Algorithm Efficiency: Measurements of

Computer Science 112
Boston University

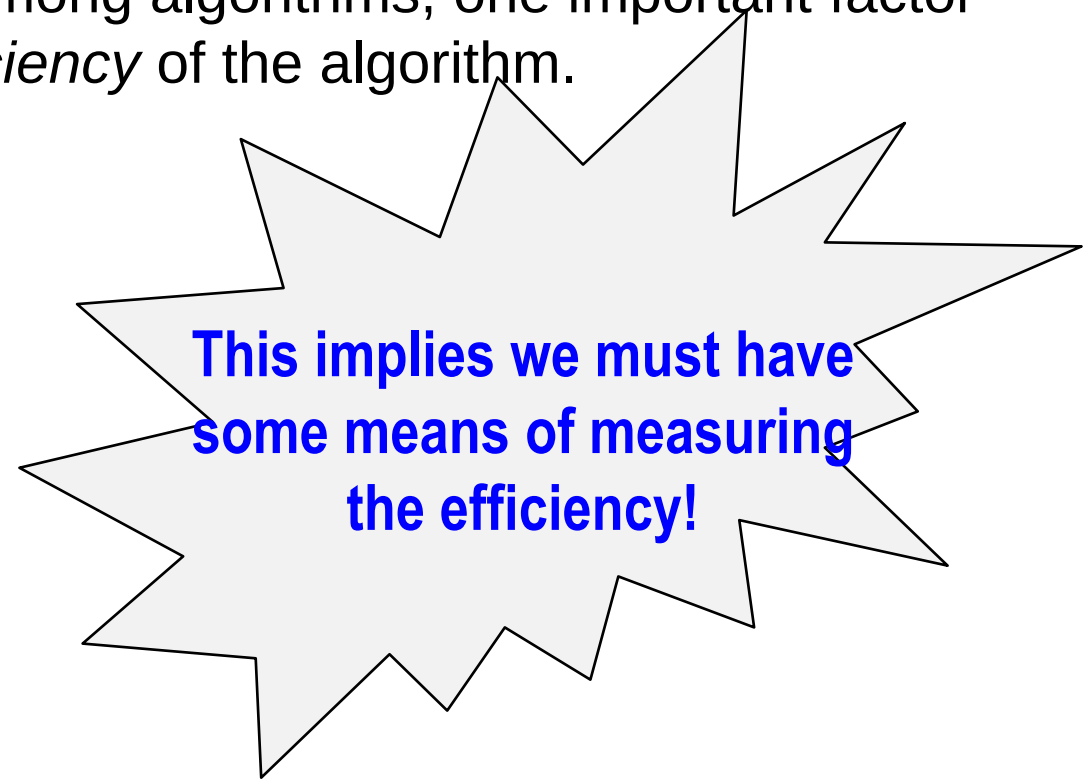
Christine Papadakis

Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm.

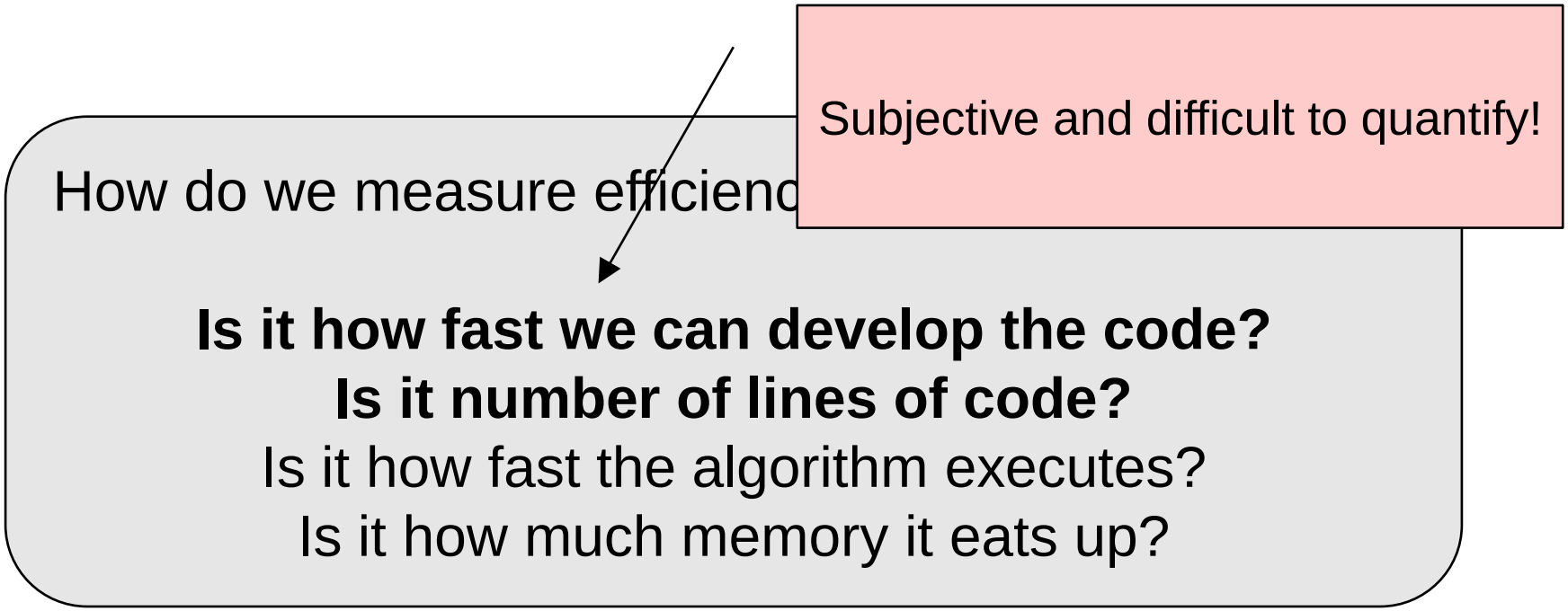
Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm.



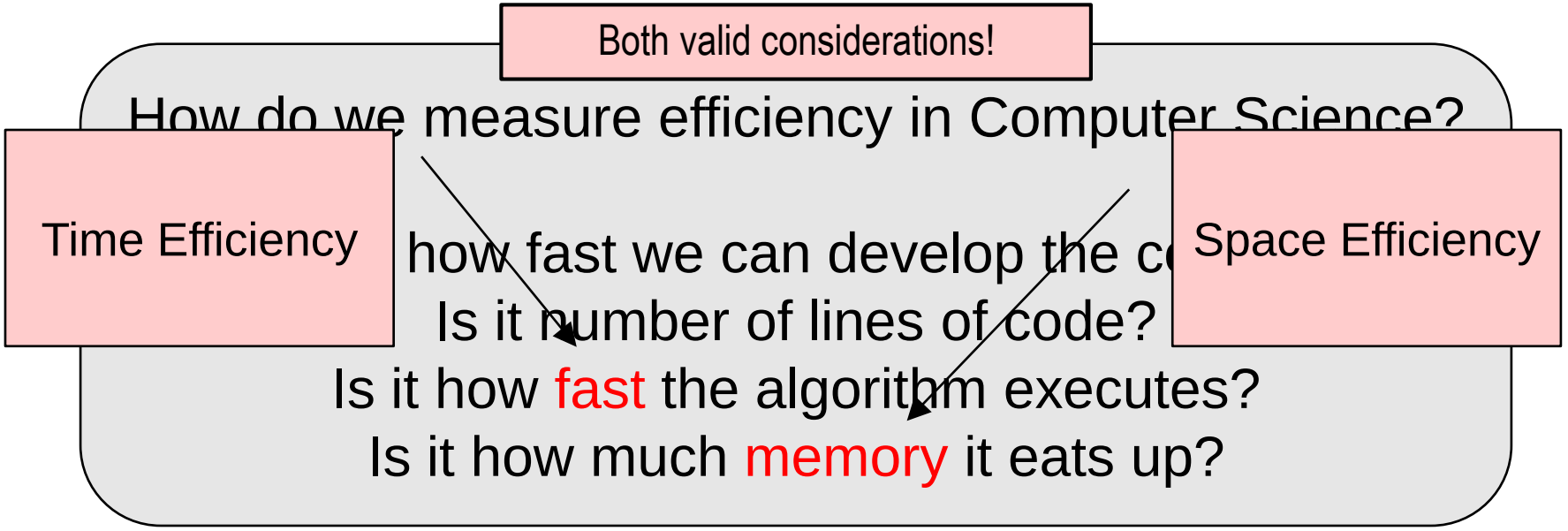
Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm.



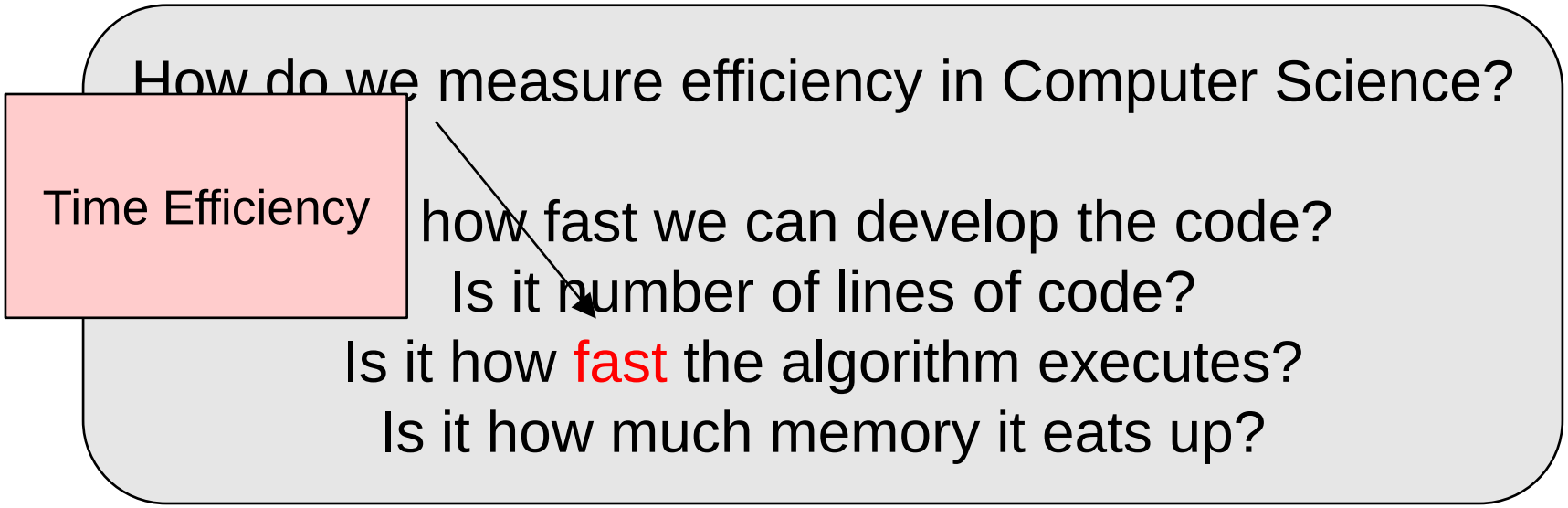
Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm.



Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm. **Let's focus on:**



Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm. Let's focus on:
  - time efficiency: *how quickly an algorithm executes?*

But, what exactly is this a measure of? Elapsed Time?

Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm. Let's focus on:
  - time efficiency: *how quickly an algorithm executes?*

Elapsed time (by itself) is not really an accurate measure of efficiency. There are many factors that influence the speed at which an algorithm executes:

- The physical machine the algorithm was executed on.
- The processes running at the same time.
- The language that the algorithm was encoded in, etc.

Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm. Let's focus on:
  - time efficiency: *how quickly an algorithm executes?*

We want to measure execution time in terms of the *number of operations* the algorithm performs.

Why?

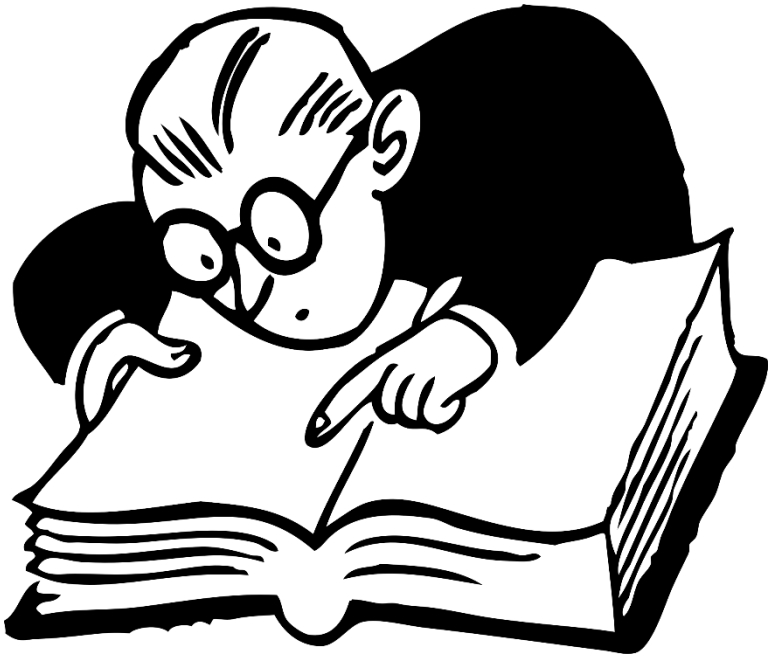
Algorithm Efficiency

- For any given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is the relative *efficiency* of the algorithm.
  - time efficiency: *how quickly an algorithm executes?*

Because this will tell us how the algorithm scales. By this we mean, how the algorithm performs on ***increasingly*** larger data sets.

Example of Comparing Algorithms

- Consider the problem of finding a phone number in a phonebook.



Finding a Phone Number:

Sequential Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    for (int i = 1; i <= phonebook.num_pages(); i++ ){
        if (person is found on the current page) {
            number = the person's phone number
            break;
        }
    }
    return number;
}
```

Scan each page of the phone book, one page at a time, until we find the person we are looking for … or we run out of pages to search.

Finding a Phone Number:

Sequential Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    for (int i = 1; i <= phonebook.num_pages(); i++ ){
        if (person is found on the current page) {
            number = the person's phone number
            break;
        }
    }
    return number
}
```

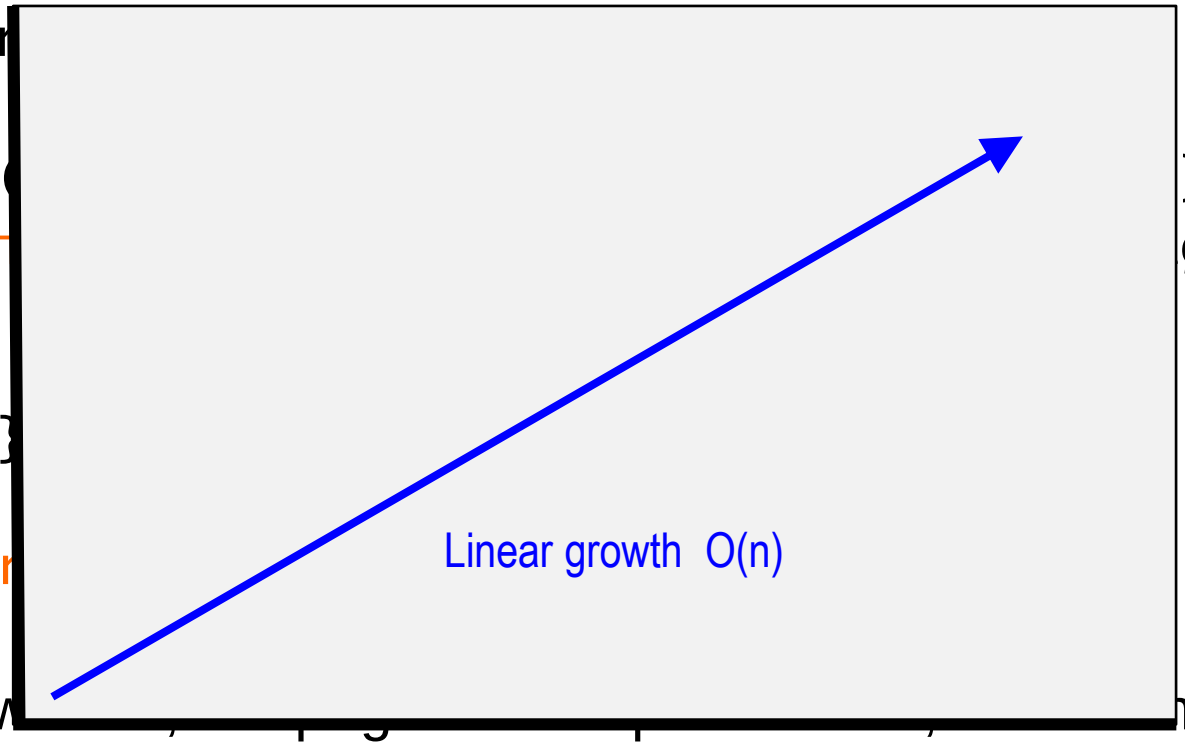
- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case? 1,000
- What if there were 1,000,000 pages? 1,000,000

The running time of this algorithm "*grows **proportionally***" to *n*, where *n* = # of pages.

Finding a Phone Number:

Sequential Search

```
public String findNumber(String name, Book phonebook){
    S
    for (                                    ); i++ ){
        -                                    ge) {
                                             r
    }
    retur
}
```



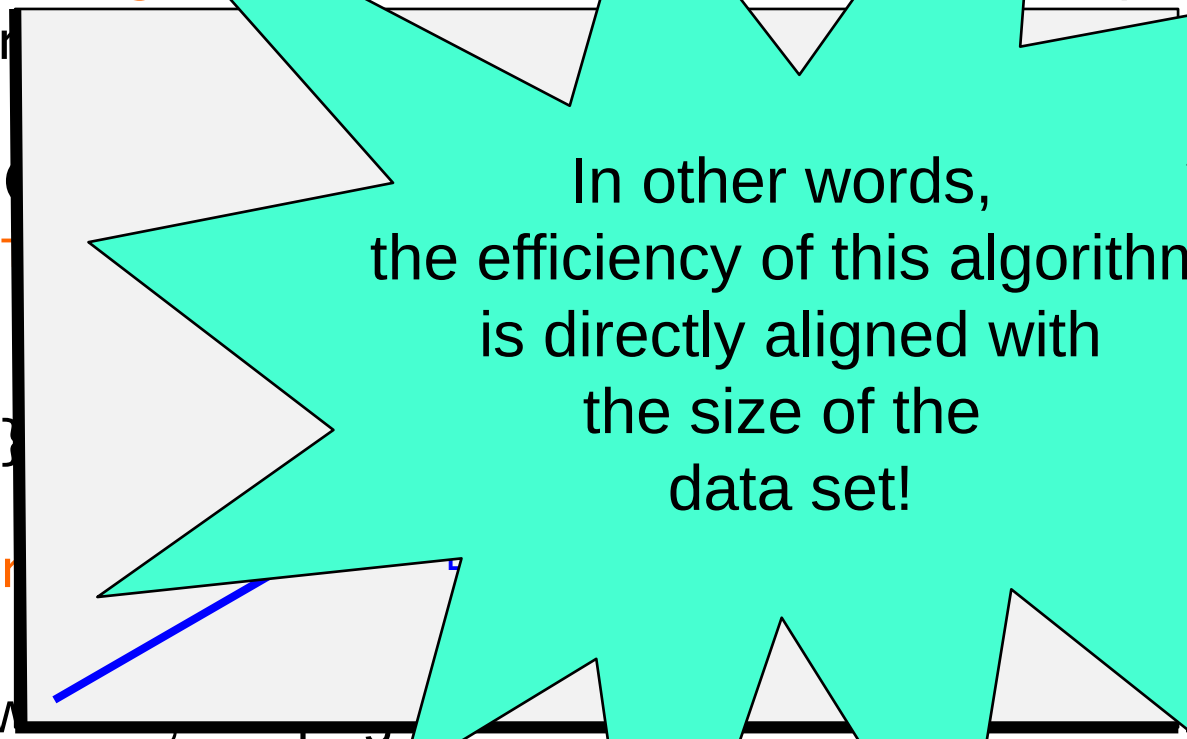
- If there w ages would this look at in the worst case? 1,000
- What if there were 1,000,000 pages? 1,000,000

The running time of this algorithm "*grows **proportionally*** to *n*, where *n* = # of pages.

Finding a Phone Number

Sequential Search

```
public String findNumber(String name, PhoneBook phonebook){
    String t
    for (                                    i++ ){
        t
    }
    retur
}
```



In other words,
the efficiency of this algorithm
is directly aligned with
the size of the
data set!

- If there w ages would this look at in the worst ca 1,00
- What if there were 1,000,000 pa s? 1,000,000

The running time of this algorithm
"*grows **proportionally***" to *n*,
where *n* = # of pages.

Finding a Phone Number:

Binary Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    int min = 1;
    int max = phonebook.num_pages();

    while (min <= max) {
        mid = (min + max) / 2;      # the middle page
        if (person is found on page mid) {
            number = the person's number
            break;
        } else if (person comes earlier in phonebook)
            max = mid - 1;
        else:
            min = mid + 1;
    }

    return number;
}
```

What is this algorithm doing with each iteration of the loop?

Finding a Phone Number:

Binary Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    int min = 1;
    int max = phonebook.num_pages();

    while (min <= max) {
        mid = (min + max) / 2;      # the middle page
        if (person is found on page mid) {
            number = the person's number
            break;
        } else if (person comes earlier in phonebook)
            max = mid - 1;
        else:
            min = mid + 1;
    }

    return number;
}
```

It cuts the search range in half!

Finding a Phone Number:

Binary Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    int min = 1;
    int max = phonebook.num_pages();

    while (min <= max) {
        mid = (min + max) / 2;      # the middle page
          if (person is found on page mid) {
            number = the person's number
            break;
          } else if (person comes earlier in phonebook)
                max = mid - 1;
          else:
                min = mid + 1;
     }
    return number;
}
```

Why?

- If there were 1,000 pages in the phonebook, how many pages would this look at in the *worst* case? approx. 10
- What if there were 1,000,000 pages? approx. 20

Finding a Phone Number:

Binary Search

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    int min = 1;
    int max = phonebook.num_pages();

    while (min <= max) {
        mid = (min + max) / 2;      # the middle page
          if (person is found on page mid) {
            number = t
            break;
          } else if (
                max =
          else:
                min =
     }
    return number;
}
```

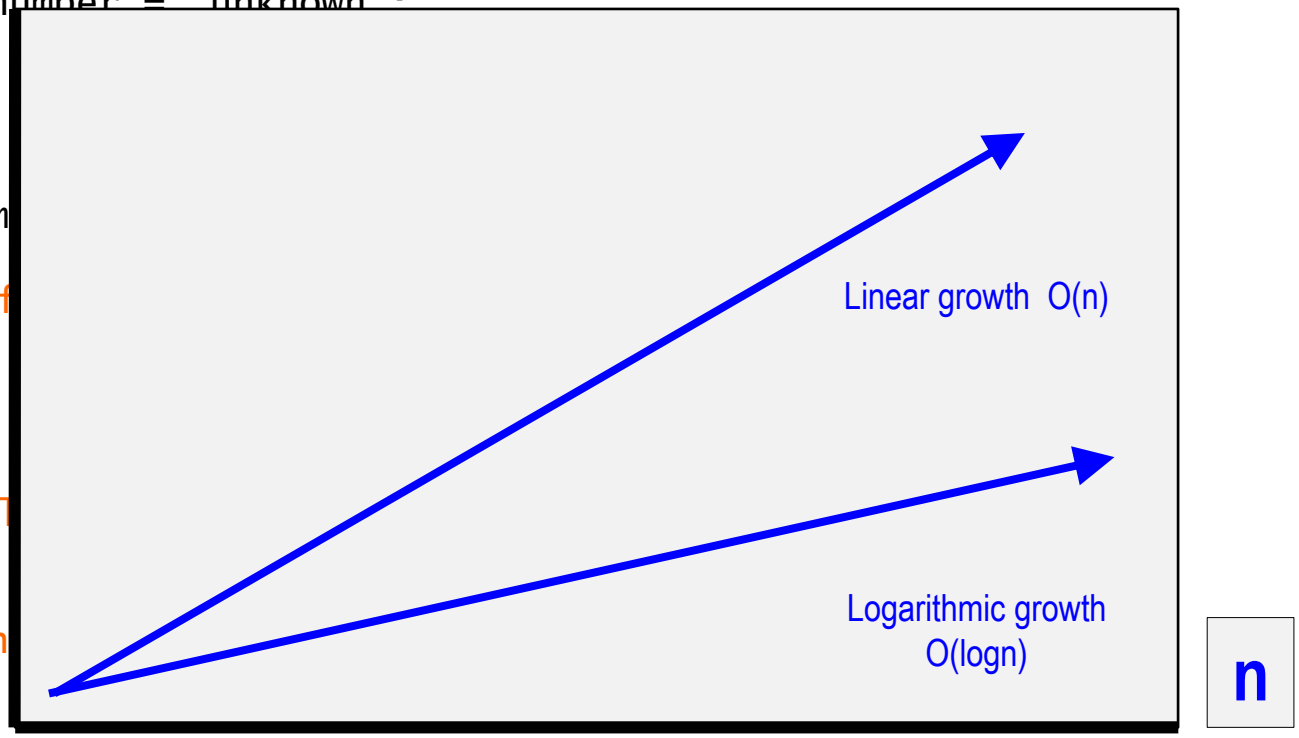
By continuously cutting the problem size in half, the running time "grows proportionally" to $\log_2 n$, *where* (n = # of pages).

- If there were 1,000 pages in the phonebook, how many pages would this look at in the *worst* case? approx. 10
- What if there were 1,000,000 pages? approx. 20

Finding a Phone Number:

algorithm 2

```
public String findNumber(String name, Book phonebook){
    String number = "unknown";

    int
    int max

    while (m
        mid
          if


          }

          el

      }
    return n
}
```



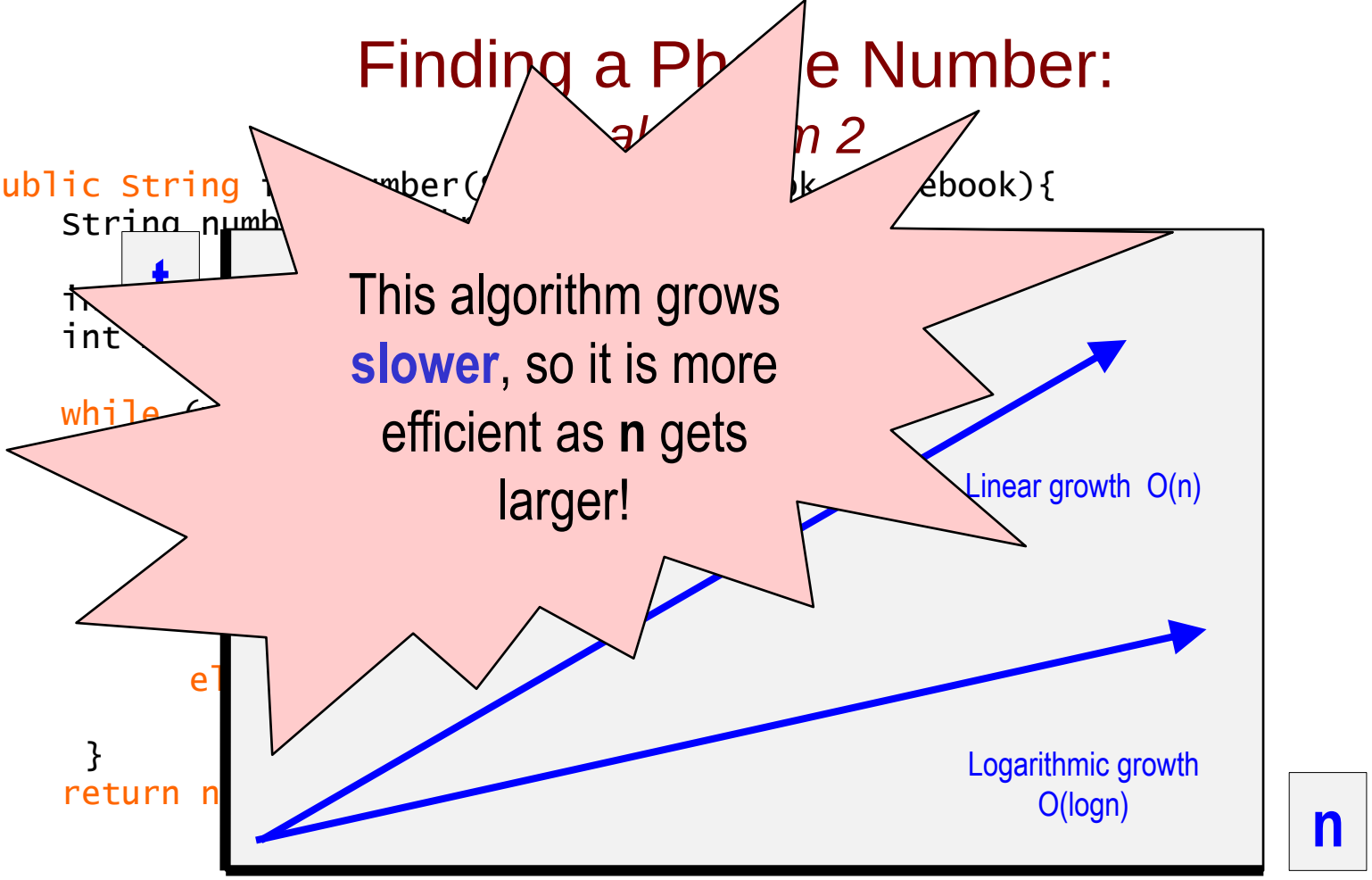
- If there were 1,000 pages in the phonebook, how many pages would this look at in the *worst* case? approx. 10
- What if there were 1,000,000 pages? approx. 20

Finding a Ph e Number:

n 2

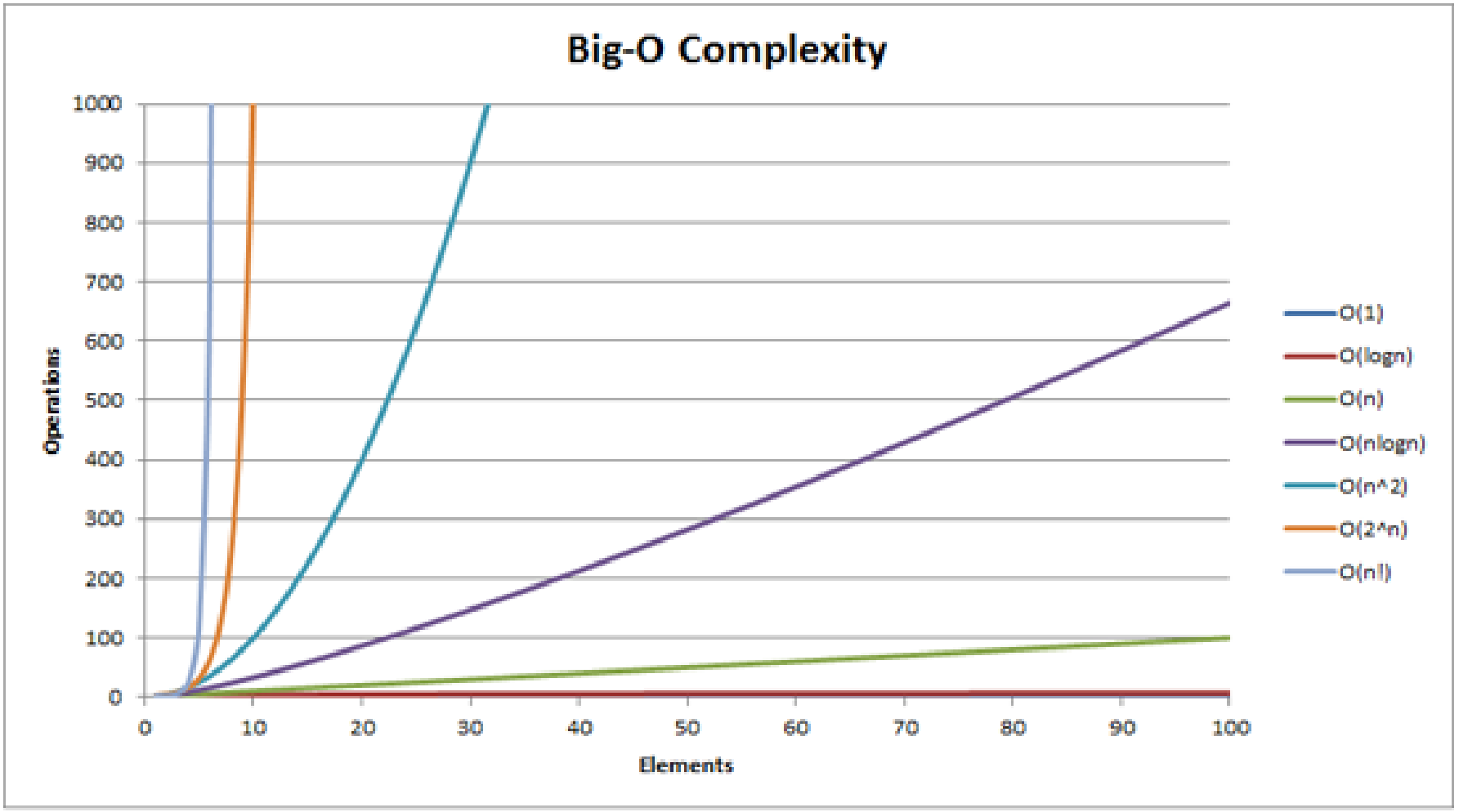
```
public String f      mber(      k      ebook){
    String numb
    f
    int

    while (

    el

     }
    return n
}
```

This algorithm grows **slower**, so it is more efficient as **n** gets larger!



- If there were 1,000 pages in the phonebook, how many pages would this look at in the *worst* case? approx. 10
- What if there were 1,000,000 pages? approx. 20

Algorithm Analysis



Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a value in an array
- A main branch of study in Computer Science is devoted to developing increasingly more efficient search algorithms.
- We have just looked at two:
  - Algorithm #1: Sequential Search
  - Algorithm #2: Binary Search

For large collections of data, binary search is **significantly** faster than sequential search.

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a value in an array
- A main branch of study in Computer Science is devoted to developing increasingly more efficient search algorithms.
- We have just looked at two:
  - Algorithm #1: Sequential Search
  - Algorithm #2: Binary Search

But there is a catch! For binary search to work, what must be true?

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a value in an array
- A main branch of study in Computer Science is devoted to developing increasingly efficient algorithms.
- We have just looked at two algorithms:
  - Algorithm #1: Linear search
  - Algorithm #2: Binary search

Data must be in sorted order!

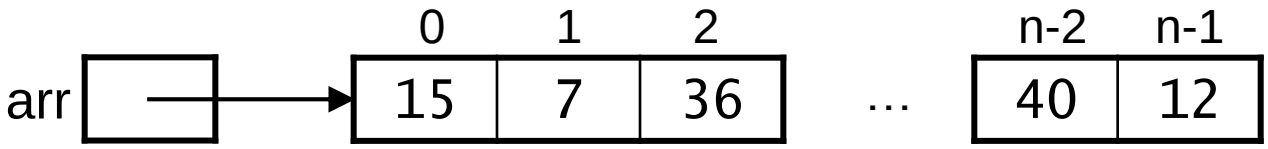
But there is a catch! For binary search to work, what must be true?

Sorting and Algorithm Analysis: The Basics

Computer Science 112
Boston University

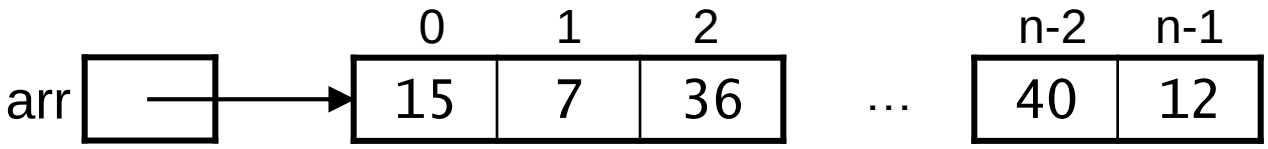
Christine Papadakis-Kanaris

Sorting an Array of Integers



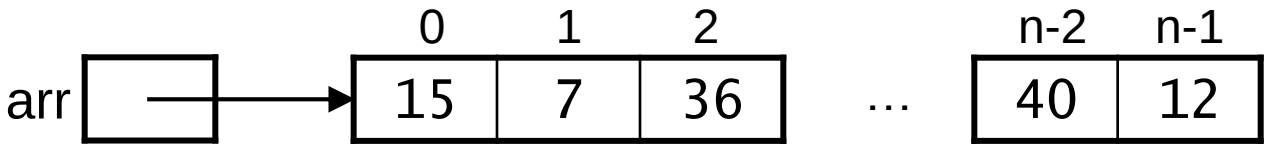
- Ground rules:
  - sort the values in increasing order
  - sort “in place,” using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort “in place,” using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of *operations* needed to sort the array.

Sorting an Array of Integers

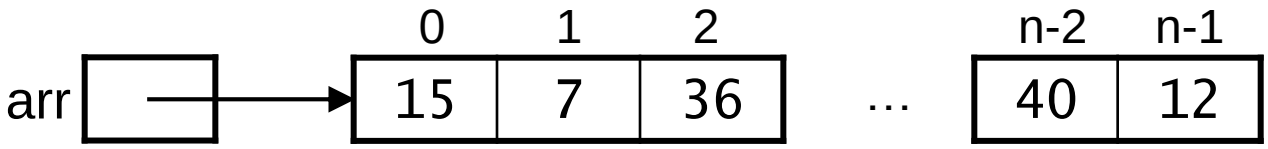


- Ground rules:
  - sort the values in increasing order
  - sort “in place,” using only a small a
- Terminology:
  - position: one of the memory locatio
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of *operations* needed to sort the array.

Which are?

.

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small a
- Terminology:
  - position: one of the memory locatio
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of *operations* needed to sort the array.

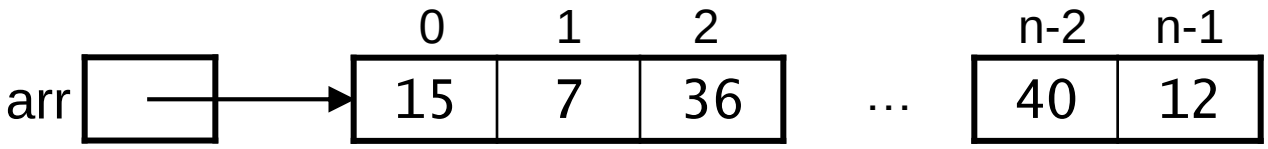
Which are?

number of *comparisons* and *moves* needed to sort the array.

```
comparison is applying a relational operation on two elements of the array.
example: arr[1] > arr[2]
```

```
move = copying an element from one position to another
example: arr[3] = arr[5];
```

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small a
- Terminology:
  - position: one of the memory locatio
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of *operations* needed to sort the array.

Which are?

number of *comparisons* and *moves* needed to sort the array.

comparison is applying a relational operation on two elements of the array.
example: arr[1] > arr[2]

move = copying an element from one position to another
example: arr[3] = arr[5];

Defining a Class for our Sort Methods

```
public class Sort {
    public static void bubbleSort(int[] arr) {
        ...
    }
    public static void insertionSort(int[] arr) {
        ...
    }
    ...
}
```

- Our `Sort` class is simply a collection of methods like Java's built-in `Math` class.
- Because we never create `Sort` objects, all of the methods in the class must be *static*.
  - outside the class, we invoke them using the class name: e.g., `Sort.bubbleSort(arr)`