

# CS2040S

## Data Structures and Algorithms

Welcome!

**ARCHIPELAGO**

is open

# How to Search!

---

## Algorithm Analysis

- Big-O Notation
- Model of computation

## Searching

## Peak Finding

- 1-dimension
- 2-dimensions

# Admin

---

## Tutorial and Recitation Registration

- Sorry for all the announcements!
- Still in progress.
- 95% settled.
- Please be patient!
- Please do not use ModReg to swap/petition/change your tutorial/recitation registration.

# Admin

---

## Midterm Exam: Week 7

- Trying to arrange in-person midterm.
  - Fewer problems in person!
  - No Zoom issues!
- Requirement:  $< 50$  students / room.
- If we cannot find sufficient rooms, may reschedule:
  - E.g., an evening exam at 7pm
  - E.g., Saturday

# How to Search!

---

## Algorithm Analysis

- Big-O Notation
- Model of computation

## Searching

## Peak Finding

- 1-dimension
- 2-dimensions

# Algorithm Analysis

---

Warm up: which takes longer?

```
void pushAll(int k) {  
    for (int i=0;  
        i<= 100*k;  
        i++)  
    {  
        stack.push(i) ;  
    }  
}
```

$O(100k)$

```
void pushAdd(int k) {  
    for (int i=0; i<= k; i++)  
    {  
        for (int j=0; j<= k; j++) {  
            stack.push(i+j) ;  
        }  
    }  
}
```

$O(k^2)$



# Algorithm Analysis

---

Warm up: which takes longer?

```
void pushAll(int k) {  
    for (int i=0;  
        i<= 100*k;  
        i++)  
    {  
        stack.push(i) ;  
    }  
}
```

$100k$  push operations

```
void pushAdd(int k) {  
    for (int i=0; i<= k; i++)  
    {  
        for (int j=0; j<= k; j++) {  
            stack.push(i+j) ;  
        }  
    }  
}
```

$k^2$  push operations

Which grows faster?

$$T(k) = 100k$$

$$T(0) = 0$$

$$T(1) = 100$$

$$T(100) = 10,000$$

$$T(1000) = 100,000$$

$$T(k) = k^2$$

$$T(0) = 0$$

$$T(1) = 1$$

$$T(100) = 10,000$$

$$T(1000) = 1,000,000$$



Always think of big input

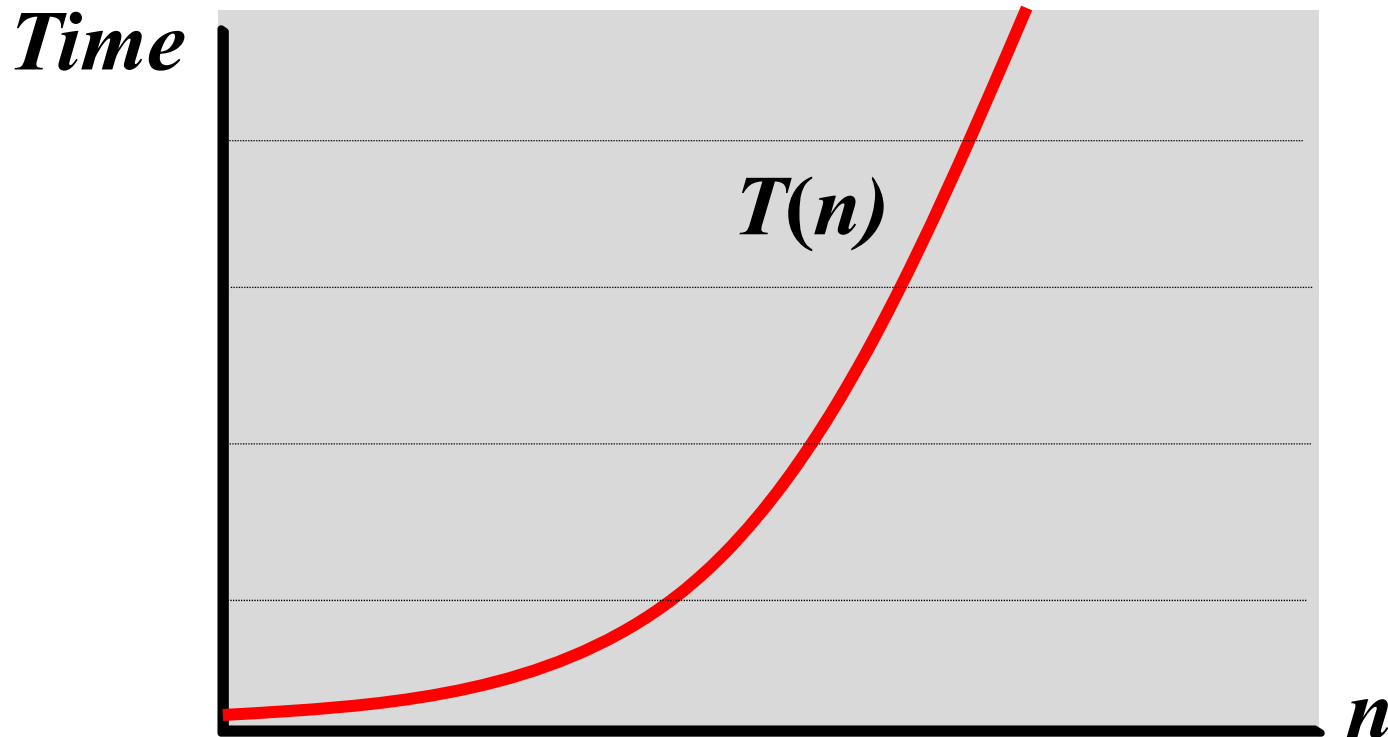


# Big-O Notation

---

How does an algorithm scale?

- For large inputs, what is the running time?
- $T(n)$  = running time on inputs of size  $n$



# Big-O Notation

---

Definition:  $T(n) = O(f(n))$  if  $T$  grows no faster than  $f$

**$T(n) = O(f(n))$**  if:

- there exists a constant  $c > 0$
- there exists a constant  $n_0 > 0$

such that for all  $n > n_0$ :

$$\mathbf{T(n) \leq c f(n)}$$

# Big-O Notation

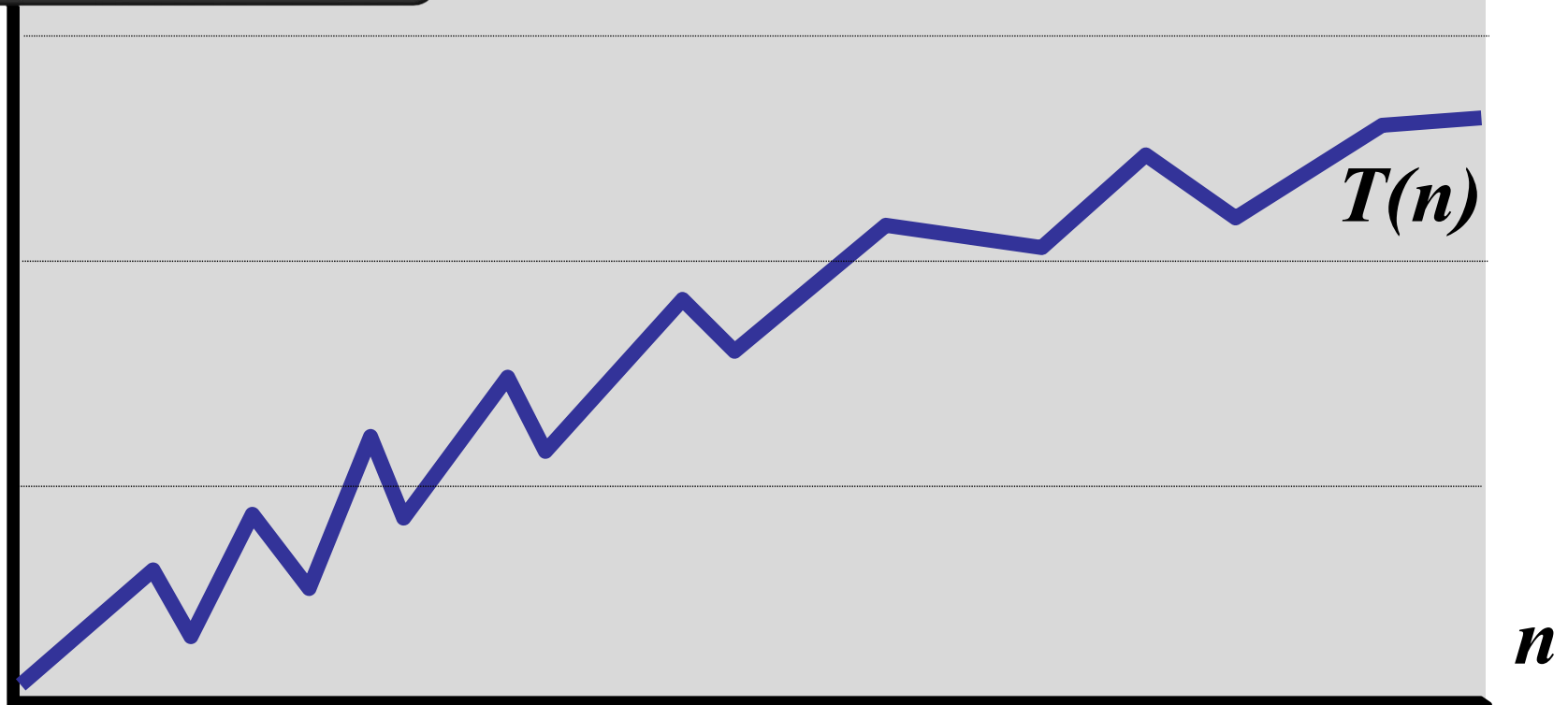
---

$T(n) = O(f(n))$  if:

- there exists a constant  $c > 0$
- there exists a constant  $n_0 > 0$

such that for all  $n > n_0$ :

$$T(n) \leq c f(n)$$



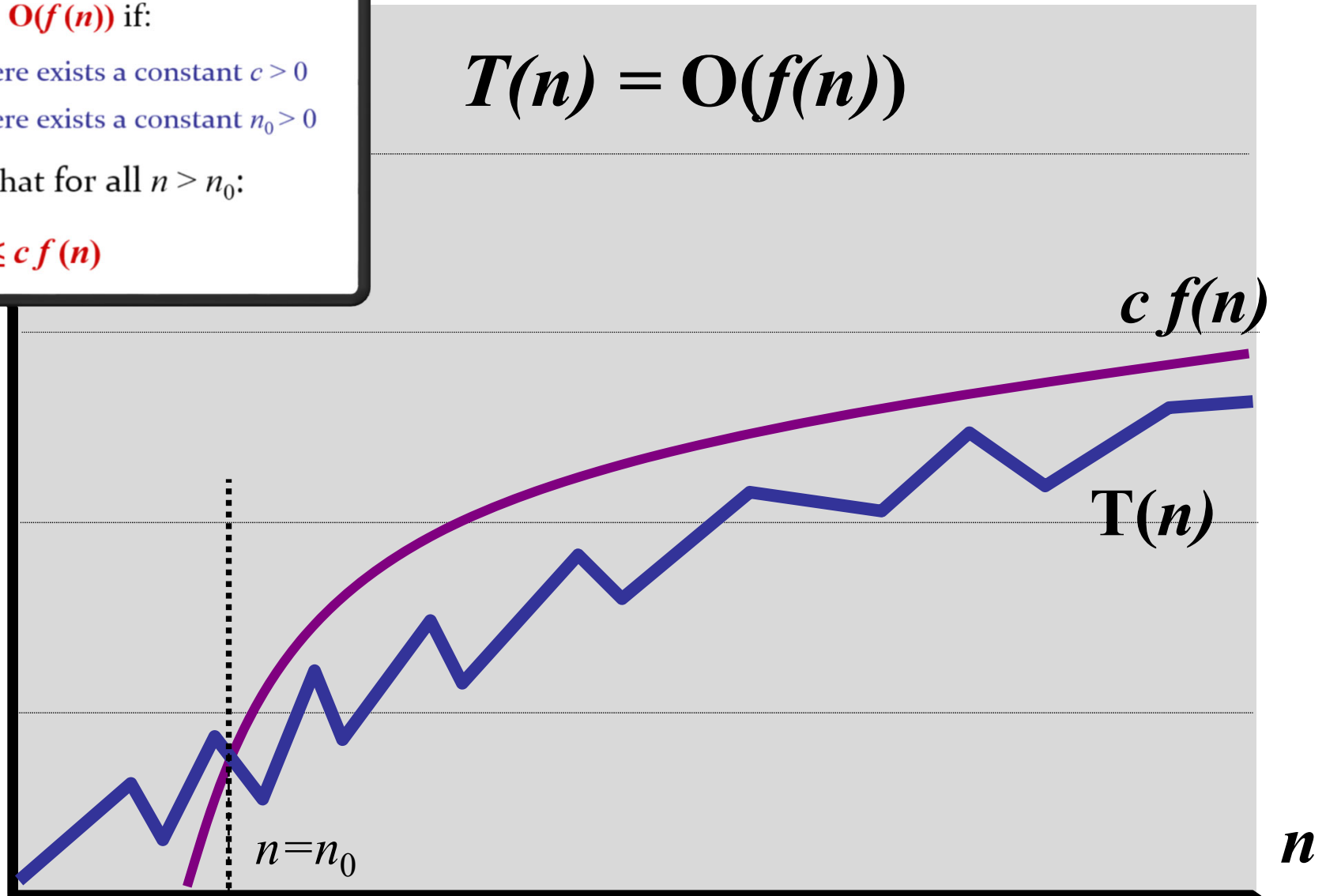
# Big-O Notation

$T(n) = O(f(n))$  if:

- there exists a constant  $c > 0$
- there exists a constant  $n_0 > 0$

such that for all  $n > n_0$ :

$$T(n) \leq c f(n)$$



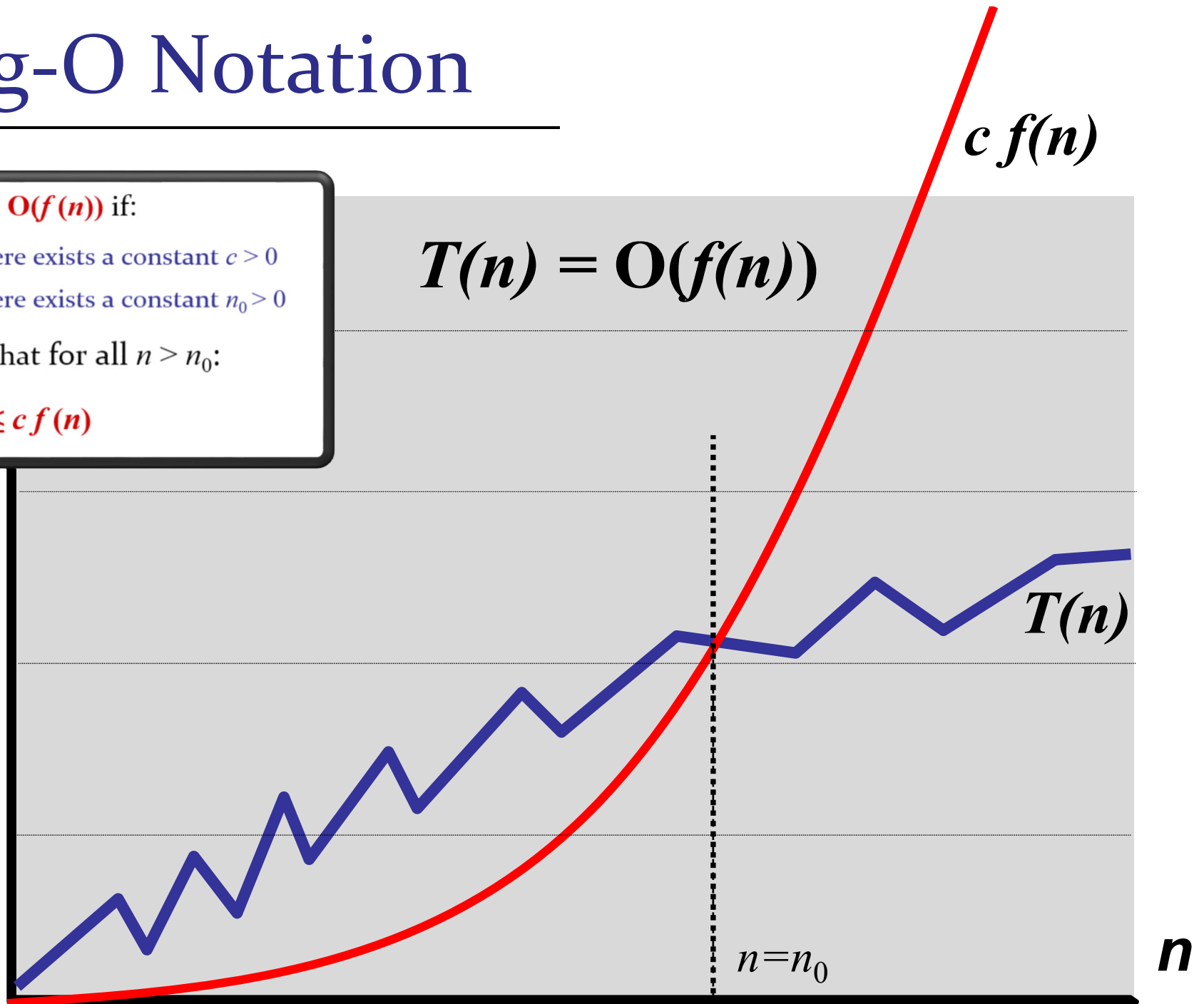
# Big-O Notation

$T(n) = O(f(n))$  if:

- there exists a constant  $c > 0$
- there exists a constant  $n_0 > 0$

such that for all  $n > n_0$ :

$$T(n) \leq c f(n)$$



# Big-O Notation

---

Example proof:  $T(n) = O(n^2)$

$$T(n) = 4n^2 + 24n + 16$$

# Example

$T(n)$

big-O

$$T(n) = 1000n$$

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

$$T(n) = 1000n$$

$$T(n) = O(n^2)$$

$$T(n) = n^2$$

$$T(n) \neq O(n)$$

Not  
tight

$$T(n) = 13n^2 + n$$

$$T(n) = O(n^2)$$



# Big-O Notation

---

Definition:  $T(n) = O(f(n))$  if  $T$  grows no faster than  $f$

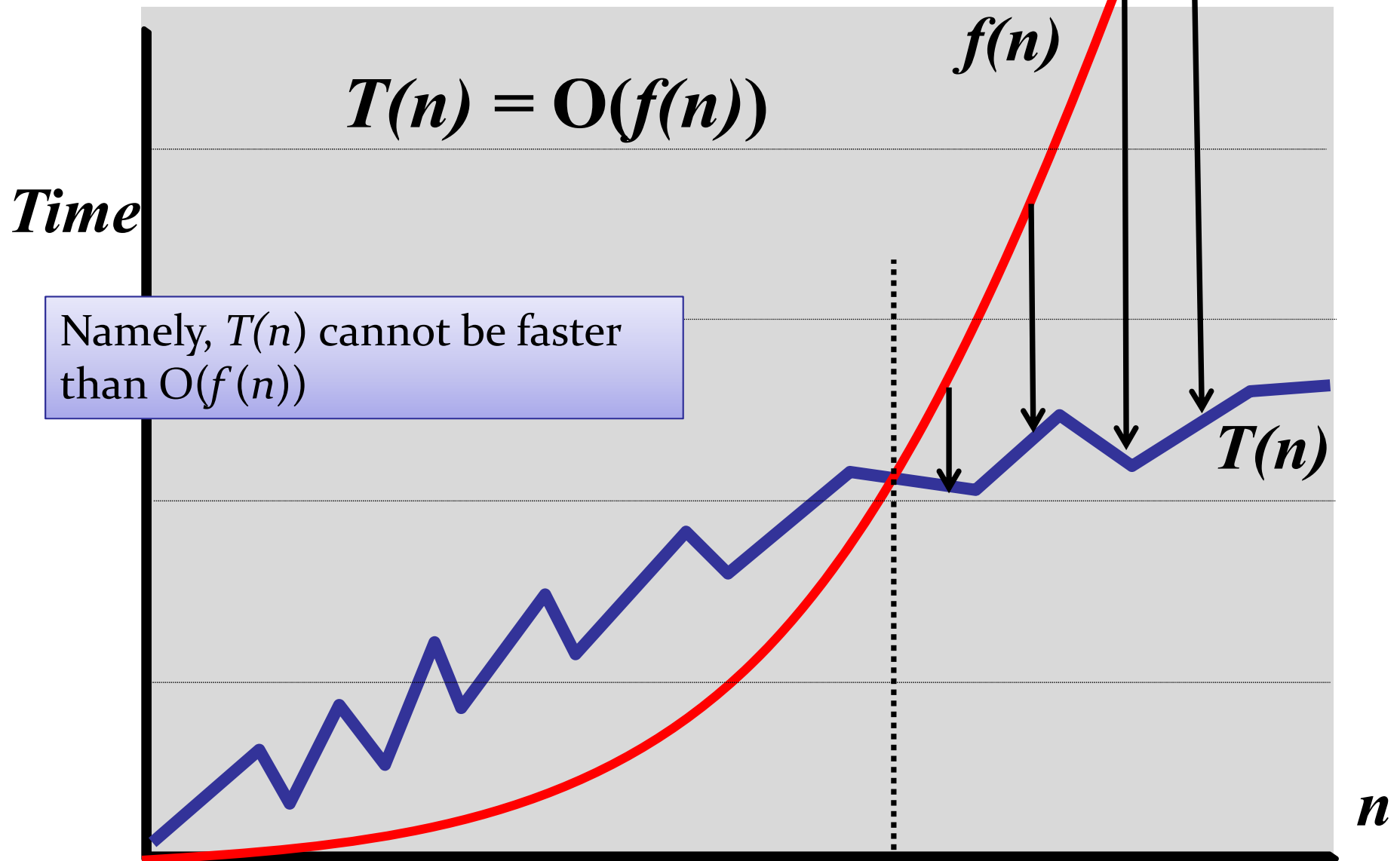
**$T(n) = O(f(n))$**  if:

- there exists a constant  $c > 0$
- there exists a constant  $n_0 > 0$

such that for all  $n > n_0$ :

$$\mathbf{T(n) \leq c f(n)}$$

# Big-O Notation as Upper Bound



# How about Lower bound?

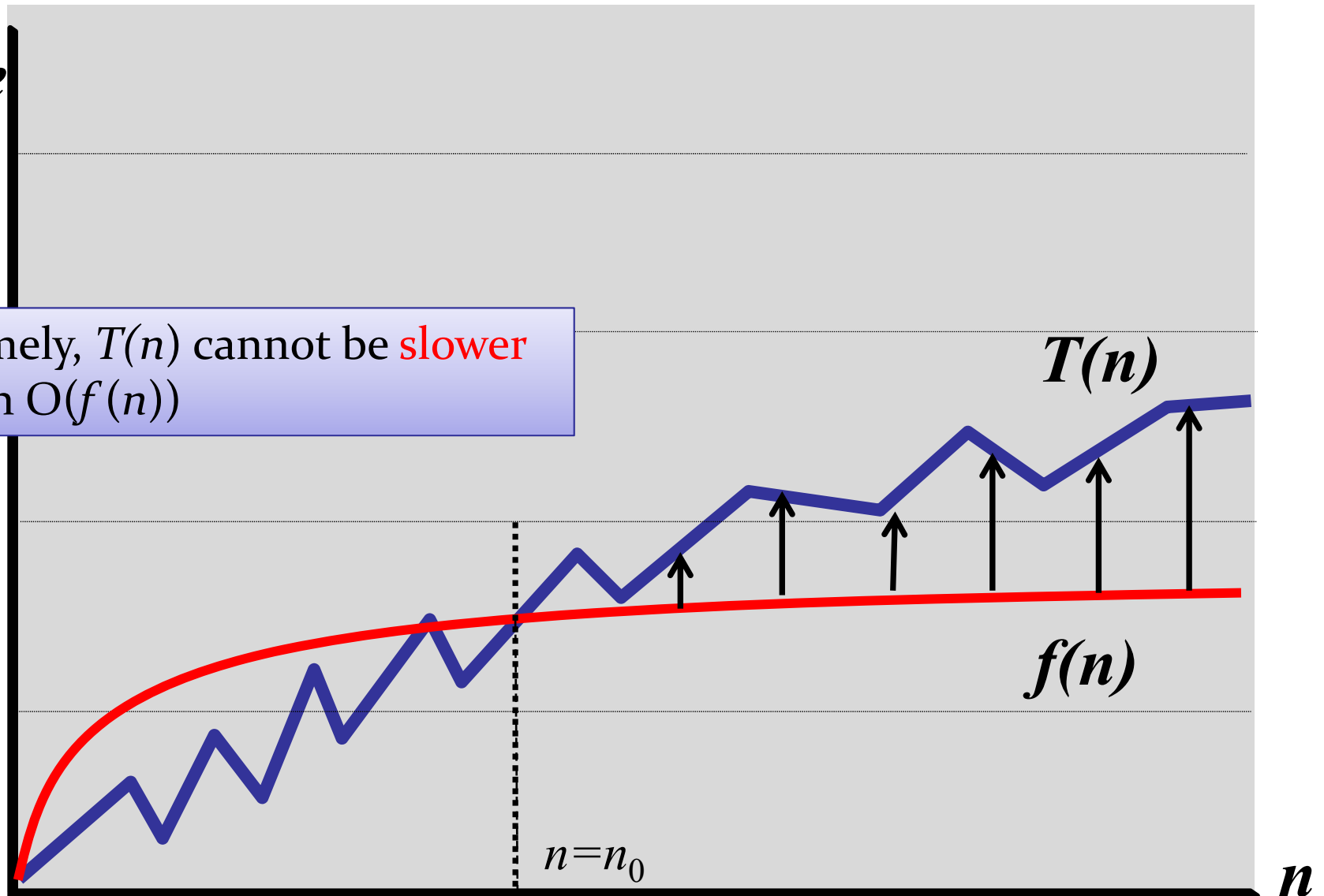
---

# How about Lower bound?

$$\begin{aligned} n_0 > 0 \\ c > 0 \\ n > n_0 \\ T(n) &\geq c f(n) \end{aligned}$$

*Time*

Namely,  $T(n)$  cannot be **slower** than  $O(f(n))$



# Big-O Notation

---

Definition:  $T(n) = \Omega(f(n))$  if  $T$  grows no slower than  $f$

$T(n) = \Omega(f(n))$  if:

- there exists a constant  $c > 0$
- there exists a constant  $n_0 > 0$

such that for all  $n > n_0$ :

$$T(n) \geq c f(n)$$

# Example

$T(n)$

Asymptotic

$$T(n) = 1000n$$

$$T(n) = \Omega(1)$$

$$T(n) = n$$

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

$$T(n) = n^2$$

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

$$T(n) = 13n^2 + n$$

$$T(n) = \Omega(n^2)$$

# Big-O Notation

---

Exercise:

True or false:

$f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$

Prove that your claim is correct using the definitions of  $O$  and  $\Omega$  or by giving an example.

# Big-O Notation

---

Definition:  $T(n) = \Theta(f(n))$  if  $T$  grows at the same rate as  $f$

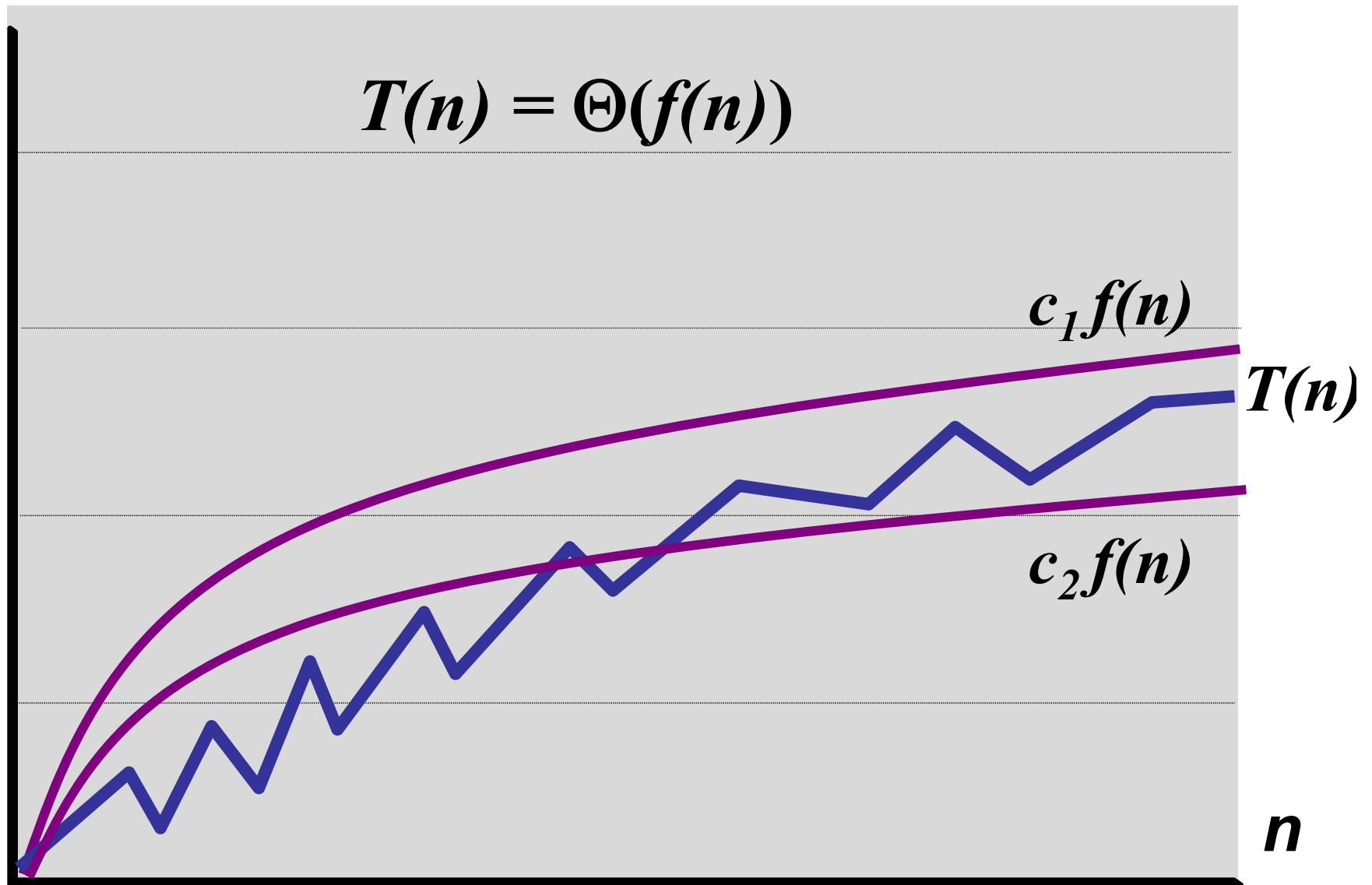
$T(n) = \Theta(f(n))$  if and only if:

- $T(n) = O(f(n))$  *and*
- $T(n) = \Omega(f(n))$



# Big-O Notation

---



# Example

$T(n)$

big-O

$$T(n) = 1000n$$

$$T(n) = \Theta(n)$$

$$T(n) = n$$

$$T(n) \neq \Theta(1)$$

$$T(n) = 13n^2 + n$$

$$T(n) = \Theta(n^2)$$

$$T(n) = n^3$$

$$T(n) \neq \Theta(n^2)$$

# Big-O Notation

---

Some simple rules for most cases...

# Big-O Notation

---

Order or size:

Function	Name
5	Constant
$\log\log(n)$	double log
$\log(n)$	logarithmic
$\log^2(n)$	Polylogarithmic
$n$	linear
$n\log(n)$	log-linear
$n^3$	polynomial
$n^3\log(n)$	
$n^4$	polynomial
$2^n$	exponential
$2^{2n}$	
$n!$	factorial

# Big-O Notation

---

Rules:

If  $T(n)$  is a polynomial of degree  $k$  then:

$$T(n) = O(n^k)$$

Example:

$$10n^5 + 50n^3 + 10n + 17 = O(n^5)$$

# Big-O Notation

---

Rules:

If  $T(n) = O(f(n))$  and  $S(n) = O(g(n))$  then

$$T(n) + S(n) = O(f(n) + g(n))$$

Example:

$$10n^2 = O(n^2)$$

$$5n\log(n) = O(n\log(n))$$

$$10n^2 + 5n\log(n) = O(n^2 + n\log(n)) = O(n^2)$$

# Big-O Notation

---

Rules:

If  $T(n) = O(f(n))$  and  $S(n) = O(g(n))$  then:

$$T(n) * S(n) = O(f(n) * g(n))$$

Example:

$$10n^2 = O(n^2)$$

$$5n = O(n)$$

$$(10n^2)(5n) = 50n^3 = O(n * n^2) = O(n^3)$$

$$n^4 + 3n^2 + n^2 + 17 = ?$$

- A.  $O(1)$
- B.  $O(n)$
- C.  $O(n^2)$
- D.  $O(n^3)$
- E.  $O(n^4)$



Why don't you try a few?

**ARCHIPELAGO**

is open

$$4n^2\log(n) + 8n + 16 = ?$$

1.  $O(\log n)$
2.  $O(n)$
3.  $O(n\log n)$
4.  $O(n^2\log n)$
5.  $O(2^n)$

$$2^{2n} + 2^n + 2 =$$

1.  $O(n)$
2.  $O(n^6)$
3.  $O(2^n)$
4.  $O(2^{2n})$
5.  $O(n^n)$

... x( )

$$\log(n!) = \log(n!)$$

1.  $O(\log n)$
2.  $O(n)$
3.  $O(n \log n)$
4.  $O(n^2)$
5.  $O(2^n)$

$$\log(n!) =$$

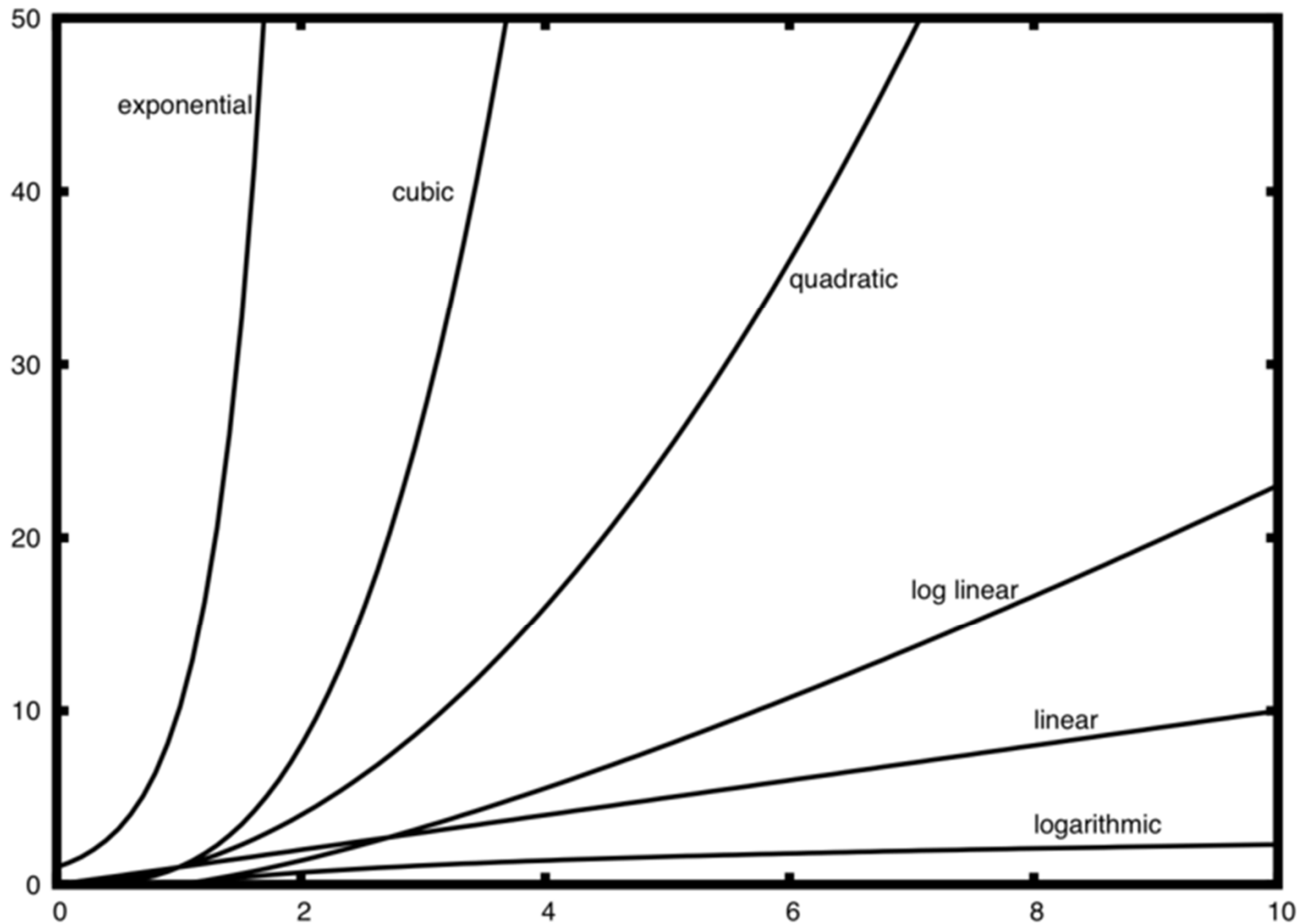
1.  $O(\log n)$
2.  $O(n)$
3.  $O(n \log n)$
4.  $O(n^2)$
5.  $O(2^n)$

Hint: Sterling's Approximation

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

# In General

---



# Model of Computation?

---

Different ways to “compute”:

- Sequential (RAM) model of computation
- Parallel (PRAM, BSP, Map-Reduce)
- Circuits
- Turing Machine
- Counter machine
- Word RAM model
- Quantum computation
- Etc.

# Model of Computation

---

## Sequential Computer

- One thing at a time
- All operations take constant time  
Addition, subtraction, multiplication, comparison



# Algorithm Analysis

---

## Example:

```
void sum(int k, int[] intArray) {
```

```
    int total=0;
```

```
    for (int i=0; i<= k; i++){
```

```
        total = total + intArray[i];
```

```
    }
```

```
    return total;
```

```
}
```

1 assignment

1 assignment

k+1 comparisons

k increments

k array access

k addition

k assignment

1 return

Total:  $1 + 1 + (k+1) + 3k + 1 = 4k+4 = O(k)$

# Algorithm Analysis

---

What is the cost of this operation?

Example:

```
void sum(int k, int[] intArray) {  
    int total=0;  
    String name="Stephanie";  
    for (int i=0; i<= k; i++){  
        total = total + intArray[i];  
        name = name + "?"  
    }  
    return total;  
}
```

Not 1!  
Not constant!  
Not k!

Moral: all costs are not 1.


# Rules

---

## Loops

$\text{cost} = (\# \text{ iterations}) \times (\text{max cost of one iteration})$

```
int sum(int k, int[] intArray) {  
    int total=0;  
    for (int i=0; i<= k; i++) {  
        total = total + intArray[i];  
    }  
    return total;  
}
```



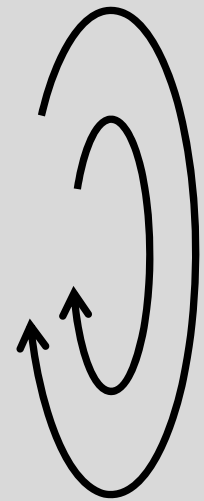
# Rules

---

## Nested Loops

$\text{cost} = (\# \text{ iterations})(\text{max cost of one iteration})$

```
int sum(int k, int[] intArray) {  
    int total=0;  
    for (int i=0; i<= k; i++){  
        for (int j=0; j<= k; j++){  
            total = total + intArray[i];  
        }  
    }  
    return total;  
}
```



# Rules

---

## Sequential statements

$\text{cost} = (\text{cost of first}) + (\text{cost of second})$

```
int sum(int k, int[] intArray) {  
    for (int i=0; i<= k; i++)  
        intArray[i] = k;  
    for (int j =0; j<= k; j++)  
        total = total + intArray[i];  
    return total;  
}
```

# Rules

---

if / else statements

cost = max(cost of first, cost of second)

$\leq$  (cost of first) + (cost of second)

```
void sum(int k, int[] intArray) {  
    if (k > 100)  
        doExpensiveOperation();  
    else  
        doCheapOperation();  
    return;  
}
```

# Rules

---

For recursive function calls.....



# Recurrences

$$T(n) = 1 + T(n-1) + T(n-2)$$
$$= O(2^n)$$


$$T(n) = 1 + T(n-1) + T(n-2)$$
$$= 1 + (1 + T(n-2) + T(n-3))$$

$$+ (1 + T(n-3) + T(n-4))$$
$$= 3 + T(n-2) + 2T(n-3) + T(n-4)$$

```
int fib(int n) {  
    if (n <= 1)  
        return n;  
    else  
        return fib(n-1) + fib(n-2);  
}
```



What is the running time?



```
for (int i = 0; i < n; i++)  
    for (int j = 0; j < i; j++)  
        store[i] = i + j;
```

$0 + 1 + 2 + \dots + n$

1.  $O(1)$
2.  $O(n)$
3.  $O(n \log n)$
4.  $O(n^2)$
5.  $O(n^2 \log n)$
6.  $O(2^n)$

# Today: Divide and Conquer!

---

## Algorithm Analysis

- Big-O Notation
- Model of computation

## Searching

## Peak Finding

- 1-dimension
- 2-dimensions

# Binary Search

---

**Sorted** array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for  $k$  in array  $A$ .

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element: 7

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element: 7
- Compare 17 to middle element:  $17 > 7$

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element: 7
- Compare 17 to middle element:  $17 > 7$

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element: 7
- Compare 17 to middle element:  $17 > 7$
- Recurse on right half



# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element
- Compare 17 to middle element
- Recurse

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element
- Compare 17 to middle element
- Recurse

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element
- Compare 17 to middle element
- Recurse

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element
- Compare 17 to middle element
- Recurse

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Search for 17 in array A.

- Find middle element
- Compare 17 to middle element
- Recurse

# Problem Solving: Reduce the Problem

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Reduce-and-Conquer:

- Start with  $n$  elements to search.
- Eliminate half of them.
- End with  $n/2$  elements to search.
- Repeat.

# programming pearls

By Jon Bentley

---

## WRITING CORRECT PROGRAMS

### The Challenge of Binary Search

Even with the best of designs, every now and then a programmer has to write subtle code. This column is about one problem that requires particularly careful code: binary search. After defining the problem and sketching an algorithm to solve it, we'll use principles of program verification in several stages as we develop the program.

Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now, and writing the code yourself. Try it.



Jon Bentley

# programming pearls

By Jon Bentley

---

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudo-code was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.



Jon Bentley



# Binary Search (buggy)

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
Search(A, key, n)
```

```
    begin = 0
```

```
    end = n
```

```
    while begin != end do:
```

```
        if key < A[(begin+end)/2] then
```

```
            end = (begin+end)/2 - 1
```

```
        else begin = (begin+end)/2
```

```
    return A[begin]
```

# Bug 1

---

Sorted array:  $A[0..n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
Search(A, key, n)
```

```
    begin = 0
```

```
    end = n
```

```
    while begin != end do:
```

```
        if key < A[(begin+end)/2] then
```

```
            end = (begin+end)/2 - 1
```

```
        else begin = (begin+end)/2
```

```
    return A[end]
```

array out of bounds!



# Bug 1

---

Sorted array:  $A[0..n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
Search(A, key, n)
```

```
begin = 0
```

```
end = n
```

```
while begin != end do:
```

```
    if key < A[(begin+end)/2] then
```

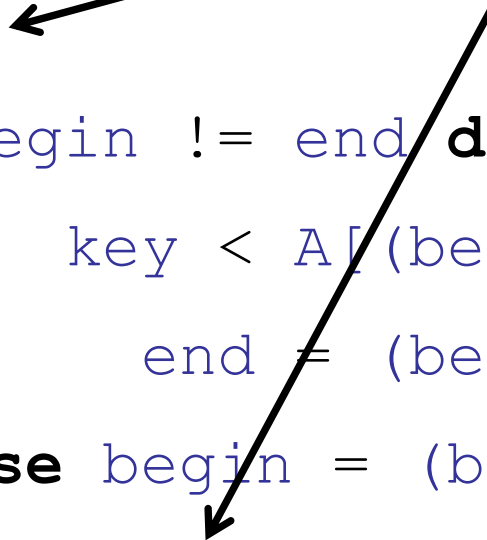
```
        end = (begin+end)/2 - 1
```

```
    else begin = (begin+end)/2
```

```
return A[end]
```

**array out of bounds!**

(Can't happen because of other bugs...)



# Bug 1

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
Search(A, key, n)
```

```
    begin = 0
```

```
    end = n-1
```

```
    while begin != end do:
```

```
        if key < A[(begin+end)/2] then
```

```
            end = (begin+end)/2 - 1
```

```
        else begin = (begin+end)/2
```

```
    return A[end]
```

# Bug 2

Sorted array

2	4	4
---	---	---

5	10
---	----

Example: search(7)

- begin = 0, end = 1
- mid =  $(0+1)/2 = 0$
- key  $\geq A[\text{mid}] \rightarrow \text{begin} = 0$

Search(A, k)

```
begin = 0
```

```
end = n-1
```

```
while begin != end do:
```

```
    if key < A[(begin+end)/2] then
```

```
        end = (begin+end)/2 - 1
```

```
    else begin = (begin+end)/2
```

```
return A[end]
```

May not terminate!

round down

# Bug 2

Sorted array

2	4	4
---	---	---

5	10
---	----

Example: search(2)

- begin = 0, end = 1
- mid =  $(0+1)/2 = 0$
- $\text{key} < A[\text{mid}] \rightarrow \text{end} = 0 - 1 = -1$

Search(A, k)

```
begin = 0
```

```
end = n-1
```

```
while begin != end do:
```

```
    if key < A[(begin+end)/2] then
```

```
        end = (begin+end)/2 - 1
```

```
    else begin = (begin+end)/2
```

```
return A[end]
```

**end < begin**

**subtract?**

*may cause  
index  
out of  
bound*

# Bug 3

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
Search(A, key, n)
```

```
    begin = 0
```

```
    end = n-1
```

```
    while begin != end do:
```

```
        if key < A[(begin+end)/2] then
```

```
            end = (begin+end)/2 - 1
```

```
        else begin = (begin+end)/2
```

```
    return A[end] ← Useful return value?
```

# Binary Search

---

## Specification:

- Returns element if it is in the array.
- Returns “null” if it is not in the array.

## Alternate Specification:

- Returns index if it is in the array.
- Returns -1 if it is not in the array.



# Binary Search

Sorted array:  $A[0 \dots n-1]$

0	1	2	3	4	5	6	7	8	9	10	11
2	4	4	5	6	7	8	9	11	17	23	28

```
int search(A, 4key, n)
```

```
    begin = 0
```

```
    end = n-1
```

```
    while begin < end do:
```

```
        if key <= A[(begin+end)/2] then
```

```
            end = (begin+end)/2
```

```
        else begin = 1+(begin+end)/2
```

```
    return (A[begin]==key) ? begin : -1
```

beg=0  
end=11  
mid=5

beg=0  
end=5  
mid=2

beg=0  
end=2  
mid=1

beg=0  
end=1  
mid=0

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
```

```
    begin = 0
```

```
    end = n-1
```

```
    while begin < end do:
```

```
        if key <= A[(begin+end)/2] then
```

```
            end = (begin+end)/2
```

```
        else begin = 1+(begin+end)/2
```

```
    return (A[begin]==key) ? begin : -1
```

less-than-or-equal



# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
```

```
    begin = 0
```

```
    end = n-1
```

```
    while begin < end do:
```

```
        if key <= A[(begin+end)/2] then
```

```
            end = (begin+end)/2
```

```
        else begin = 1+(begin+end)/2
```

```
    return (A[begin]==key) ? begin : -1
```

strictly greater than



# Binary Search

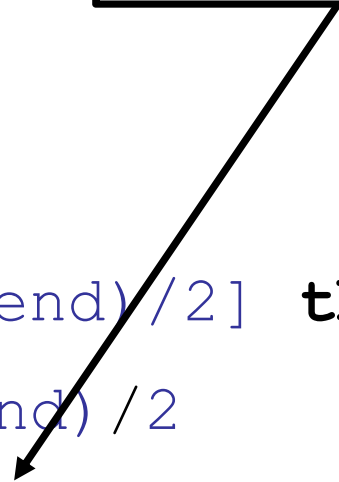
---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if key <= A[(begin+end)/2] then
            end = (begin+end) / 2
        else begin = 1+(begin+end) / 2
    return (A[begin]==key) ? begin : -1
```

Array of out  
bounds?



# Binary Search

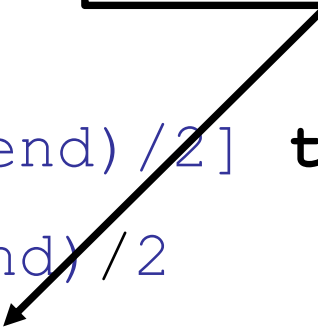
---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return (A[begin]==key) ? begin : -1
```

Array of out  
bounds?  
No: division  
rounds down.



# Bug 4

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key,  
    begin = 0  
    end = n-1
```

What if **begin** > MAX\_INT/2?

```
while begin < end do:  
    if key <= A[(begin+end)/2] then  
        end = (begin+end)/2  
    else begin = 1+(begin+end)/2  
return (A[begin]==key) ? begin : -1
```

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key,  
    begin = 0  
    end = n-1
```

```
    while begin < end do:
```

```
        if key <= A[(begin+end)/2] then
```

```
            end = (begin+end)/2
```

```
        else begin = 1+(begin+end)/2
```

```
    return (A[begin]==key) ? begin : -1
```

What if  $\text{begin} > \text{MAX\_INT}/2$ ?

Overflow error:  $\text{begin} + \text{end} > \text{MAX\_INT}$

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```



# Moral of the Story

Easy algorithms are \*hard\* to write correctly.

Binary search is 8 lines of code.

If you can't write 8 correct lines of code, how do you expect to write thousands of lines of bug-free code??

# Precondition and Postcondition

---

## Precondition:

- Fact that is true when the function begins.
- Usually important for it to work correctly.

## Postcondition:

- Fact that is true when the function ends.
- Usually useful to show that the computation was done correctly.

# Binary Search

ARCHIPELAGO

is open

Sorted array:  $A[0..n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

What are useful  
preconditions and  
postconditions?

# Binary Search

---

Functionality:

- If element is in the array, return index of element.
- If element is not in array, return -1.

# Binary Search

---

## Functionality:

- If element is in the array, return index of element.
- If element is not in array, return -1.

## Preconditions:

- Array is of size  $n$
- Array is sorted

# Binary Search

---

Functionality:

- If element is in the array, return index of element.
- If element is not in array, return -1.

Preconditions:

- Array is of size  $n$
- Array is sorted



You can usually check  
this directly.

# Binary Search

---

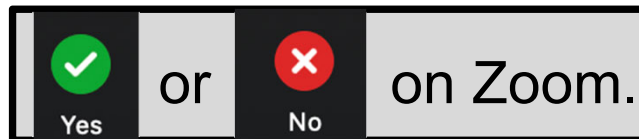
Functionality:

- If element is in the array, return index of element.
- If element is not in array, return -1.

Preconditions:

- Array is of size  $n$
- Array is sorted

Should we do input validation to make sure array is sorted??



# Binary Search

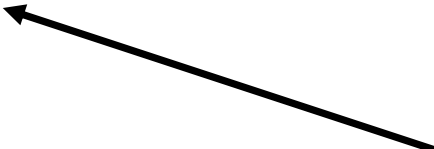
---

## Functionality:

- If element is in the array, return index of element.
- If element is not in array, return -1.

## Preconditions:

- Array is of size  $n$
- Array is sorted



Should we do input  
validation to make sure  
array is sorted??  
**NO! Too slow!**



# Binary Search

---

Functionality:

- If element is in the array, return index of element.
- If element is not in array, return -1.

Preconditions:

- Array is of size  $n$
- Array is sorted

Postcondition:

- If element is in the array:  $A[\text{begin}] = \text{key}$

# Invariants

---

Invariant:

- relationship between variables that is always true.

# Invariants

---

Invariant:

- relationship between variables that is always true.

Loop Invariant:

- relationship between variables that is true at the beginning (or end) of each iteration of a loop.

# Binary Search

ARCHIPELAGO

is open

Sorted array:  $A[0..n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

What are useful  
invariants?

# Binary Search

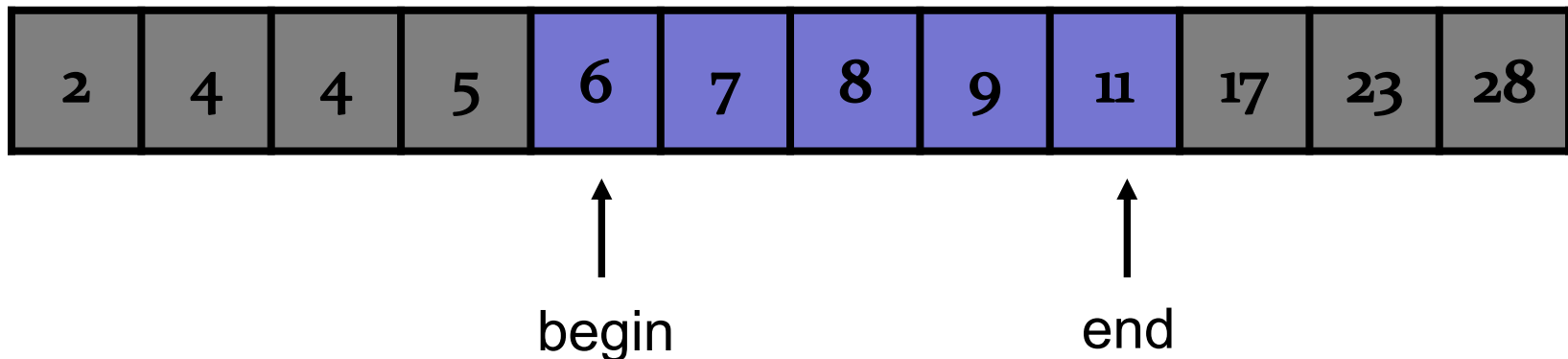
---

Loop invariant:

- $A[\text{begin}] \leq \text{key} \leq A[\text{end}]$

Interpretation:

- The key is in the range of the array



# Binary Search

---

Loop invariant:

- $A[\text{begin}] \leq \text{key} \leq A[\text{end}]$

Interpretation:

- The key is in the range of the array

Error checking:

```
if ((A[begin] > key) or (A[end] < key))  
    System.out.println("error");
```

# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

# Binary Search

---

$n$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

$n/2$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

$n/4$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

$n/8$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----



# Binary Search

---

Sorted array:  $A[0 \dots n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Iteration 1:  $(\text{end} - \text{begin}) = n$

Iteration 2:  $(\text{end} - \text{begin}) = n/2$

Iteration 3:  $(\text{end} - \text{begin}) = n/4$

...

Iteration  $k$ :  $(\text{end} - \text{begin}) = n/2^k$

Another invariant!

$$n/2^k = 1 \quad \rightarrow \quad k = \log(n)$$

# Key Invariants:

---

## Correctness:

- $A[\text{begin}] \leq \text{key} \leq A[\text{end}]$

## Performance:

- $(\text{end} - \text{begin}) \leq n/2^k$  in iteration  $k$ .

# Binary Search

---

Sorted array:  $A[0..n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Not just for searching arrays:

- Assume a complicated function:

`int complicatedFunction(int s)`

- Assume the function is always increasing:

`complicatedFunction(i) < complicatedFunction(i+1)`

- Find the minimum value  $j$  such that:

`complicatedFunction(j) > 100`

# A problem...

---

## Tutorial allocation

# A problem...

---

## Tutorial allocation

**Tutorials**  
(in order  
of tutor  
preference)

T<sub>1</sub>

T<sub>2</sub>

T<sub>3</sub>

T<sub>4</sub>

T<sub>5</sub>

T<sub>6</sub>

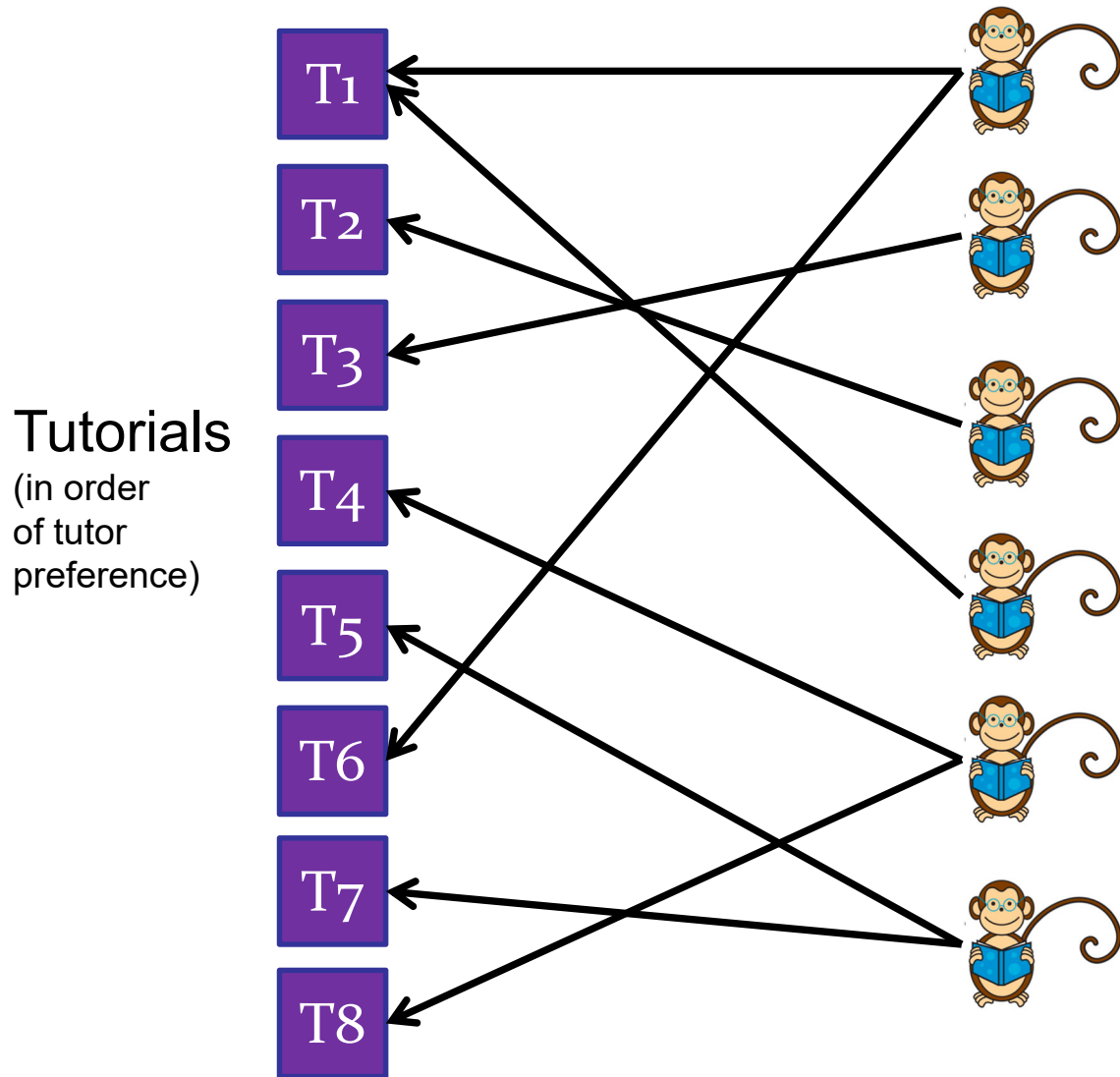
T<sub>7</sub>

T<sub>8</sub>

# A problem...

---

## Tutorial allocation

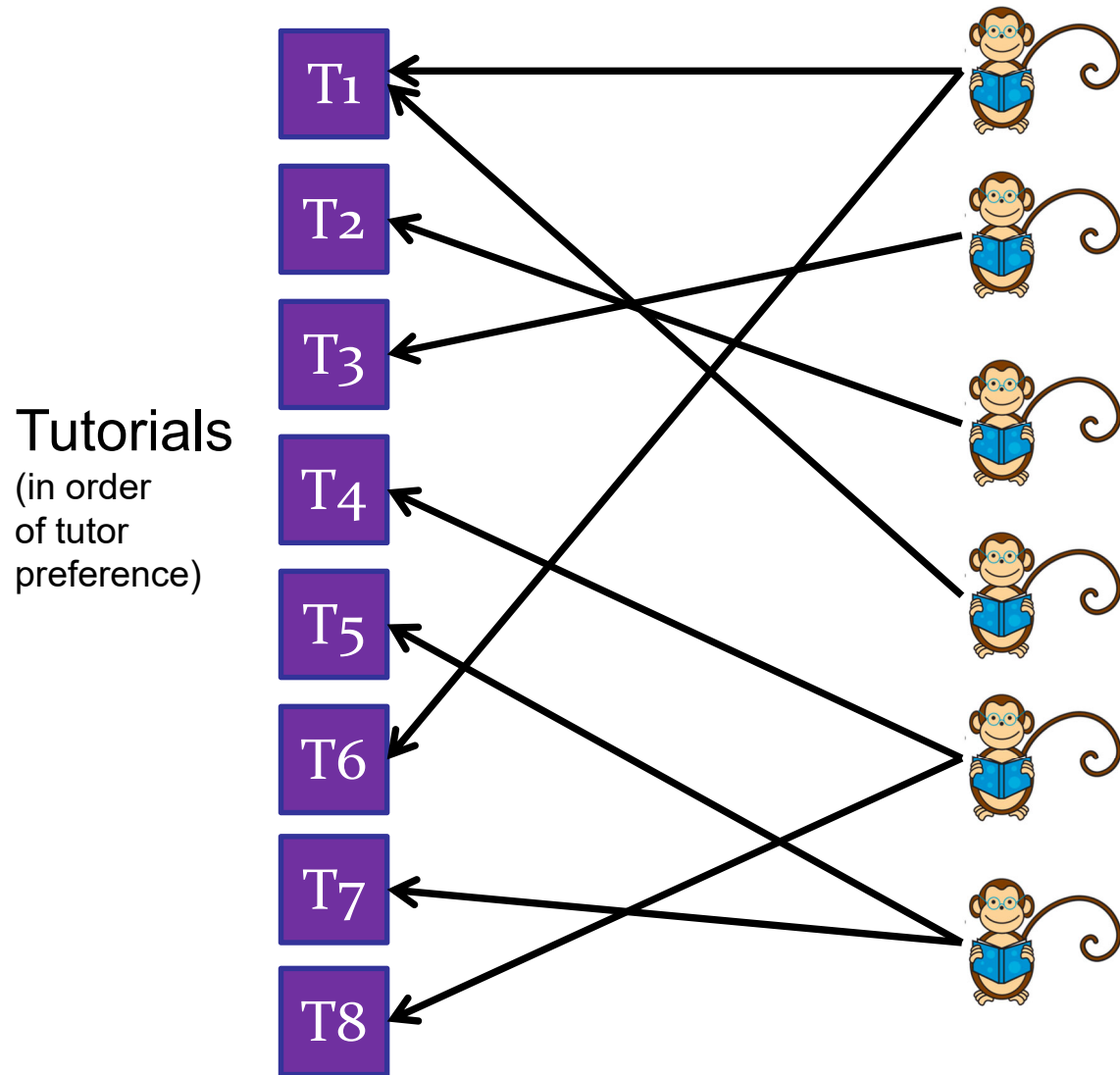


Students want  
certain tutorials.

# A problem...

---

## Tutorial allocation



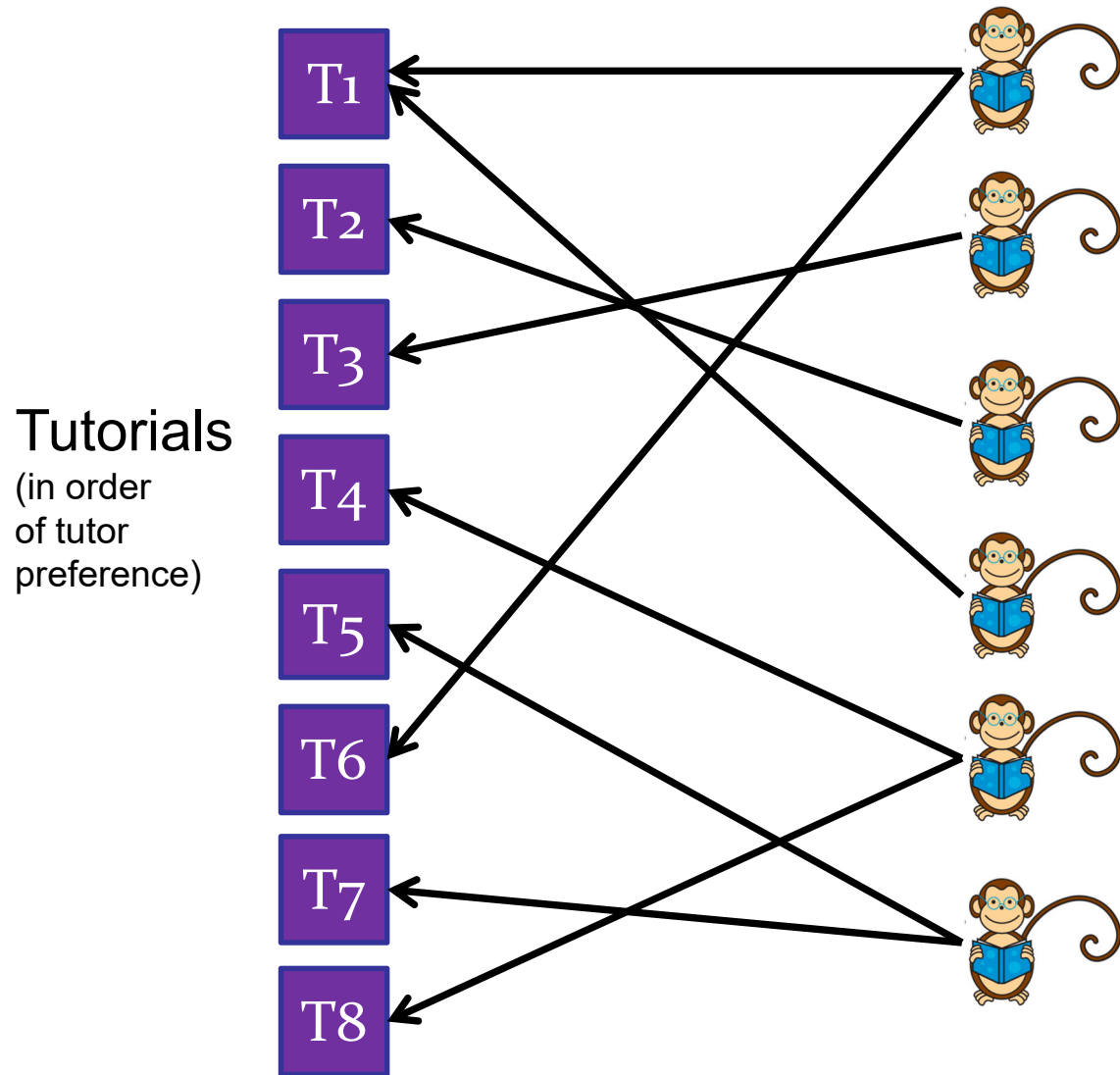
Students want  
certain tutorials.

We want each  
tutorial to have  
< 15 students..

# A problem...

---

## Tutorial allocation



Students want  
certain tutorials.

We want each  
tutorial to have  
< 15 students..

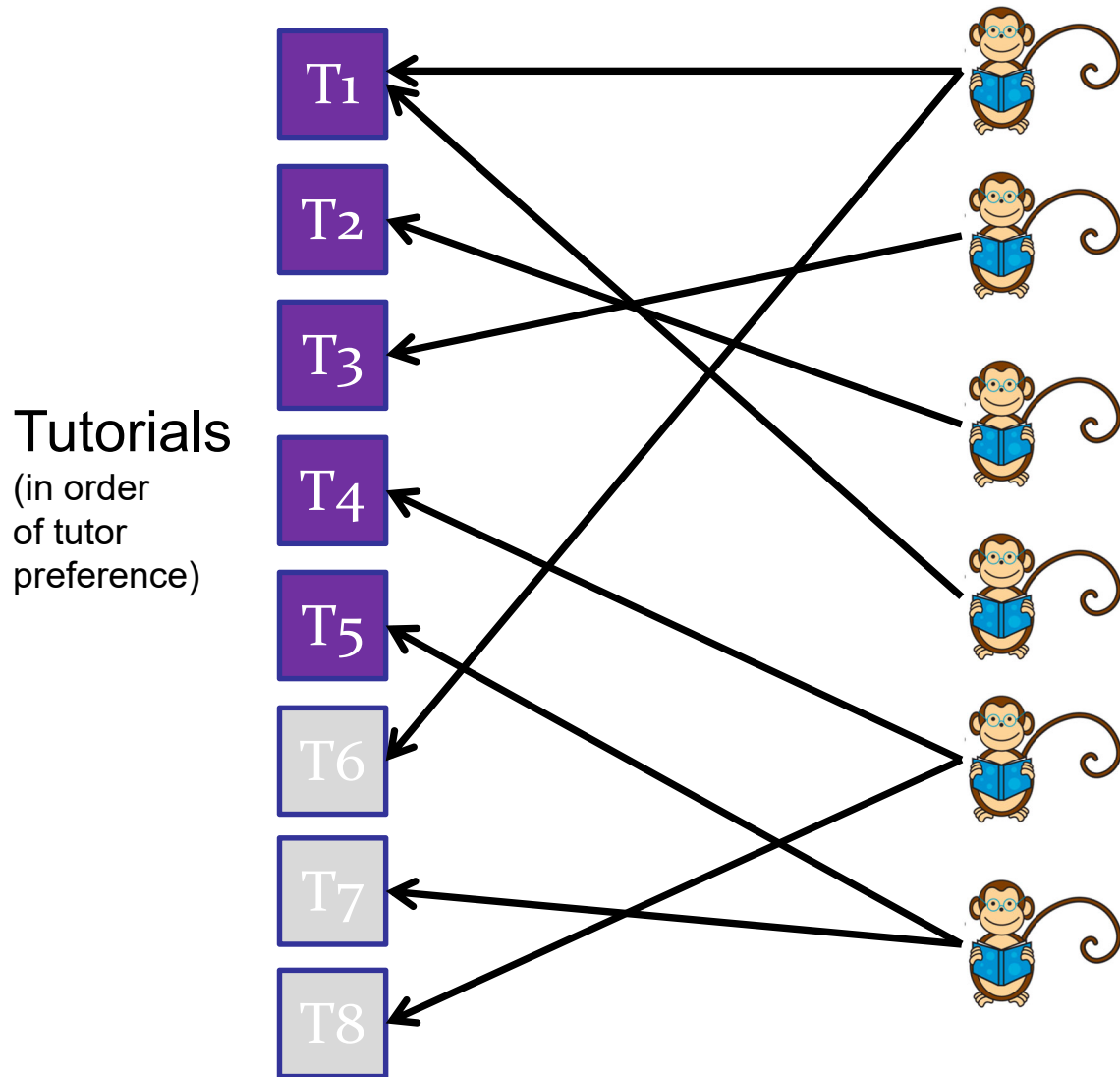
How many tutorials  
do we need to run?



# A problem...

---

## Tutorial allocation



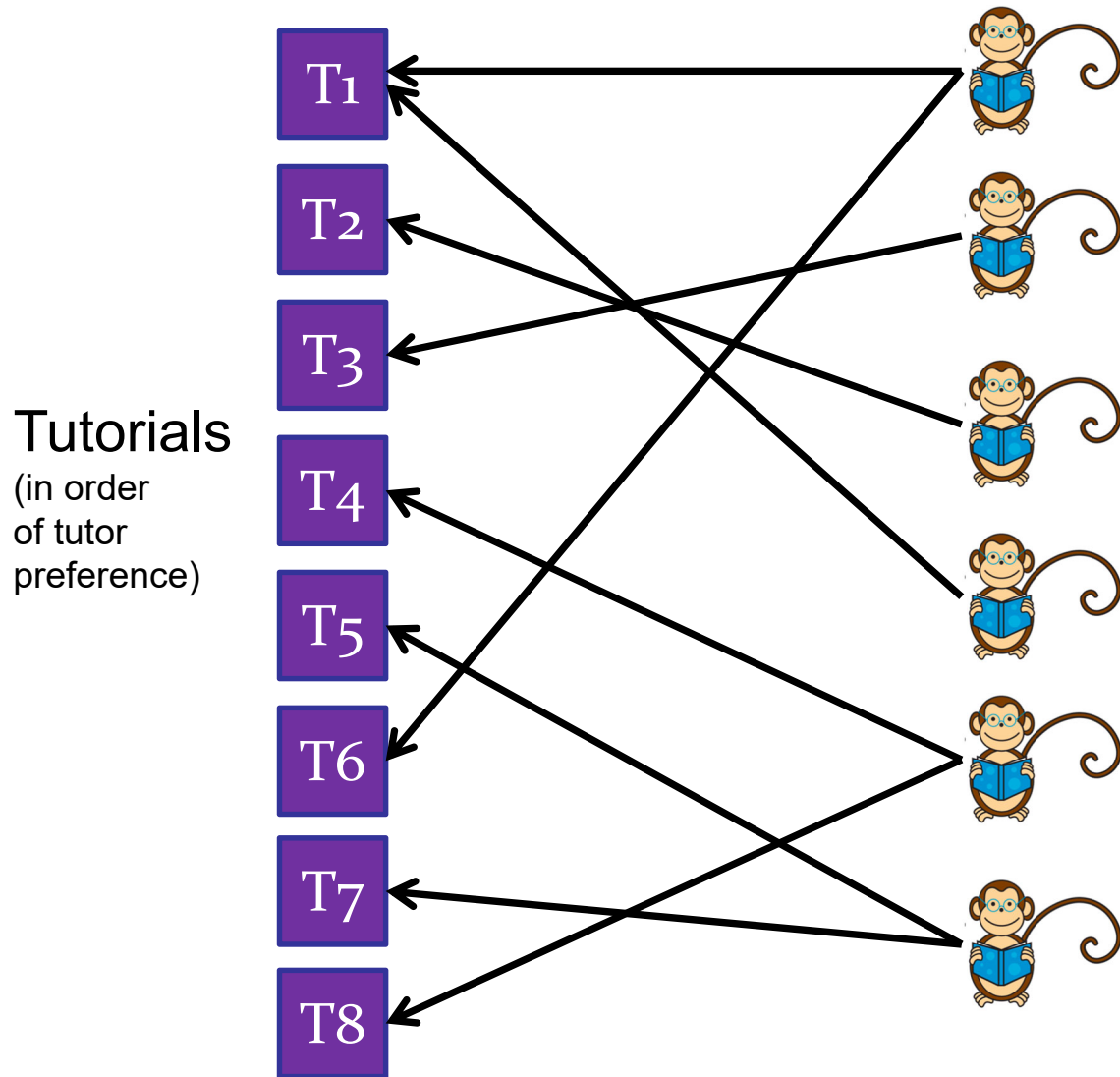
Students want  
certain tutorials.

We want each  
tutorial to have  
< 15 students..

How many tutorials  
do we need to run?

# A problem...

## Tutorial allocation

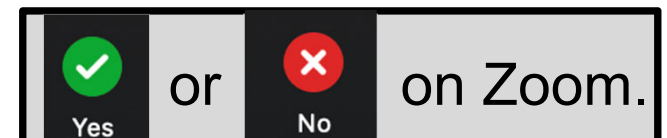


Can we do  
greedy allocation?

First, fill T1.  
Then fill T2.  
Then fill T3.

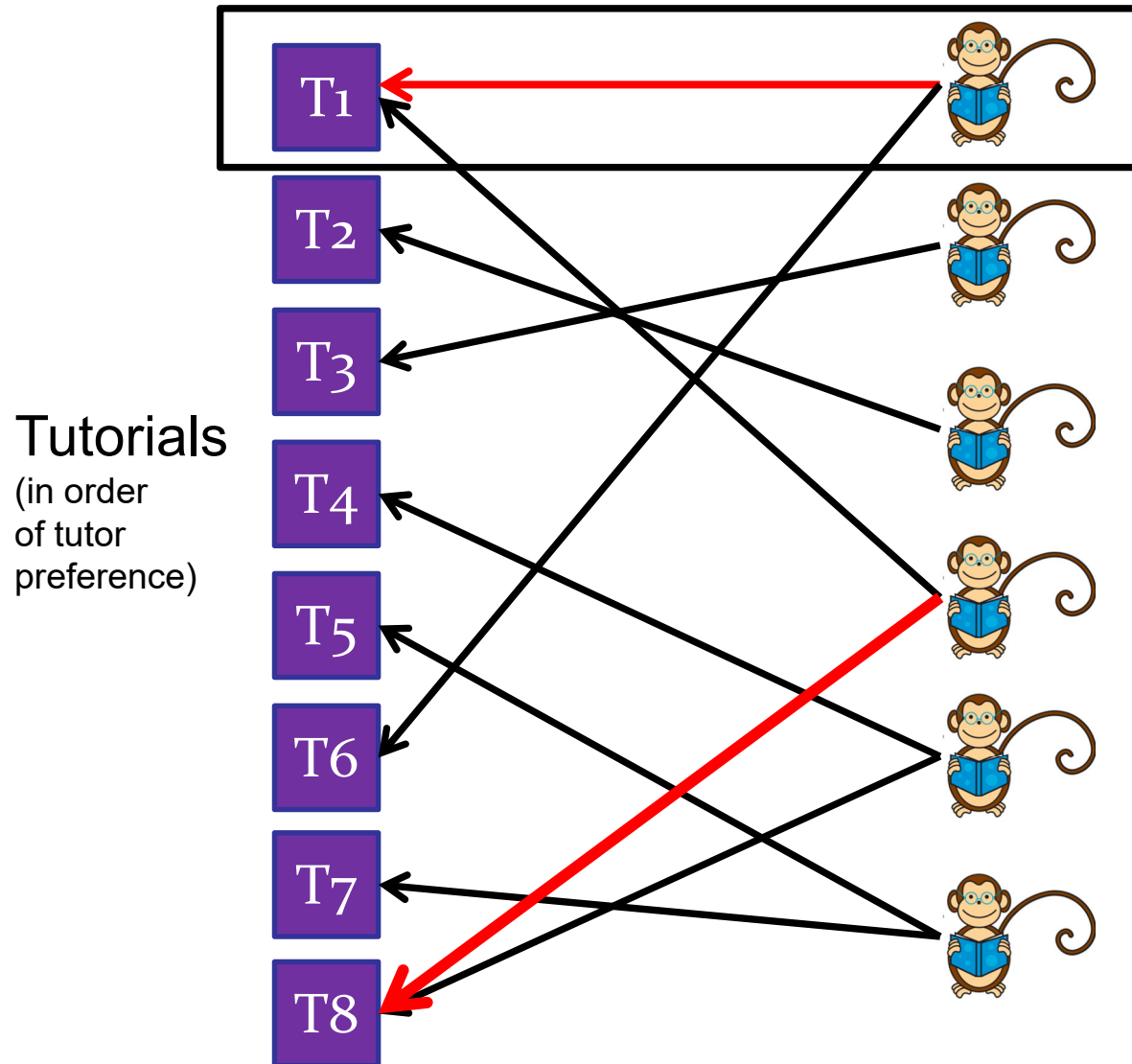
...

Stop when all  
students are  
allocated



# A problem...

## Tutorial allocation



Can we do  
greedy allocation?

NO

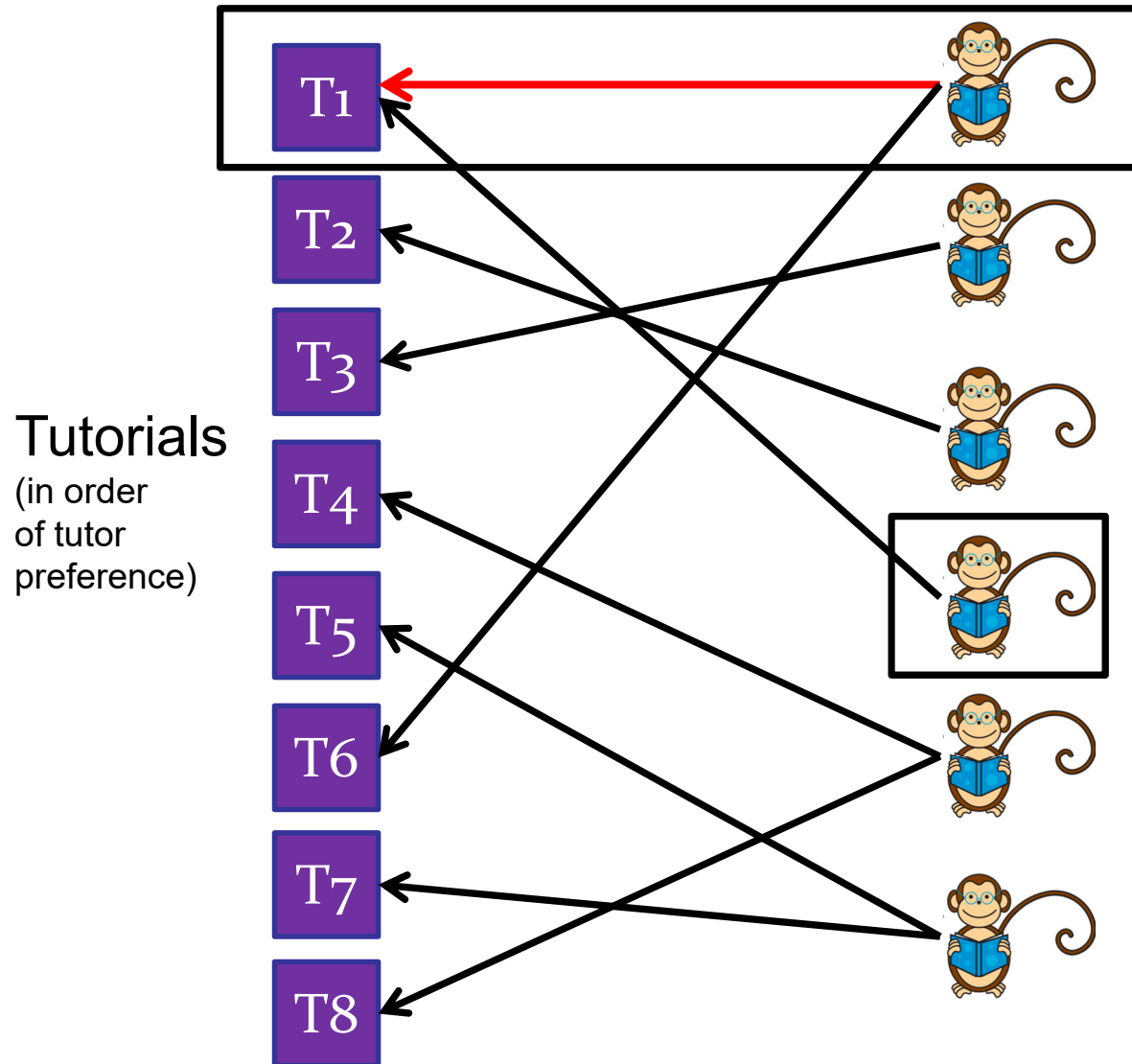
Assume max  
tutorial size is 1.

Assign student 1 to  
tutorial 1.

Now we need all 8  
tutorials.

# A problem...

## Tutorial allocation



Can we do  
greedy allocation?

NO

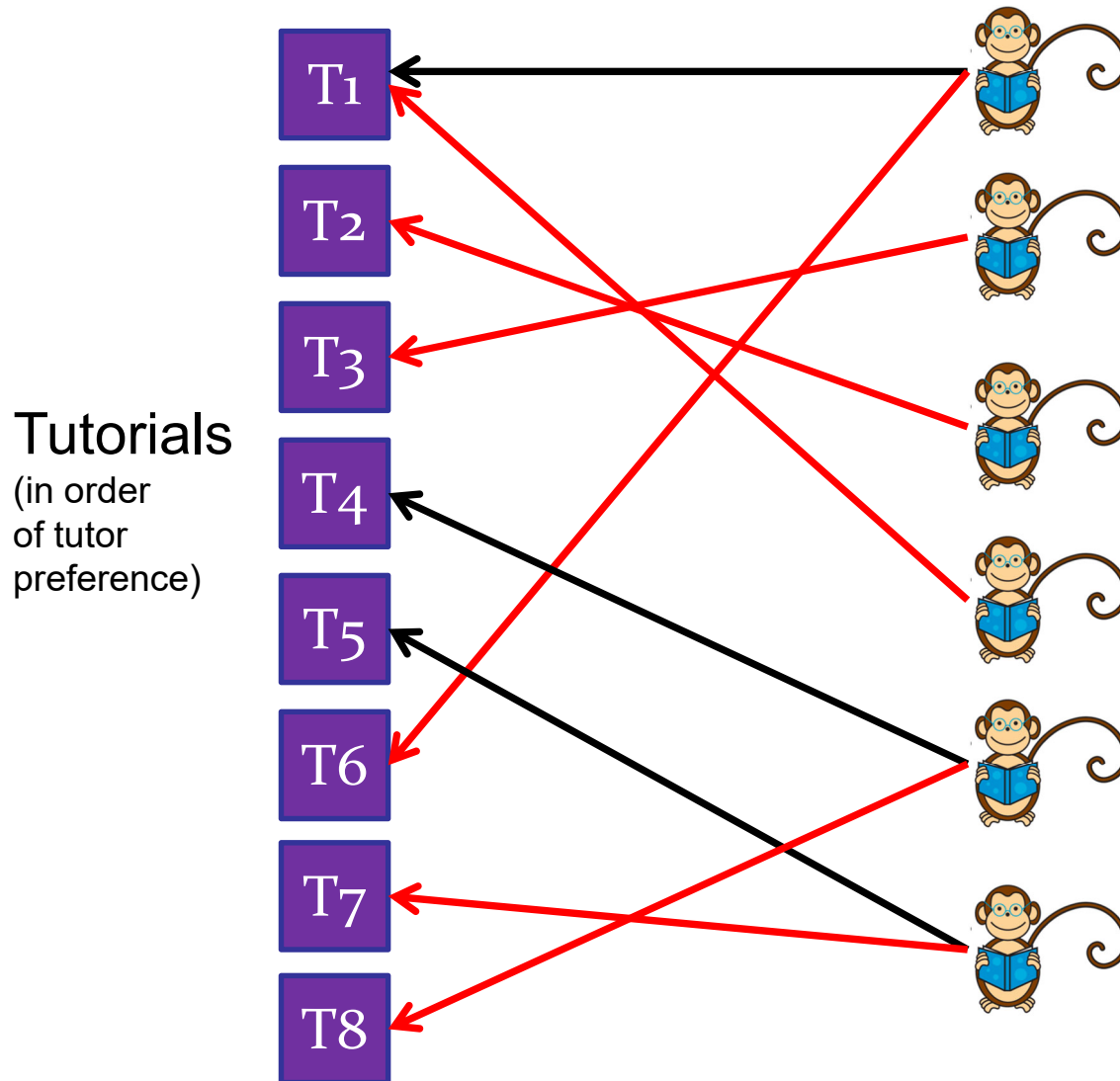
Assume max  
tutorial size is 1.

Assign student 1 to  
tutorial 1.

Now one student  
has no feasible  
allocation!

# A problem...

## Tutorial allocation



Assume we can solve allocation problem:

Given a fixed set of tutorials and a fixed set of students, find an allocation where every student has a slot.

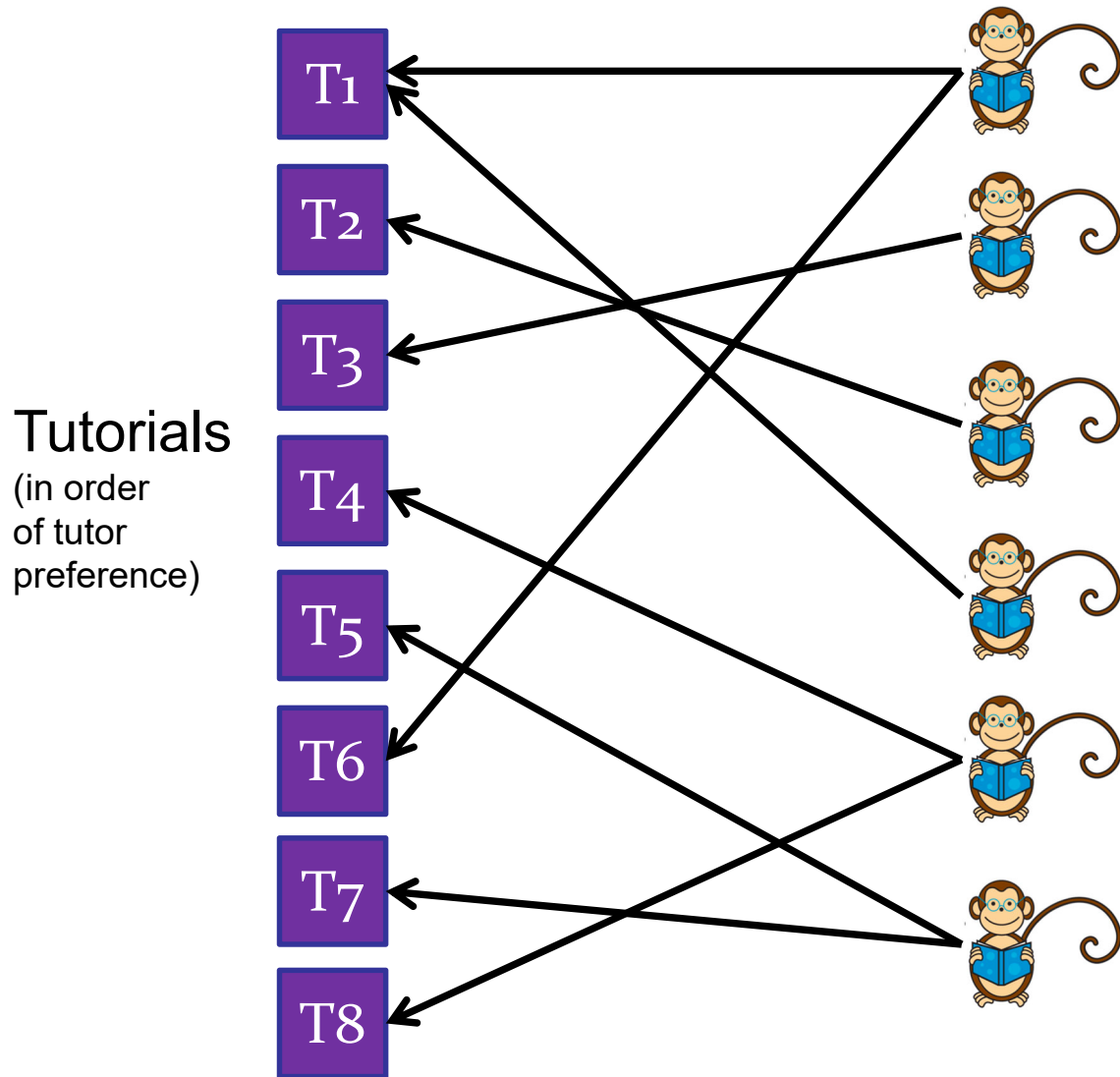
**Warning:**

- may be **> 15** students in a slot!
- minimizes max students in a slot.

# A problem...

---

## Tutorial allocation

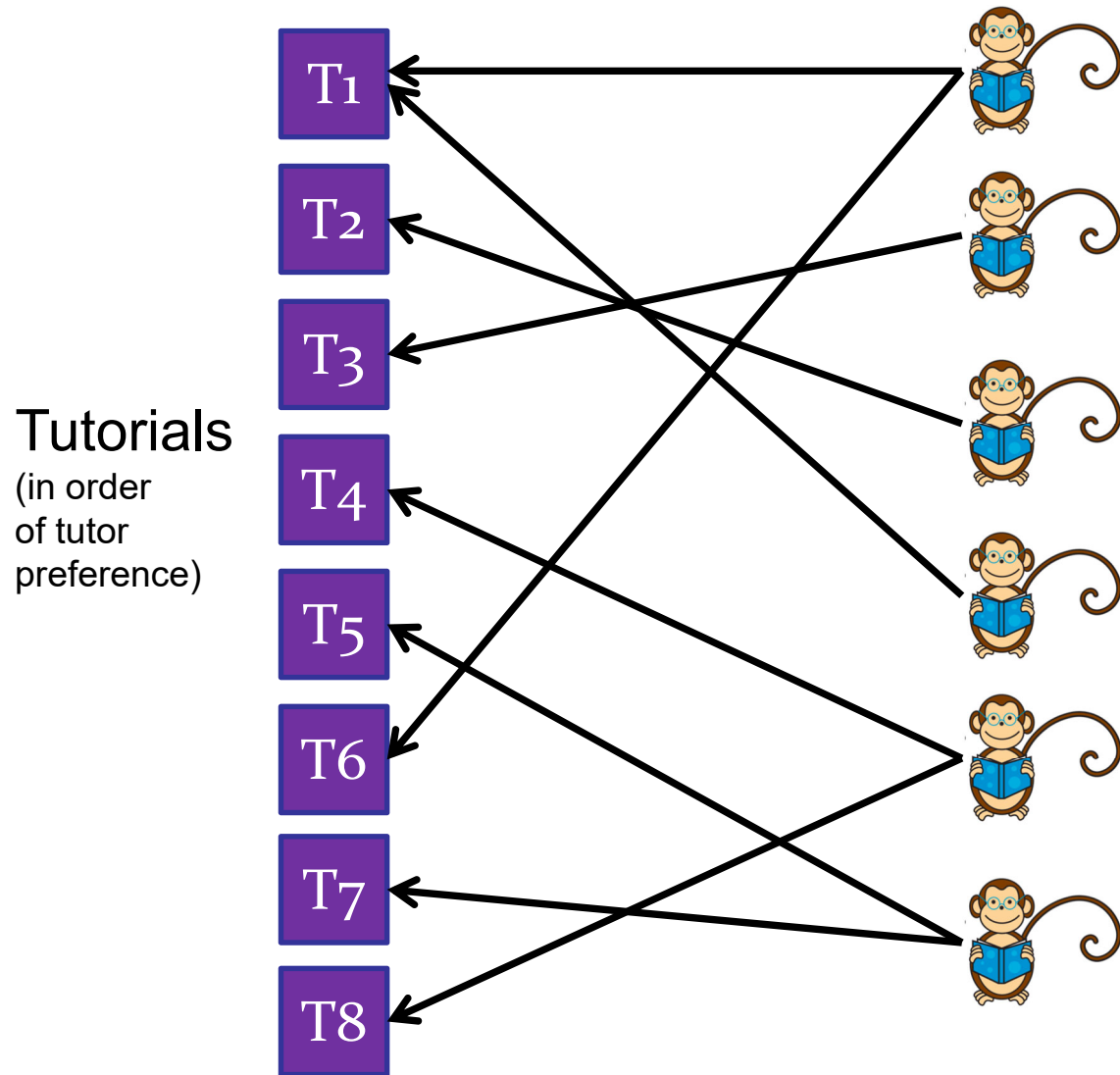


How to find  
minimum number  
of tutorials that we  
need to open to  
ensure: **no tutorial  
has more than 15  
students.**

# A problem...

---

## Tutorial allocation



Observation:

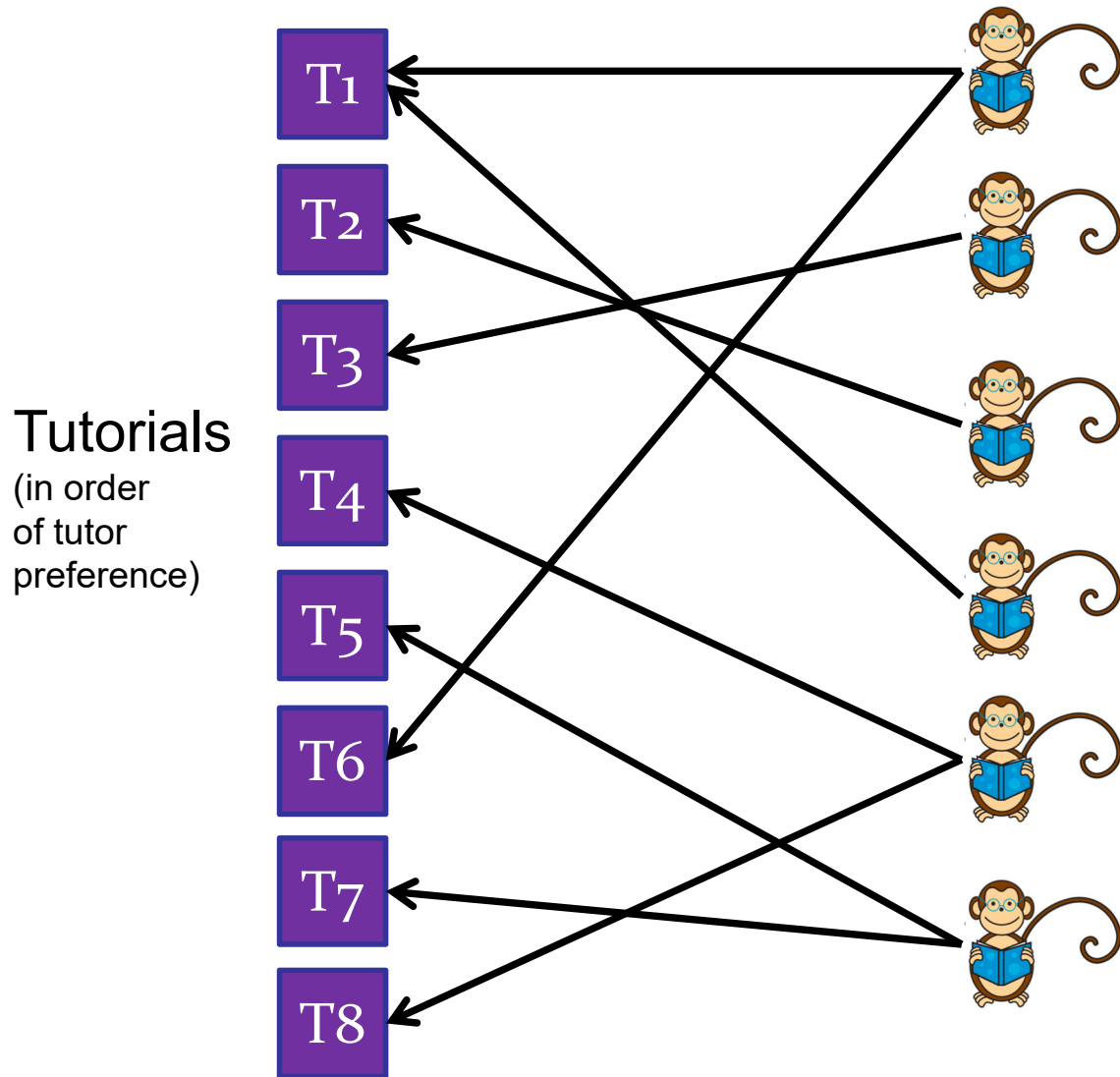
Number of  
students in  
WORST a tutorial  
only **decreases** as  
number of tutorials  
**increases**.

**Monotonic  
function of  
number of  
tutorials!**

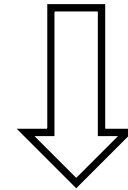
# A problem...

---

## Tutorial allocation



Monotonic  
function of  
number of  
tutorials!



Binary Search



# A problem...

---

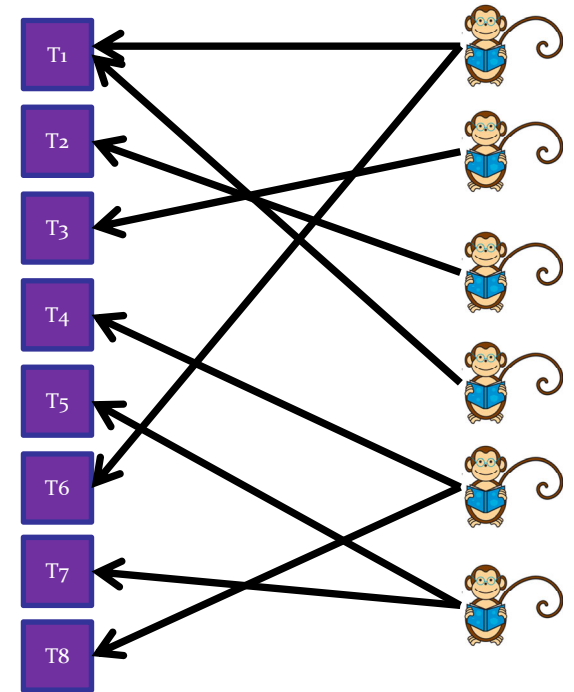
## Tutorial allocation

Solution:

Binary Search

Define:

$\text{MaxStudents}(x)$  = number of students in most crowded tutorial,  
if we offer  $x$  tutorials.



# Binary Search

---

**MaxStudents(x)** = number of students in most crowded tutorial, if we offer **x** tutorials.

Search (n)

begin = 0

end = n-1

**while** begin < end **do**:

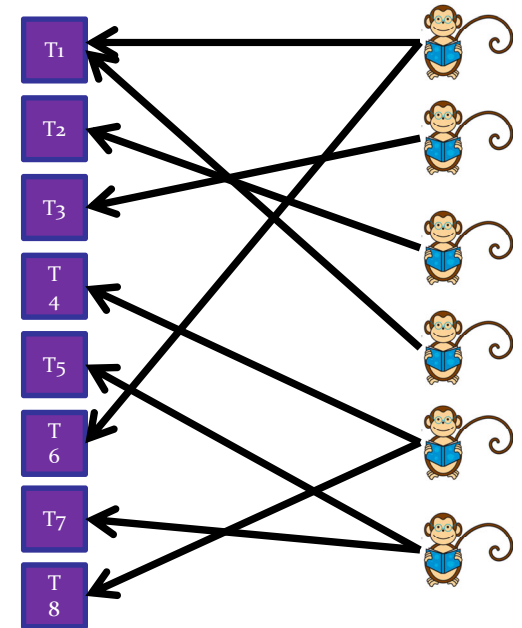
mid = begin + (end-begin)/2;

**if** MaxStudents(mid) <= 15 **then**

end = mid

**else** begin = mid+1

**return** begin



# Binary Search

---

Sorted array:  $A[0..n-1]$

2	4	4	5	6	7	8	9	11	17	23	28
---	---	---	---	---	---	---	---	----	----	----	----

Not just for searching arrays:

- Assume a complicated function:

`int complicatedFunction(int s)`

- Assume the function is always increasing:

`complicatedFunction(i) < complicatedFunction(i+1)`

- Find the minimum value  $j$  such that:

`complicatedFunction(j) > 100`

# Today: Divide and Conquer!

---

## Algorithm Analysis

- Big-O Notation
- Model of computation

## Searching

## Peak Finding

- 1-dimension
- 2-dimensions