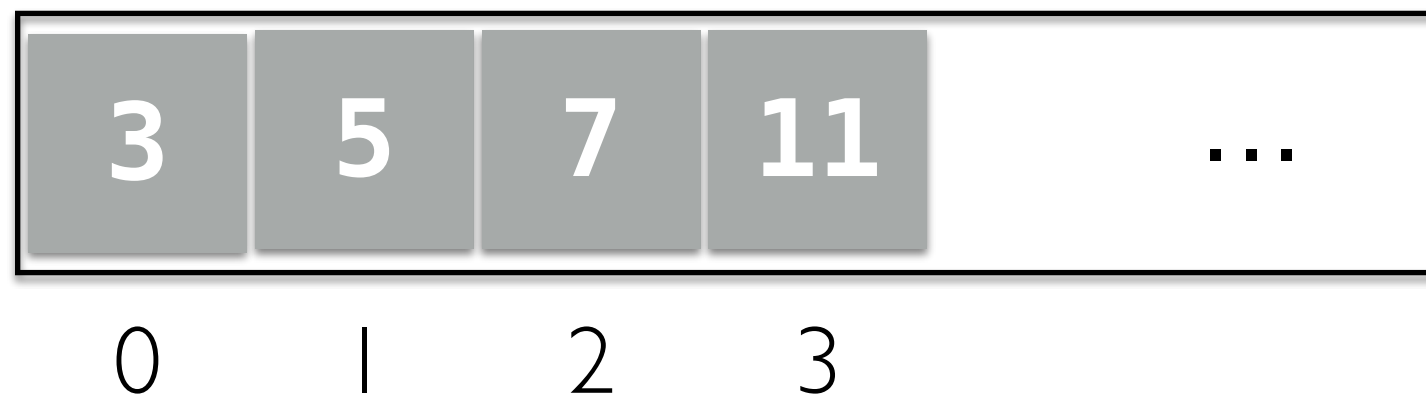
One equality check per
iteration: assume comparing
`elem == item` is one step



Searching in an Array

- Can we do better?
 - Not if the elements are in arbitrary order
- What if the sequence is **sorted**?
 - Can we utilize this somehow and search more efficiently?

How do we search for an item (say 10) in a **sorted** array?

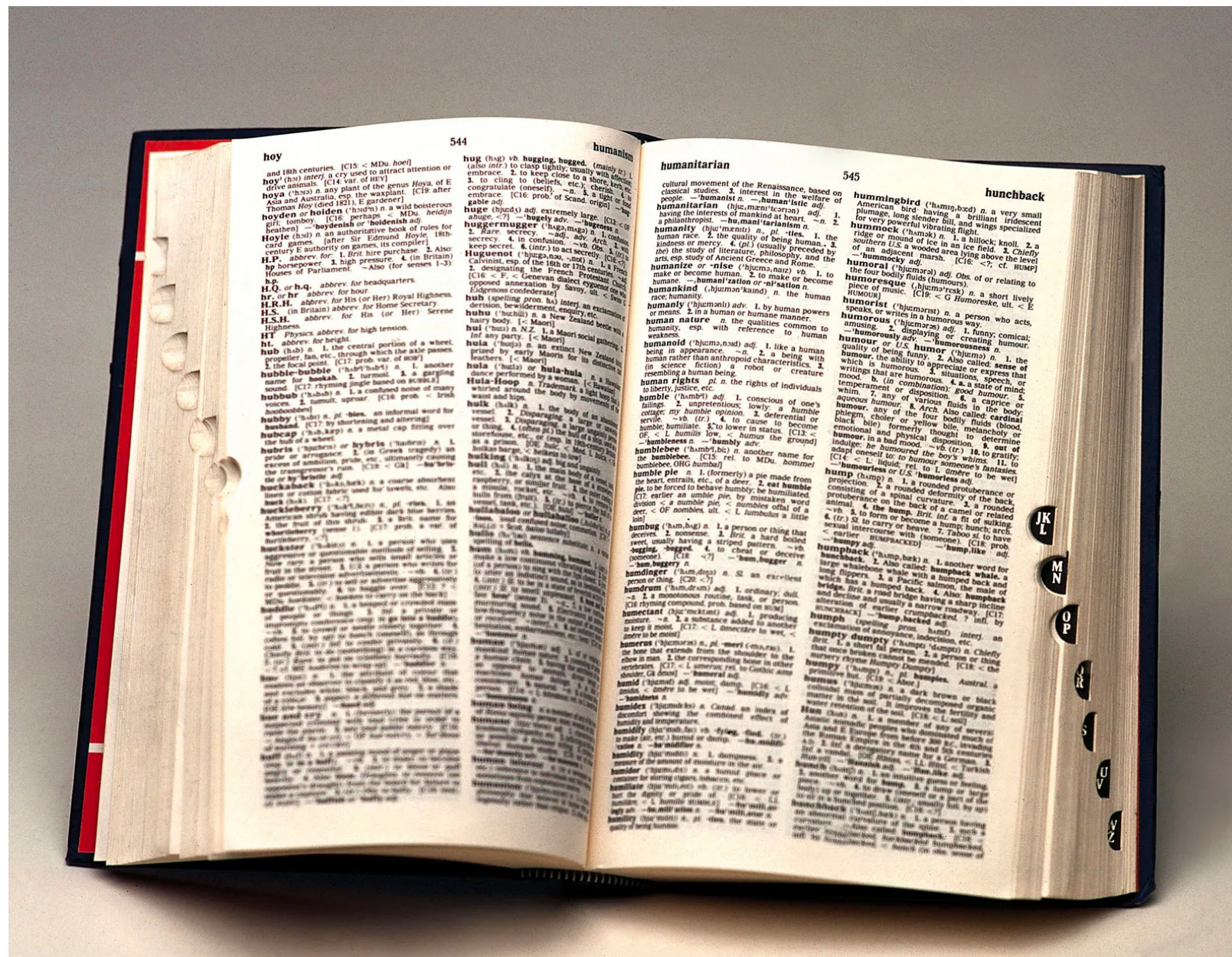


Let's Play a Game

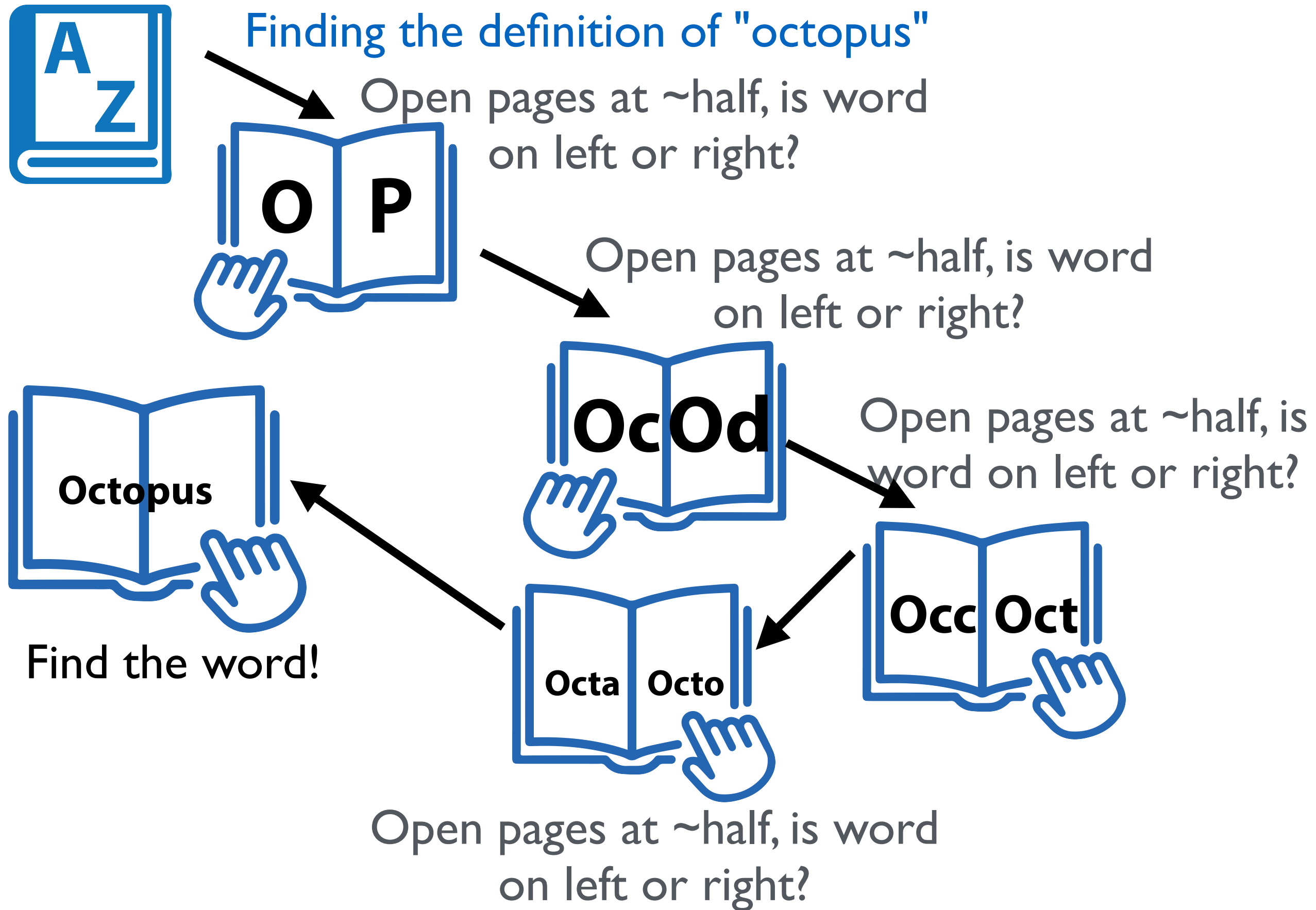
- I'm thinking of a number between 0 and 100...
- If you guess a number, I'll tell you either:
 - You've guessed my number!
 - My number is larger than your guess
 - My number is smaller than your guess
- What is your guessing strategy?
- What if I picked a number between 0 and 1 million?

Example: Dictionary

- How do we look up a word in a (physical) dictionary?
- Words are listed in alphabetical order



Example: Dictionary



How Good is This Method?

- **Goal:** Analyze # pages we need to look at until we find the word
- We want the worst case: possible to get lucky and find the word right on the middle page, but we don't want to consider luck!
- Each time we look at the “middle” of the remaining pages, the number of pages we need to look at is divided by 2
- A 1024-page dictionary requires at most 11 lookups:
1024 pages, < 512, <256, <128, <64, <32, <16, <8, <4, <2, <1 page.
- Only needed to look at 11 pages out of 1024 !
- **Challenge:** What if we have an n page dictionary, what function of n characterizes the (worst-case) number of lookups?



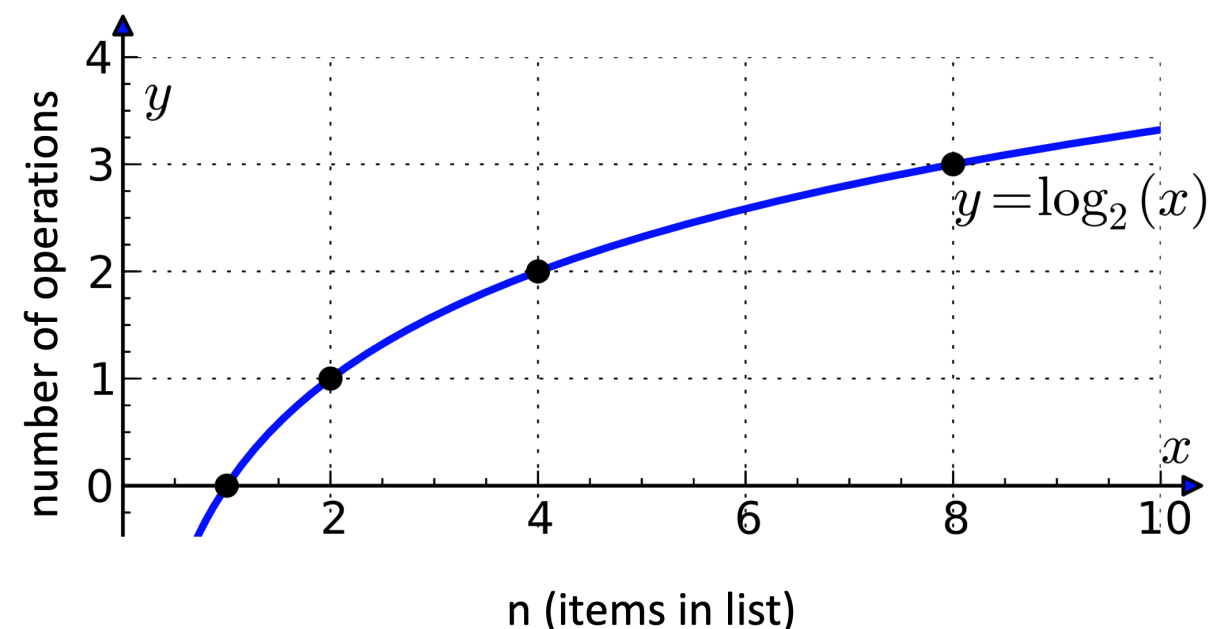
Logarithms: our favorite **function**

- Logarithms are the inverse function to exponentiation
- $\log_2 n$ describes the exponent to which **2** must be raised to produce ***n***
- That is, $2^{\log_2 n} = n$
- Alternatively:
 - $\log_2 n$ (essentially) describes the number of times ***n*** must be divided by **2** to reduce it to **1** or below
- For us, here's the important takeaway:
 - How many times can we divide ***n*** by **2** until we get down to **1**
 - $\approx \log_2 n$



$O(\log n)$

- When you double the number of elements, it only increases the number of operations by 1
- 2 items in the list, 1 operation
 - $\log 2 = 1$
- When you have 4 items, increases operations to 2
 - $\log 4 = 2$
- When you have 8 items, only 3 operations
 - $\log 8 = 3$



Binary Search

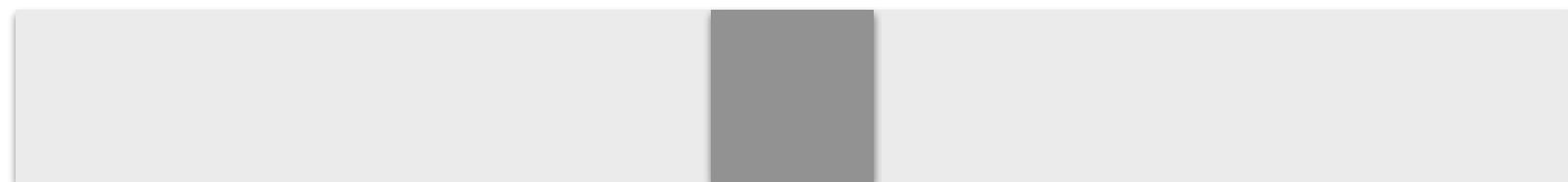
- The **recursive search algorithm** we described to search in a sorted array is called **binary search**
- It can be much more efficient than a **linear search**
 - Takes $\approx \log n$ lookups if we can index into sequence efficiently
- Which data structure supports fast access/indexing?
 - Accessing an item at index i in an array requires constant time
 - Accessing an item at index i in a LinkedList can require traversing the whole list (even if it is sorted!): linear time
- To get a more efficient search algorithm, it is not only important to use the right algorithm, we need to use the right data structure as well!

Binary Search

- Base cases? When are we done?
 - If list is too small (or empty) to continue searching
 - If item we're searching for is the middle element

```
def binary_search(a_lst, item):  
    """Assume a_lst is sorted."""  
    n = len(a_lst)  
    mid = n // 2  
    # base case 1  
    if n == 0:  
        return False  
    # base case 2  
    elif item == a_lst[mid]:  
        return True  
  
    # recursive cases...
```

Check middle



$mid = n // 2$

Binary Search

- Recursive case:
 - Recurse on left side if item is smaller than middle
 - Recurse on right side if item is larger than middle

If $\text{item} < a_lst[mid]$, then need to search in $a_lst[:mid]$



$$\text{mid} = n // 2$$

Binary Search

- Recursive case:
 - Recurse on left side if item is smaller than middle
 - Recurse on right side if item is larger than middle

If $\text{item} > a_lst[\text{mid}]$, then need to search in $a_lst[\text{mid}+1:]$



$$\text{mid} = n // 2$$

Binary Search

```
def binary_search(a_lst, item):  
    """ Assume a_lst is sorted. """  
    n = len(a_lst)  
    mid = n // 2  
    # base case 1  
    if n == 0:  
        return False  
    # base case 2  
    elif item == a_lst[mid]:  
        return True  
  
    # recurse on left  
    elif item < a_lst[mid]:  
        return binary_search(a_lst[:mid], item)  
    # recurse on right  
    else:  
        return binary_search(a_lst[mid + 1:], item)
```

Technically, there is one *small* problem with our implementation. List splicing is actually $O(n)$!

Binary Search: Better

```
def binary_search_better(a_lst, item, index_start, index_end):  
    """ Assume a_lst is sorted. """  
  
    n = index_end - index_start  
    mid = (n // 2) + index_start  
    # base case 1  
    if n <= 0:  
        return False  
    # base case 2  
    elif item == a_lst[mid]:  
        return True  
  
    # recurse on left  
    elif item < a_lst[mid]:  
        return binary_search_better(a_lst, item, 0, mid)  
    # recurse on right  
    else:  
        return binary_search_better(a_lst, item, mid+1, index_end)
```

Passing start/end indices as arguments avoids the need to splice. No extra copying of the list elements!

Big-O Notation

- Tells you how fast an algorithm is / the run-time of algorithms
 - But not in seconds!
- Tells you how fast the algorithm grows in number of operations

O(log n)
"Big O" Number of Operations

Understanding Big-O

- Notation: n often denotes the number of elements (size)
- **Constant time** or $O(1)$: when an operation does not depend on the number of elements, e.g.
 - Addition/subtraction/multiplication of two values, or defining a variable etc are all constant time
- **Linear time** or $O(n)$: when an operation requires time proportional to the number of elements, e.g.:

```
for item in seq:  
    <do something>
```
- **Quadratic time** or $O(n^2)$: nested loops are often quadratic, e.g.,

```
for i in range(n):  
    for j in range(n):  
        <do something>
```

Big-O: Common Functions

- Notation: n often denotes the number of elements (size)
- Our goal: understand efficiency of some algorithms at a high level

