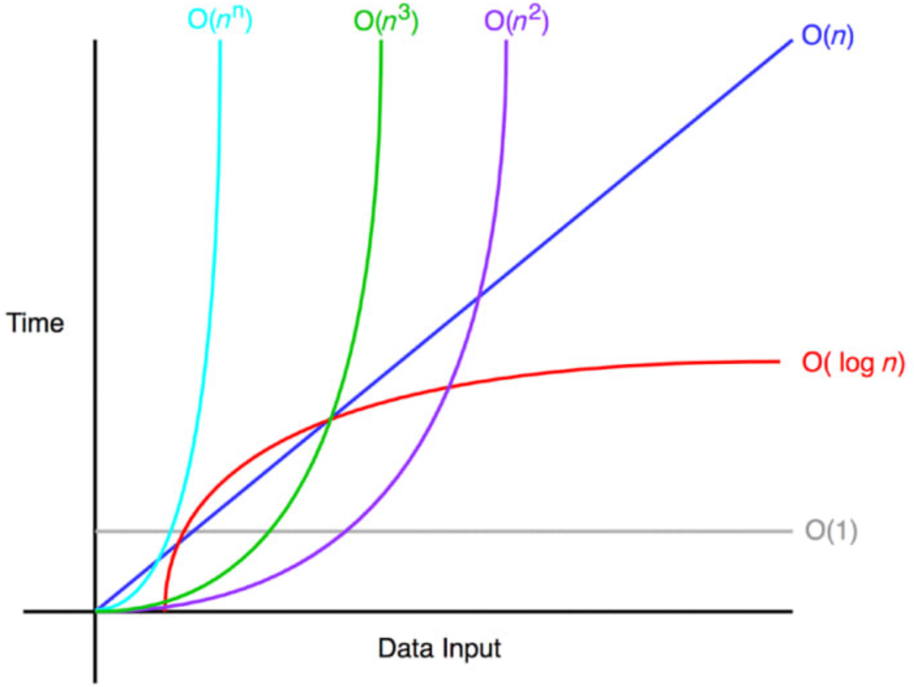
Big O and Searching

(Divide-and-conquer)

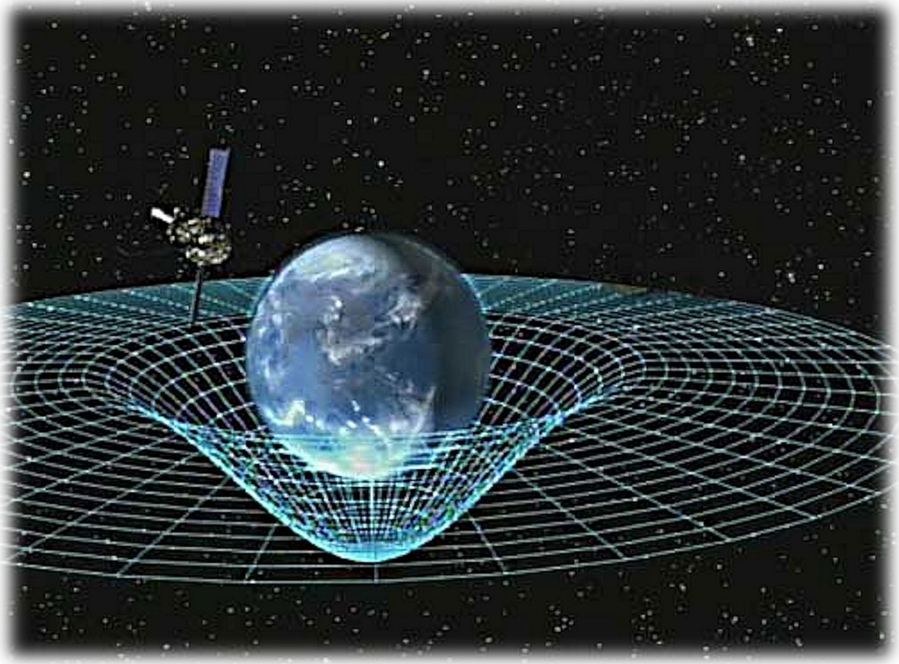
Order of Growth: The Big O



In Physics, We consider

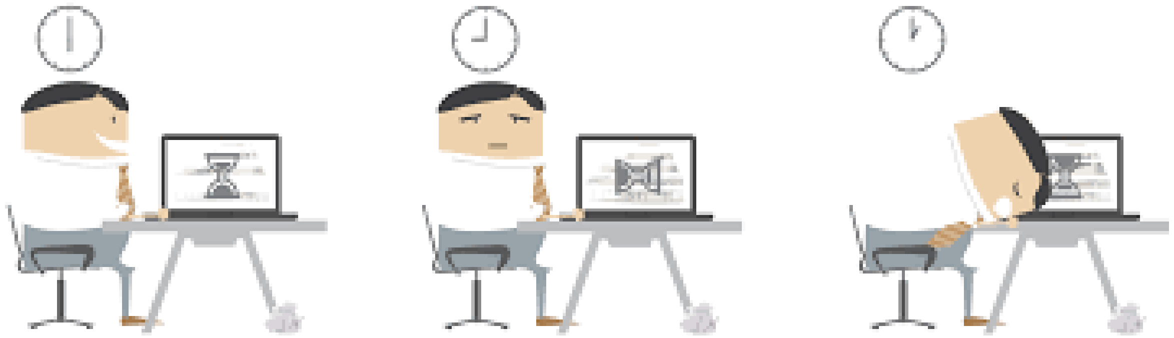
- Time

- Space

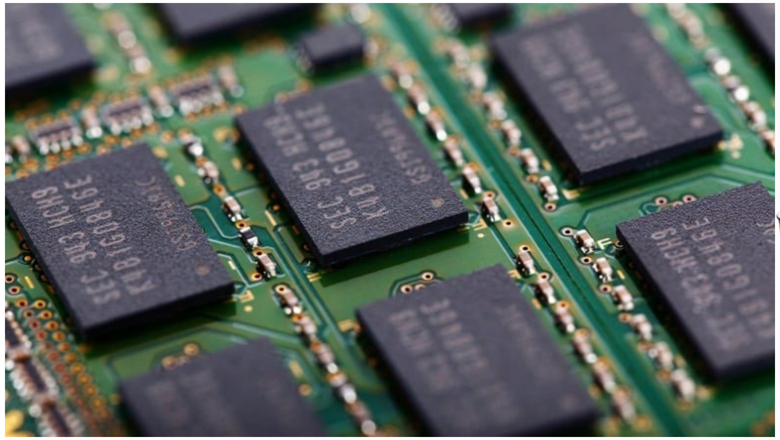


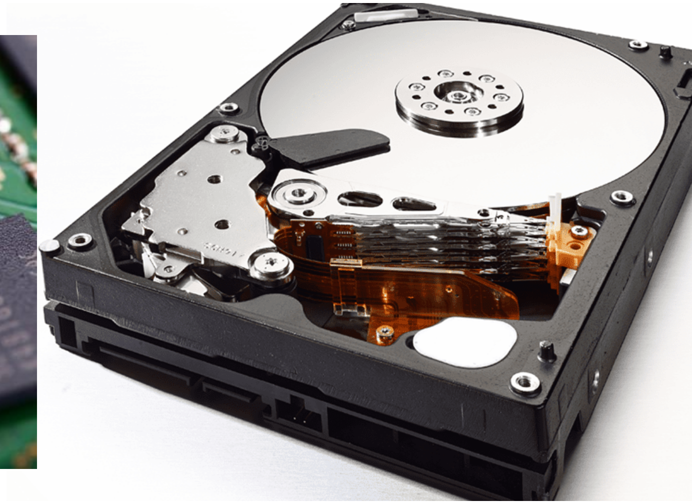
In CS, we consider

- Time
  - how long it takes to run a program



- Space
  - how much memory do we need to run the program





Order of Growth Analogy

- Suppose you want to buy a Blu-ray movie from Amazon (~40GB)
- Two options:
  - Download
  - 2-day Prime Shipping

- Which is faster?

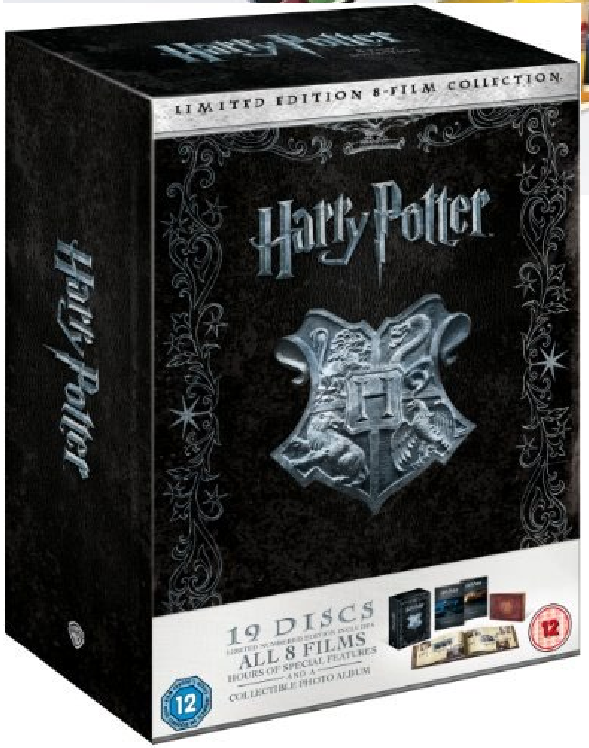
The Infinity Saga Box Set

Order of Growth Analogy

- Buy the full set?
  - 23 movies
- Two options:
  - Download
  - 2-day Prime Shipping
- Which is faster?

Order of Growth Analogy

- Or even more movies?



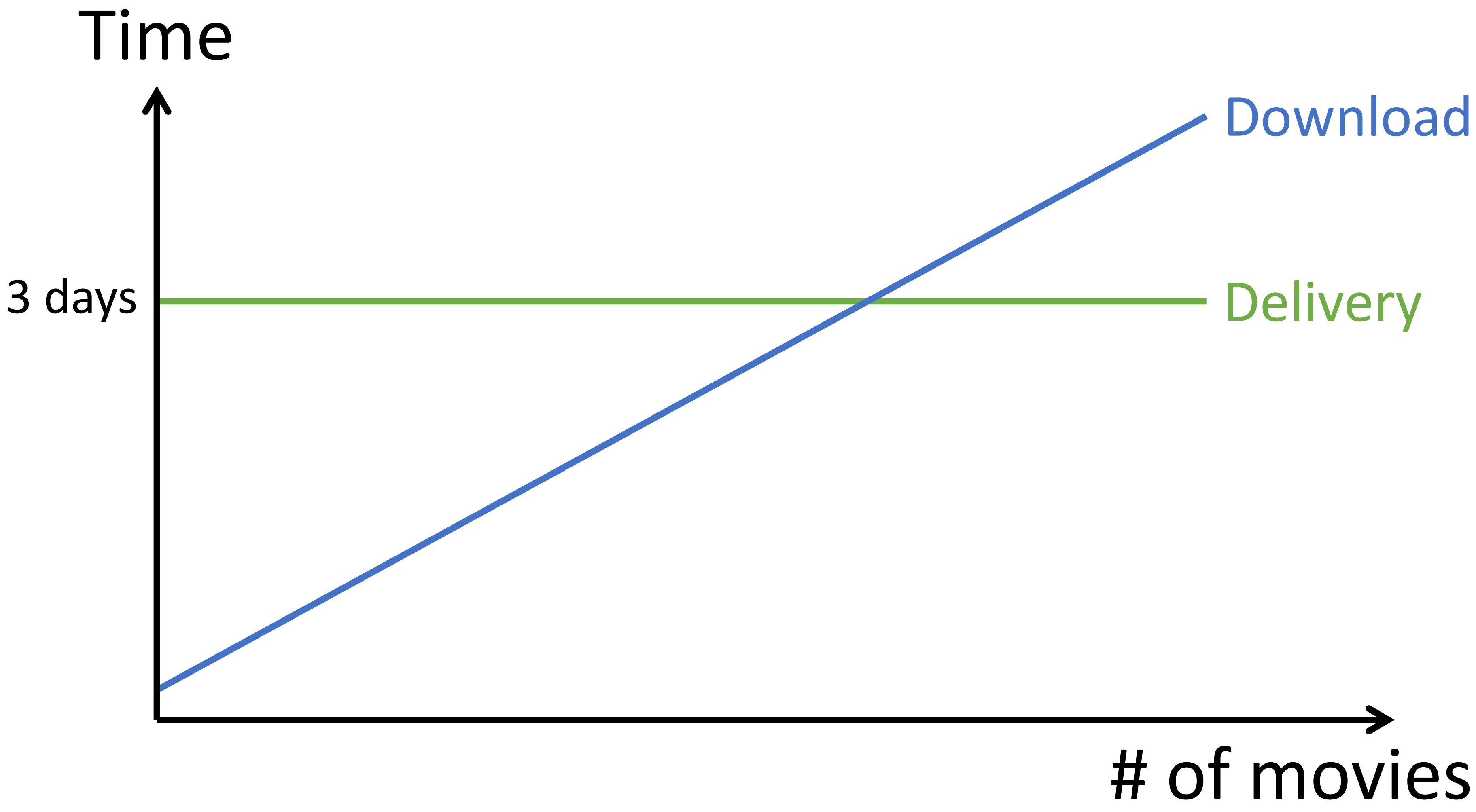
Details

- If downloading a movie require h hours,
  - Then downloading 1 movie requires h hours
  - Downloading 2 movie requires 2h hours
  - Downloading 3 movie requires 3h hours
  - …
- If shipping takes 3 days
  - Buying 1 movie need 3 days,
  - Buying 10 movies need 3 days
  - Buygin 100 movies need 3 days

Time needed

- For n movies, time needed will be
  - $T_d(n) = h \times n$ hours
- For n movies, time needed will be
  - $T_s(n)$ = 3 days

Download vs Delivery



Or

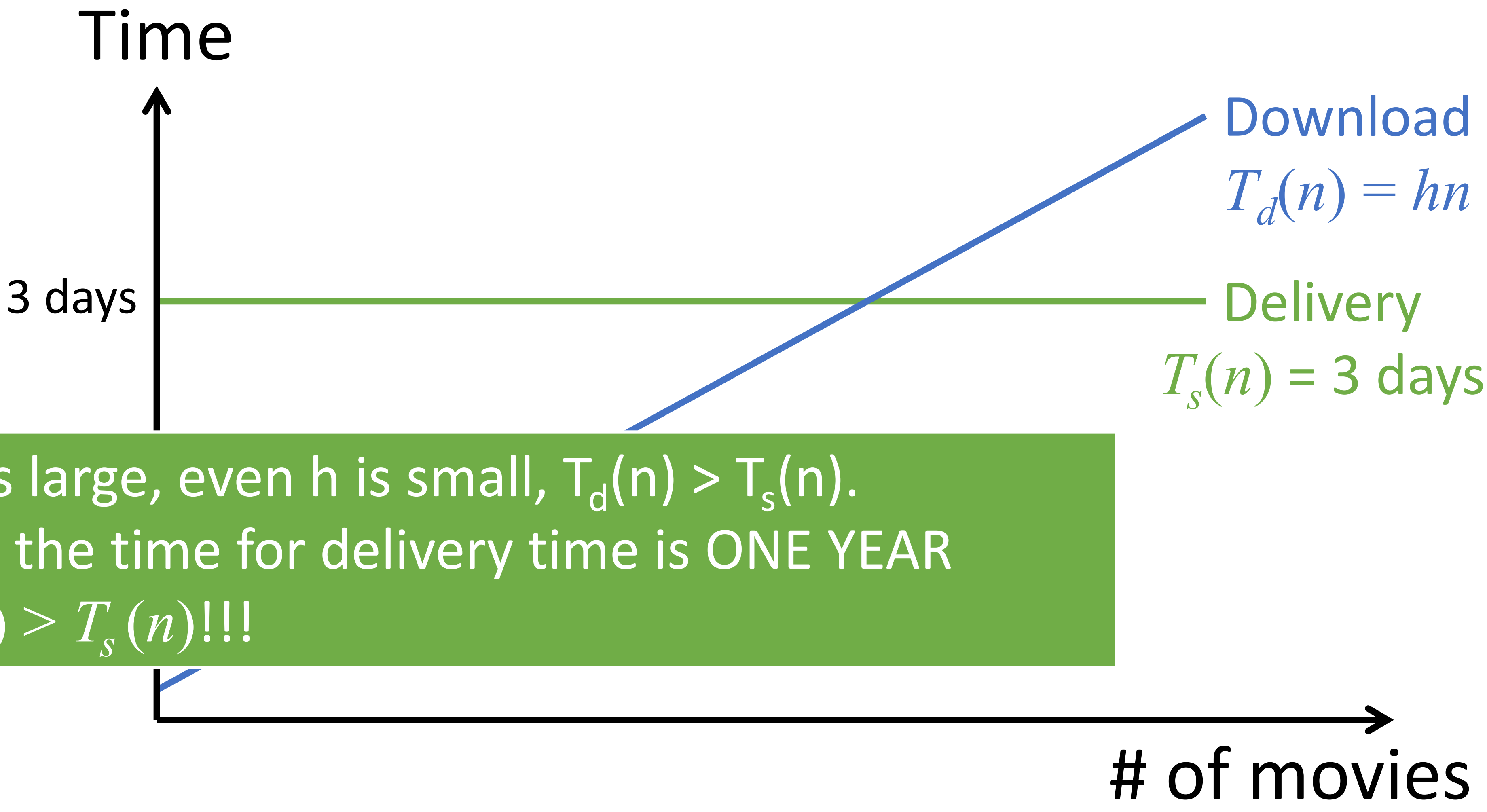
What if I want to download all the movies in the world! n = 100,000,000!!!

- Let's say we have an extremely super fast network
  - Downloading one movie only need 1 s
    - $h = 1/3600$
- Will you choose downloading or shipping

Time needed

- For n movies, time needed will be
  - $T_d(n) = h \times n$ hours
- For n movies, time needed will be
  - $T_s(n)$ = 3 days

Download vs Delivery



When n is large, even h is small, $T_d(n) > T_s(n)$.
And even the time for delivery time is ONE YEAR
Still $T_d(n) > T_s(n)$!!!

Which takes longer?

100k push operations

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

***k^2* push operations**

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

Which grows faster?

$T(k) = 100k$	$T(k) = k^2$
$T(0) = 0$	$T(0) = 0$
$T(1) = 100$	$T(1) = 1$
$T(100) = 10{,}000$	$T(100) = 10{,}000$
$T(1000) = 100{,}000$	$T(1000) = 1{,}000{,}000$

Time Required in Terms of the Size of Input

- Given three programs that perform the same functionality

- The time they needed to compute the results for n input are:
  - $T_1(n) = 10000000000000$
  - $T_2(n) = 100000n + 99999999$
  - $T_3(n) = n^2$

- Rank the fastest to the slowest in terms of time when n is very big?

Time Required in Terms of the Size of Input

- The time they needed to compute the results for n input are:
  - $T_1(n) = 1000000$
  - $T_2(n) = 10n + 999000$
  - $T_3(n) = n^2$
- Ordering:
  - $T_2(n) > T_1(n)$ when $n > 100$
  - $T_3(n) > T_1(n)$ when $n > 1000$
  - $T_3(n) > T_2(n)$ when $n > 1004$
- Conclusion, when n > 1004
  - $T_1(n) < T_2(n) < T_3(n)$

The Coefficients Do Not Contribute Much

- For the three timings
  - $T_1(n) = 10000000000000$
  - $T_2(n) = 100000n + 99999999$
  - $T_3(n) = n^2$

- You can see the order from fastest to slowest is the same as
  - $T_1(n) = 1$
  - $T_2(n) = n$
  - $T_3(n) = n^2$

Big O Notation

- For a function T(n), we “conclude” it in Big O Notation
- For example
  - $T(n) \;= 1234 = O(1)$
  - $T(n) = 453n\text{-}123 = O(n)$
  - $T(n) = 2n^2 + 9n + 1 = O(n^2)$
  - $T(n) = 5n^3 + n^2 + 10n - 9 = O(n^3)$

- Any idea how to convert into the Big O notation?
- In a naïve way,
  - Pick the highest degree/order term
  - Stripped all the coefficients

Big O Notation Definition (Formal)

- $T(n) = \mathrm{O}(\,f\,(n)\,)$ if:
  - there exists a constant $c > 0$
  - there exists a constant $n_0 > 0$
  - such that for all $n > n_0$:

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

- Example:
  - If $T(n) = 1234$
  - $T(n) < c\,f\,(n)$ for
    - $c = 1234$ (or 1235 for "greater than")
    - $F\,(n) = 1$
    - $n_0 = 1$ (actually totally doesn't matter)
  - Therefore $T(n) = \mathrm{O}(1)$

Big O Notation Definition (Formal)

- $T(n) = \mathrm{O}(\, f(n)\,)$ if:
  - there exists a constant $c > 0$
  - there exists a constant $n_0 > 0$
  - such that for all $n > n_0$:

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

- Example:
  - If $T(n) = 453n\text{-}123$
  - $T(n) < c\, f(n)$ for
    - $c = 453$
    - $f(n) = n$
    - $n_0 = 0$
  - Therefore $T(n) = \mathrm{O}(n)$

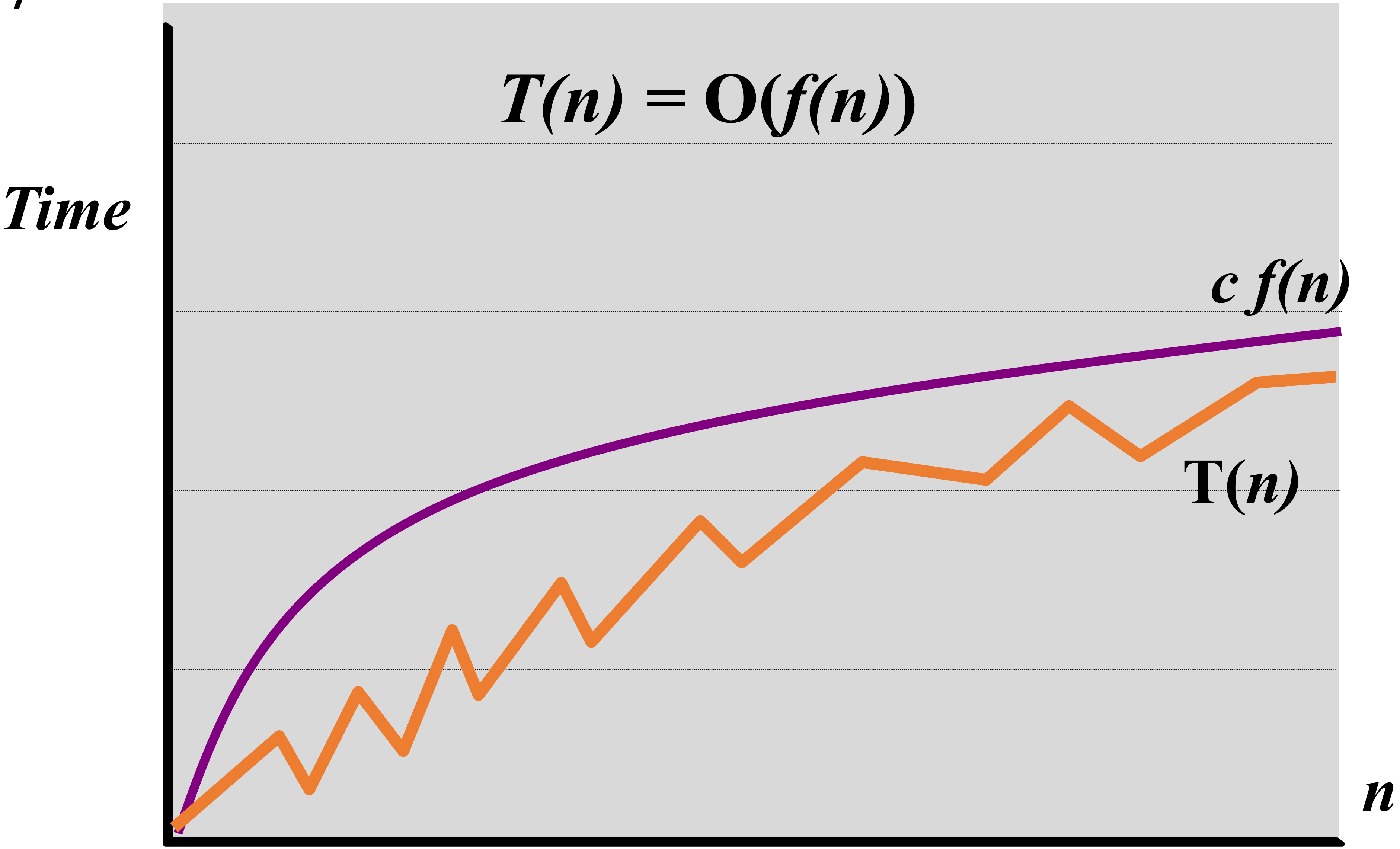
Big O Notation Definition (Formal)

- $T(n) = \mathrm{O}(\, f(n)\,)$ if:
  - there exists a constant $c > 0$
  - there exists a constant $n_0 > 0$
  - such that for all $n > n_0$:

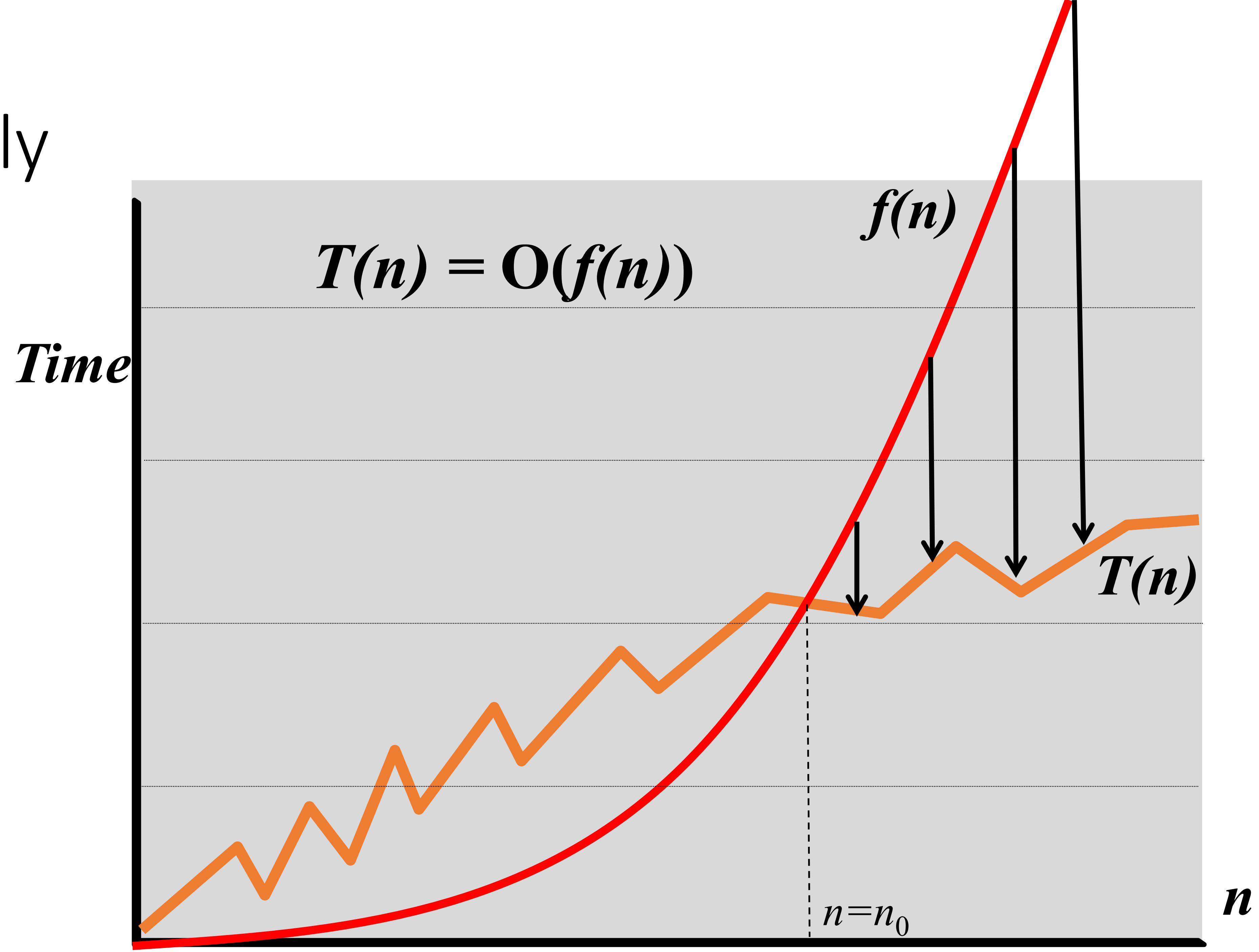
$$T(n) \leq c\, f(n)$$

- Example:
  - If $T(n) = 4n^2 + 24n + 16$
  - $T(n) < 4n^2 + 24n^2 + 16n^2 = 44n^2$
    - $c = 44$
    - $f(n) = n^2$
    - $n_0 = 1$
  - Therefore $T(n) = \mathrm{O}(n^2)$

Graphically



Graphically



Examples

$T(n)$	$f(n)$	big-O
$T(n) = 1000n$	$f(n) = n$	$T(n) = O(n)$
$T(n) = 1000n$	$f(n) = n^2$	$T(n) = O(n^2)$
$T(n) = n^2$	$f(n) = n$	$T(n) \neq O(n)$
$T(\mathrm{n}) = 13n^2 + n$	$f(n) = n^2$	$T(n) = O(n^2)$

Not tight

Some "Arithmetic" Rules

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

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

  - E.g. $10n^5 + 50n^3 + 10n + 17 = O(n^5)$
- If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then

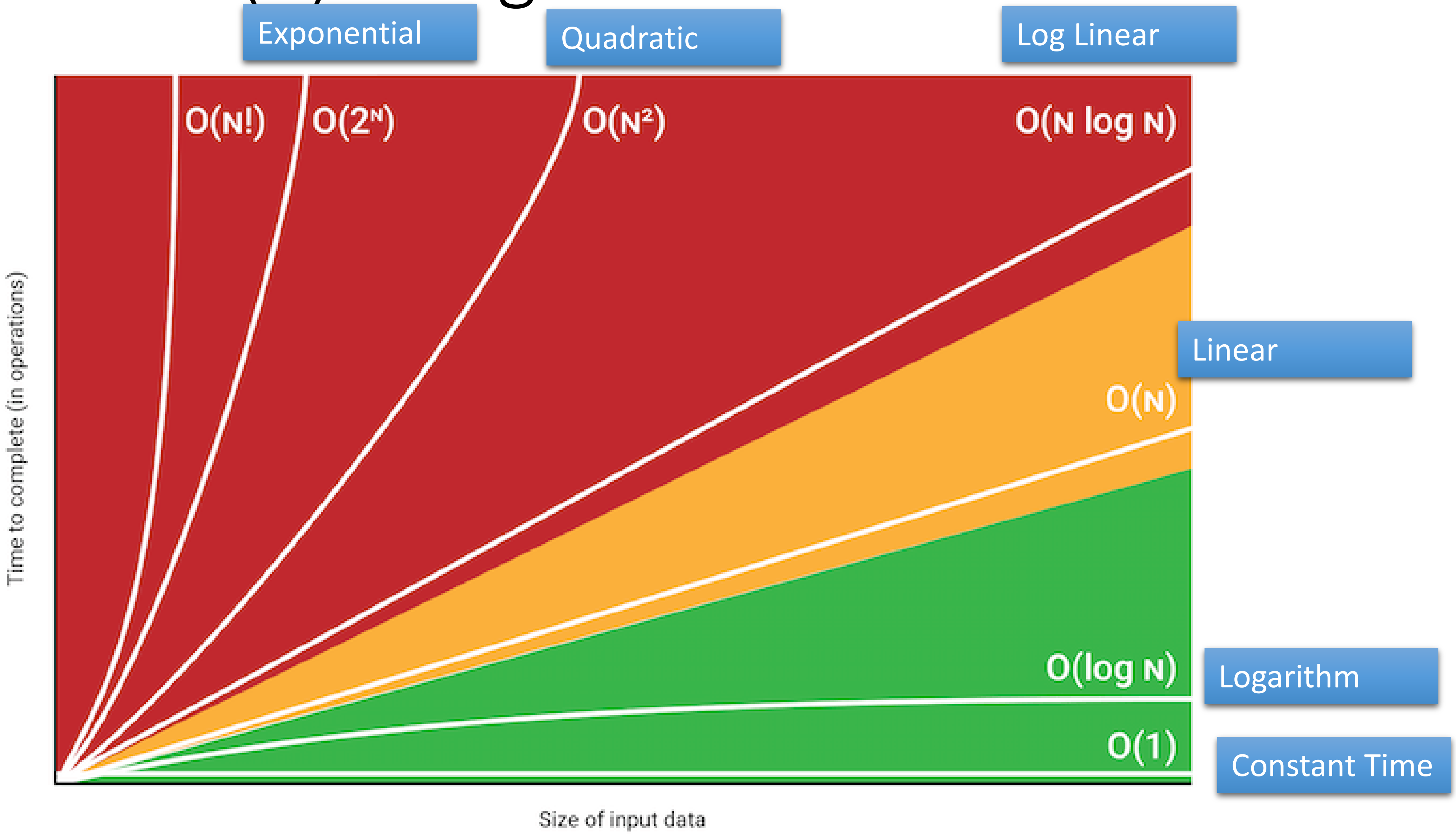
$$T(n) + S(n) = O(f(n) + g(n))$$

  - E.g. $10n^2 = O(n^2)$ and $5n = O(n)$, then

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

- If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then:
- $T(n) \times S(n) = O(f(n) \times g(n))$
  - E.g. $(10n^2)(5n) = 50n^3 = O(n \times n^2) = O(n^3)$

Common f(n) in Big-O In This Module



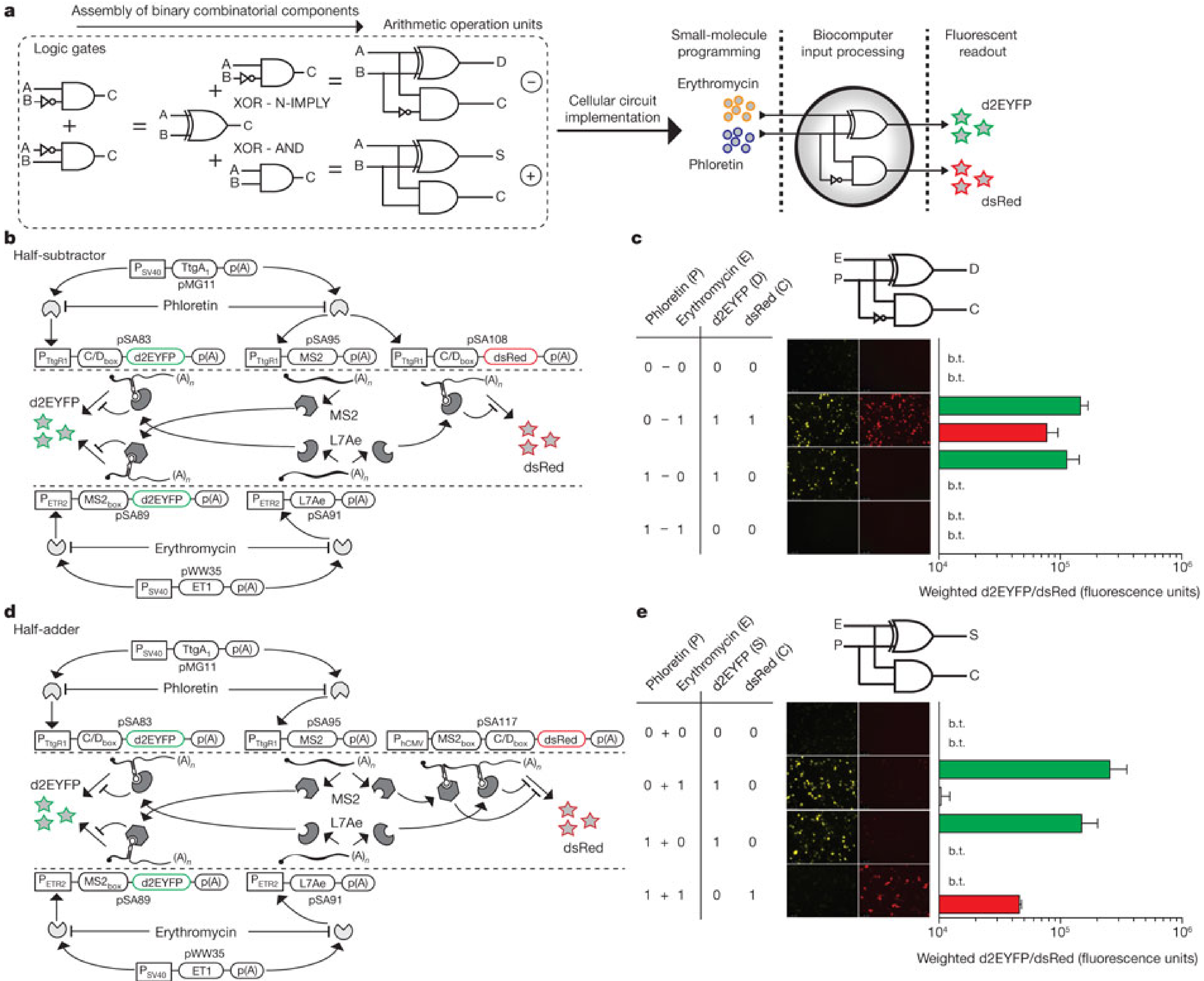
Algorithm Analysis

How do I know that my algorithm is in which class?

Model of Computations

- First, What are the different types of “computations” or different types of “computers”
  - Sequential vs Parallel
  - Deterministic vs Probabilistic
  - E.g. Biocomputers

Biocomputer



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

Rules

- Loops
  - cost = (# iterations)x(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
  - cost = (# iterations)x(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
  - cost = (cost of first) + (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) <= (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

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

$T(n\text{-}1)$ $T(n\text{-}1)$

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

Searching

You have an array. How do you find something in the array?

Linear Search

- Idea: go through the list from start to finish

5	2	3	4

- Example: Search for 3

5	2	3	4

3 not found, move on

5	**2**	3	4

3 not found, move on

5	2	**3**	4

Found 3.

Linear Search Complexity?

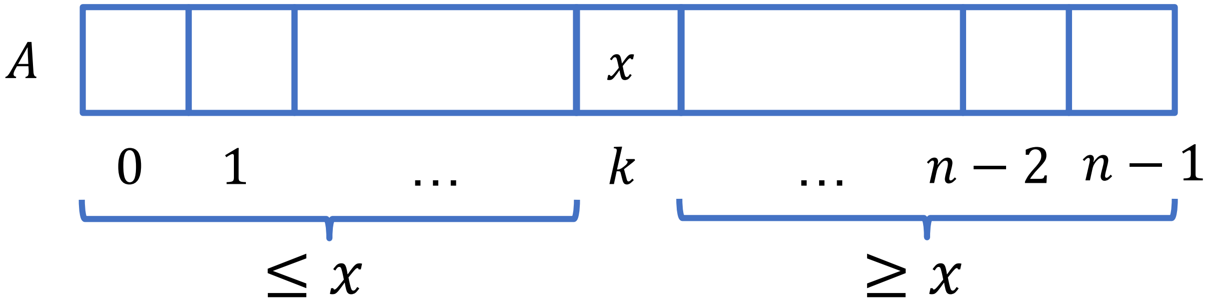
- If the array has n elements, how long does it take to search an item in the array?
  - Best case?
  - Worst case?
- When we talk about the complexity of an algorithm, we talked about the worst case if it’s not specified.
- Worst case of linear search: O(n)

Divide-and-Conquer

Binary Search

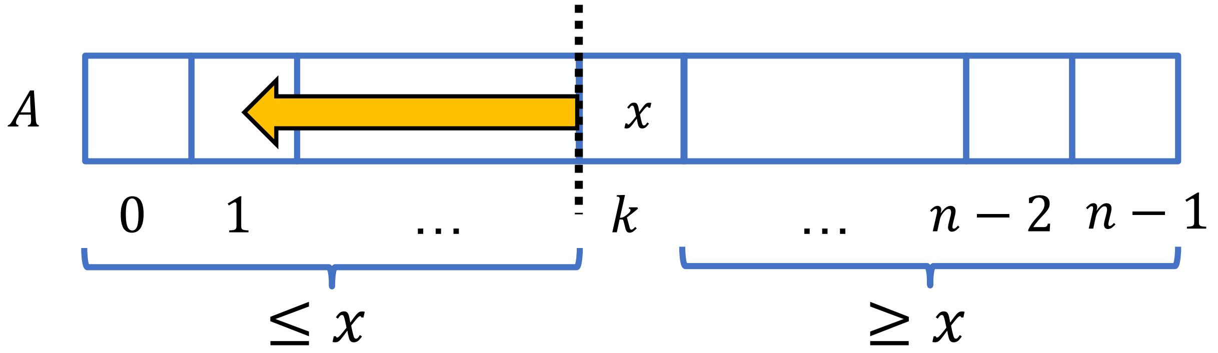
Idea

- If an array is sorted, we can “divide-and-conquer”
- Assuming an array A is sorted in ascending order with n elements
- For an index k, the element x = A[k] will divide the array into two
  - Left array: all smaller than x
  - Right array: all greater than x



Searching

- If the kth element is larger than what we are looking for, then we only need to search in the indices $<k$
  - Namely, the left array



- Otherwise? If the kth element is smaller than what we are looking for

Binary Search

1. Find the middle element.
2. If it is what we are looking for (key), return True.
3. If our key is smaller than the middle element, repeat search on the left of the list.
4. Else, repeat search on the right of the list.

- Until?

Looking for 25 (key)

5	9	12	18	25	34	85	100	123	345

Find the middle element: 34

5	9	12	18	25	34	85	100	123	345

Not the thing we're looking for: $34 \neq 25$

5	9	12	18	25	34	85	100	123	345

$25 < 34$, so we repeat our search on the left half:

5	9	12	18	25	34	85	100	123	345

Find the middle element: 12

5	**9**	**12**	**18**	**25**	**34**	**85**	**100**	**123**	**345**

25 > 12, so we repeat the search on the right half:

5	**9**	**12**	**18**	**25**	**34**	**85**	**100**	**123**	**345**

Find the middle element: 25

5	**9**	**12**	**18**	**25**	**34**	**85**	**100**	**123**	**345**

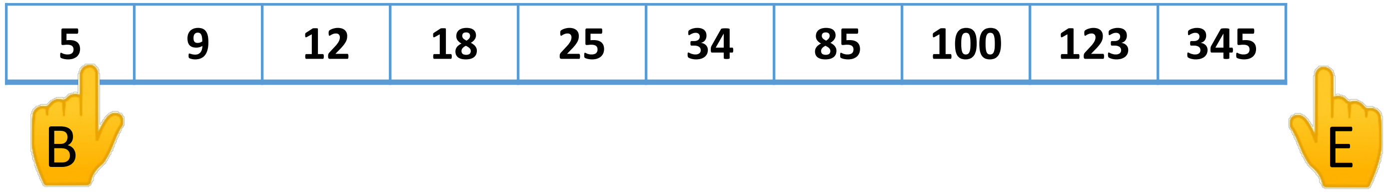
Great success: 25 is what we want

5	**9**	**12**	**18**	**25**	**34**	**85**	**100**	**123**	**345**

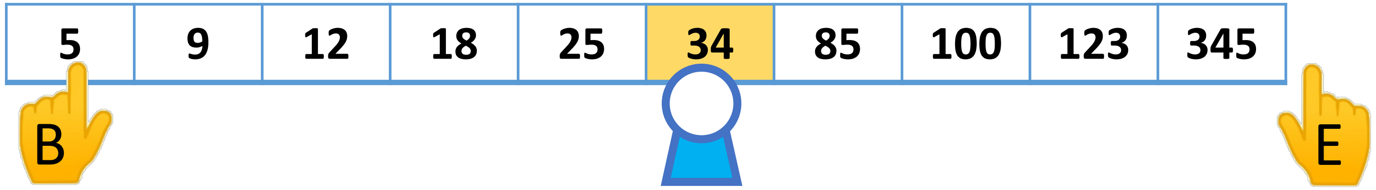
Pseudo Code

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

begin = 0, end = 10

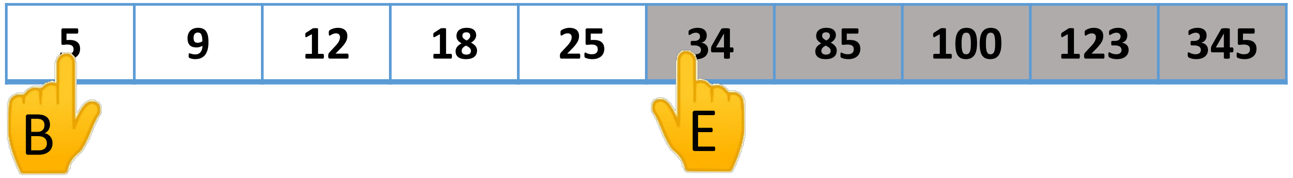


Middle index = (0+10)/2 = 5



Not the thing we’re looking for: 34 ≠ 25

25 < 34, end = (0+10)/2 = 5



Middle index = (0+5)/2 = 2 (integer division)

5	9	12	18	25	34	85	100	123	345

B E

25 > 12, begin = 1 + (0+5)/2 = 3

5	9	12	18	25	34	85	100	123	345

B E

Middle index = (3+5)/2 = 4

5	9	12	18	25	34	85	100	123	345

Great success: 25 is what we want

5	9	12	18	25	34	85	100	123	345

Pseudo Code

```
Search(A, key, n)
    begin = 0
    end = n
    while begin < end do:
        mid = (begin+end)/2
        if  key == A[mid] then return mid
        if  key < A[mid] then
            end = mid
        else begin = 1+mid
    return -1 // return -1 if not found
```

Complexity?

```
Search(A, key, n)
    begin = 0
    end = n
    while begin < end do:
        mid = (begin+end)/2
        if  key == A[mid] then return mid
        if  key < A[mid] then
            end = mid
        else begin = 1+mid
    return -1 // return -1 if not found
```

- Worst Case when?
  - When key is in A, or
  - When key is NOT in A?
- Every single line should be O(1)
- But how many times does this loop repeat?

How Many Times?

- Given an array with n elements
- How many times we can cut it into half until only one element left?
  - Cut 0 time: size of array = n
  - Cut 1 time: size of array = n/2
  - Cut 2 times: size of array = n/4 = $n/(2^2)$
  - Cut 3 times: size of array = n/8 = $n/(2^3)$
  - …
  - Cut d times: size of array = $n/(2^d)$

- How many times until $n/(2^d) = 1$?
  - $n/(2^d) = 1$
  - $n = 2^d$
  - $\log(n) = \log(2^d)$
  - $\log(n) = d \log(2)$
  - $d = \log(n)/\log(2)$
  - $d = O(\log(n))$

Binary Search Complexity

- Complexity of Binary Search
  - $O(\log n)$
- Does it matter what base it is for the logarithm?
  - Base 2?
  - Base e?
  - Base 10?

- How many times until $n/(2^d) = 1$?
  - $n/(2^d) = 1$
  - $n = 2^d$
  - $\log(n) = \log(2^d)$
  - $\log(n) = d \log(2)$
  - $d = \log(n)/\log(2)$
  - $d = O(\log(n))$

More Divide-and-Conquer

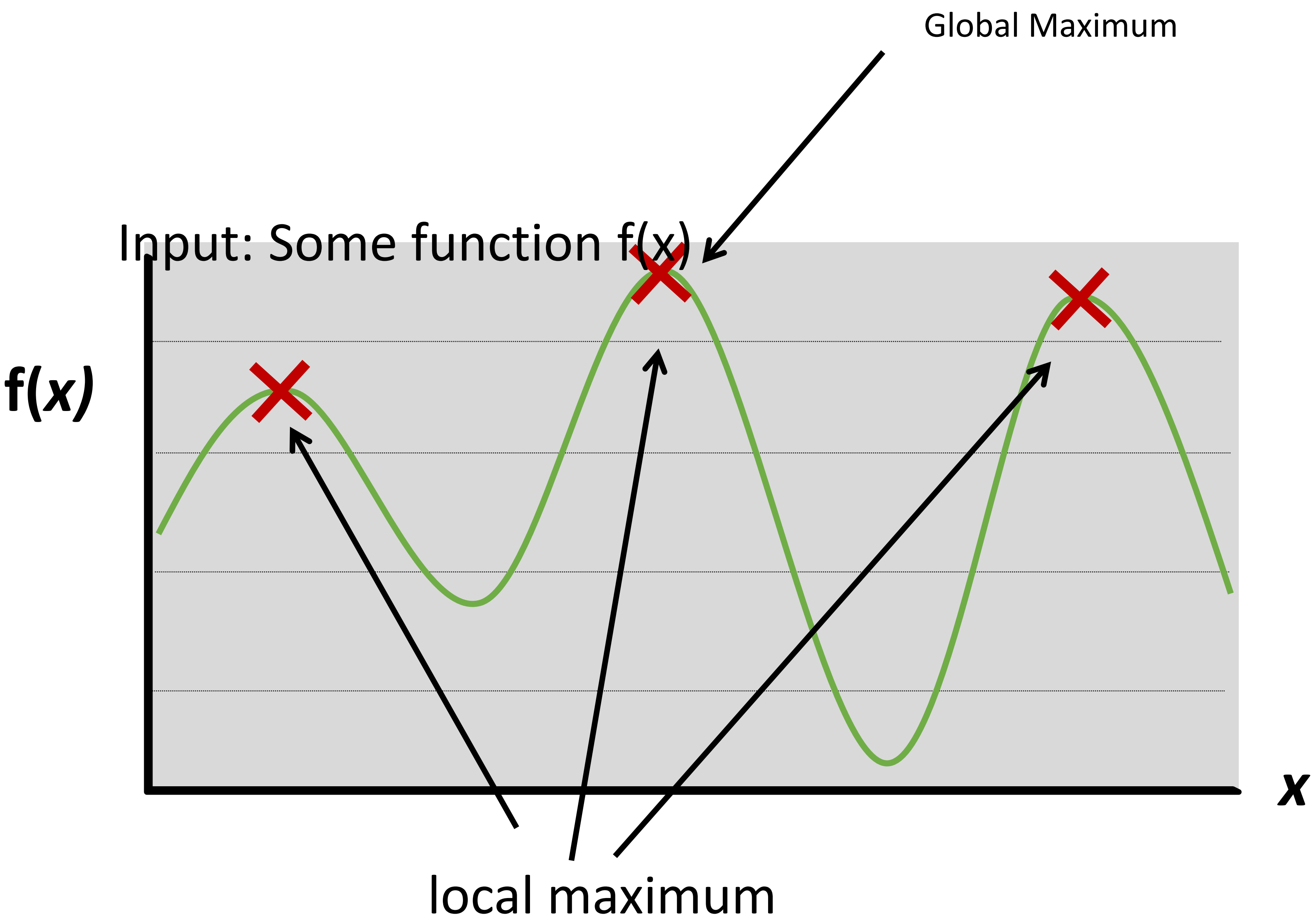
Given an Array A with n Elements

- Assuming no two elements are the same
- How do I find the global maximum?
- Complexity?

Given an Array A with n Elements

- Assuming no two elements are the same
- How do I find the local maximum?
  - An element `A[i]` is a local maximum if its neighbor(s) is/are smaller than `A[i]`
  - Most of the elements have two neighbors
  - `A[0]` and `A[n-1]` have only one neighbor each.

Peak Finding



What for?!

- Global Maximum for Optimization problems:
  - Find a good solution to a problem.
  - Find a design that uses less energy.
  - Find a way to make more money.
  - Find a good scenic viewpoint.
  - Etc.
- Why local maximum (a peak)?
  - Finds a good enough solution.
  - Local maxima are close to the global maximum?
  - Much, much faster.

Find the Global Maximum (Unsorted Array)

```
FindMax(A,n)
        max = A[0]
        for i = 0 to n-1 do:
            if (A[i]>max) then max=A[i]
        return max
```

- Time Complexity: $O(n)$

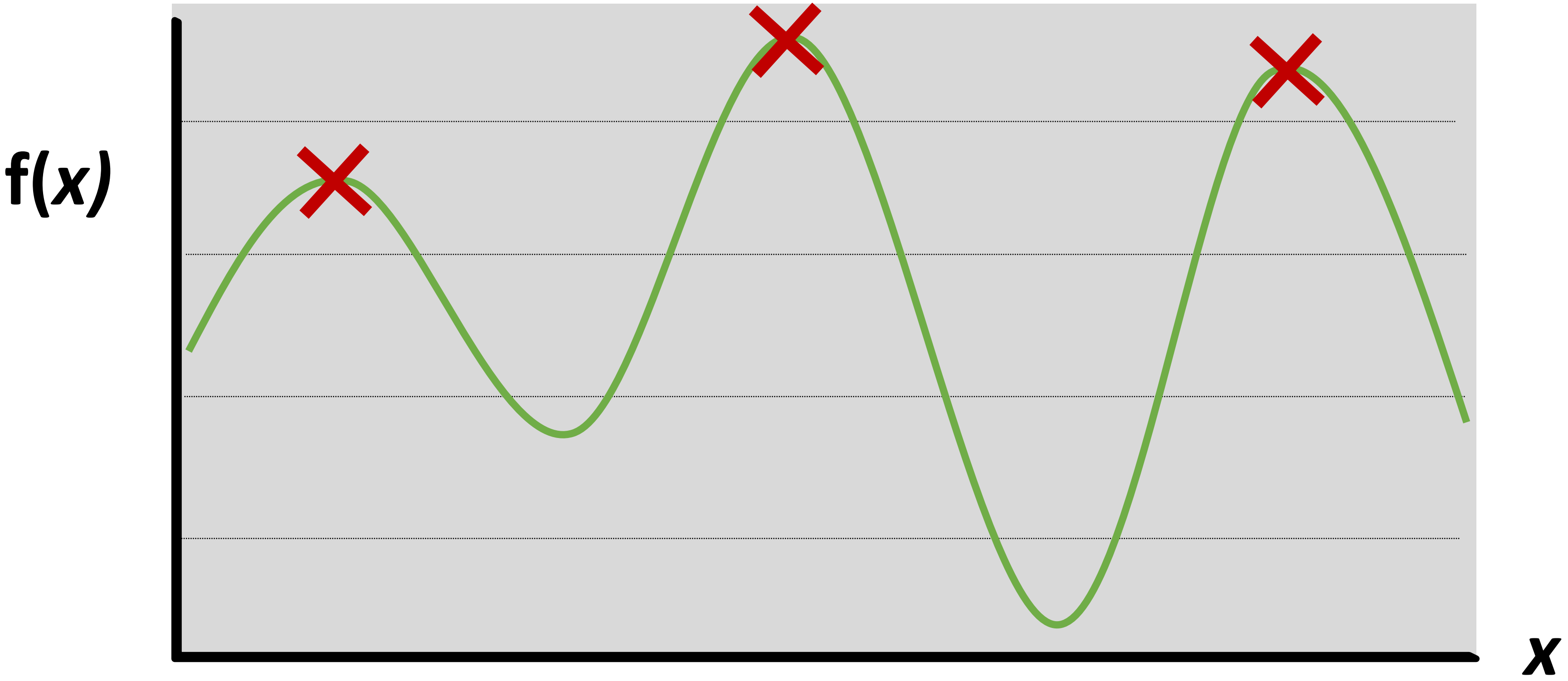
Find the Global Maximum (Sorted Array)

```
FindMax(A,n)
        return A[n-1]
```

- Time Complexity: $O(1)$

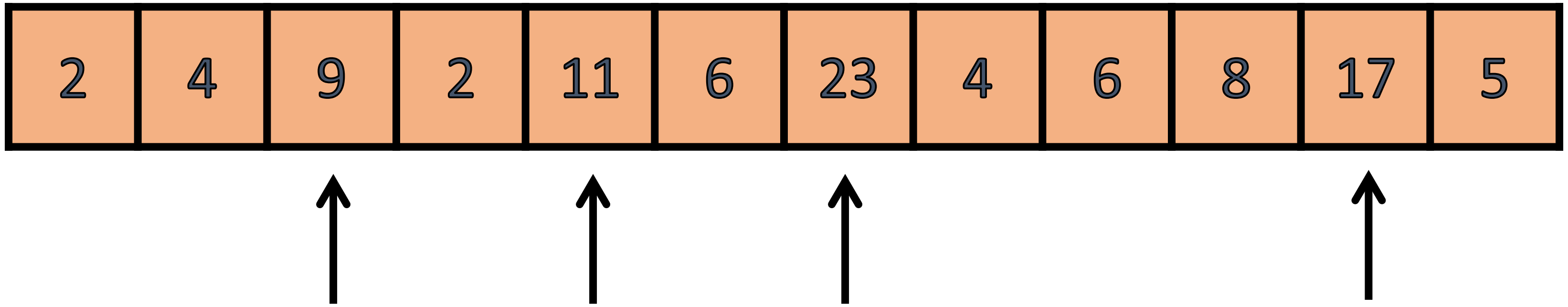
Peak (Local Maximum) Finding

- Just report ANY ONE Local Maximum



Peak Finding

- Given some array A



Output: a local maximum A[i] in A such that

$$A[i-1] \le A[i] \quad \textbf{and} \quad A[i+1] \le A[i]$$

And we assume that

$$A[0] = A[n] = -\infty$$

Peak Finding: Algorithm 2

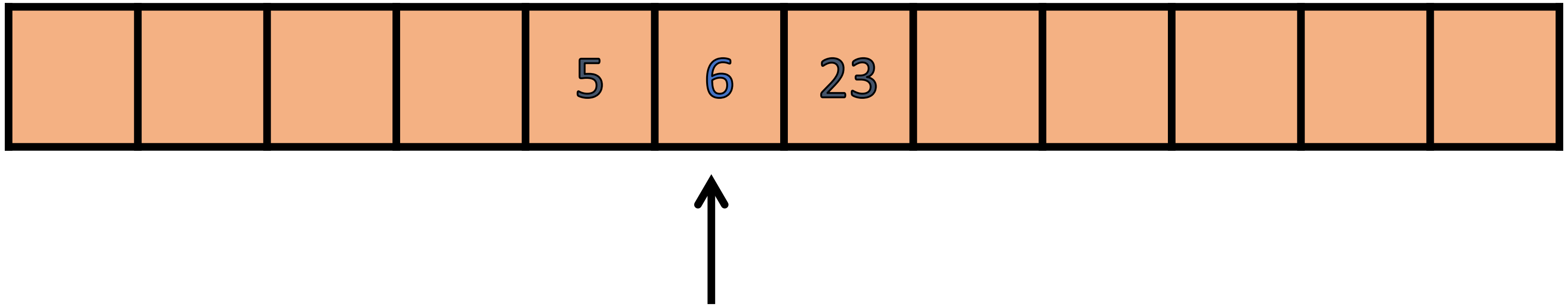
- Given some array A

2	4	9	2	5	6	23	4	6	8	17	5

- Can we do divide-and-conquer?
- Let’s say we check the middle one
  - If it is a peak, BINGO! Finished
  - But what if it’s not a peak?

Peak Finding: Algorithm 2

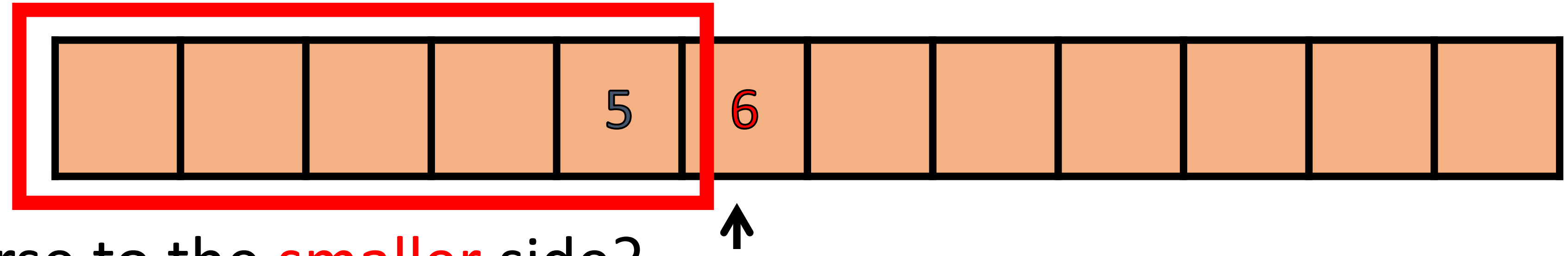
- Given some array A



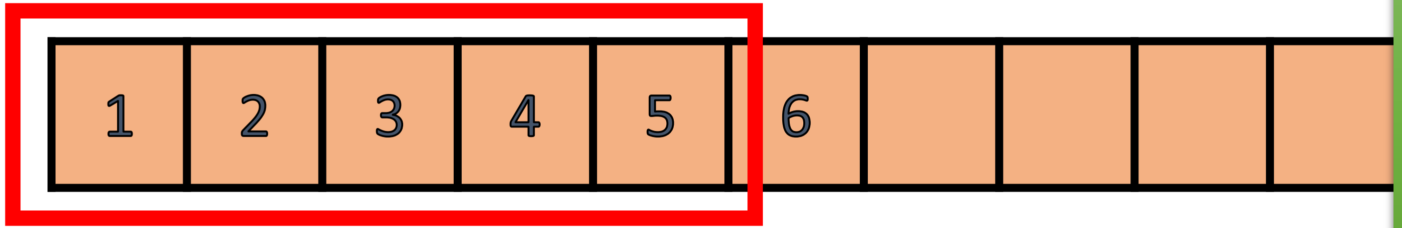
- If A[i] is NOT a peak, chances are:
  - ~~Left is smaller than A[i], Right is smaller than A[i]~~
  - Left is smaller than `A[i]`, Right is bigger than `A[i]`
  - Left is bigger than `A[i]`, Right is bigger than `A[i]`
  - Left is bigger than `A[i]`, Right is smaller than `A[i]`
- If I want to do recursion, should we recurse to the left or the right?

Peak Finding: Algorithm 2

- Given some array A



- Recurse to the smaller side?
  - Are we sure there will be a peak in the smaller side?
  - What if

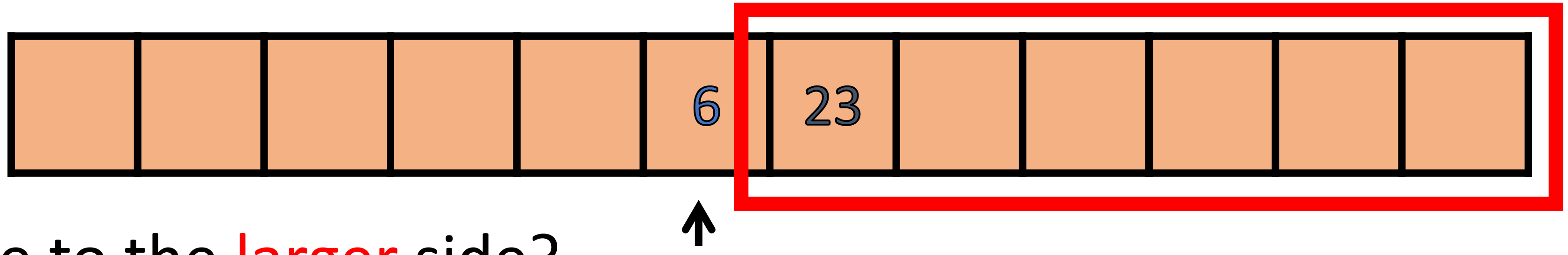


There is a chance that the smaller side has NO peak!!!!

  - We assume `A[-1]` = -∞
  - It could happen to the right sub-array if `A[i+1] < A[i]`

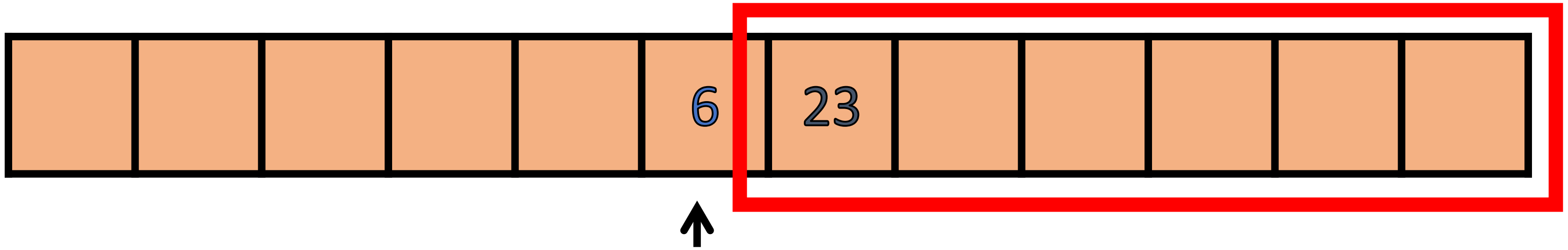
Peak Finding: Algorithm 2

- Given some array A

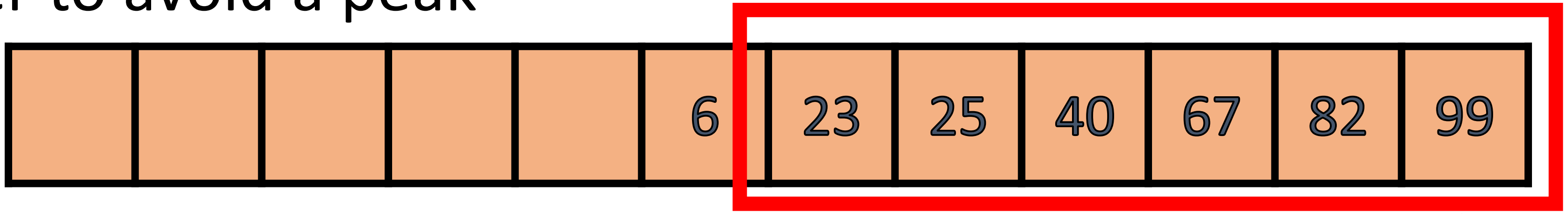


- Recurse to the larger side?
  - Is there a chance that there will be NO peak?
  - Prove by contradiction, assuming there is NO peak on the larger side
  - Since `A[i+1] > A[i]`, the next `A[i+2]` must be larger than `A[i+1]` if there is no peak
    - Otherwise, `A[i+1]` is a peak!

Prove by contradiction, assuming there is NO peak on the larger side

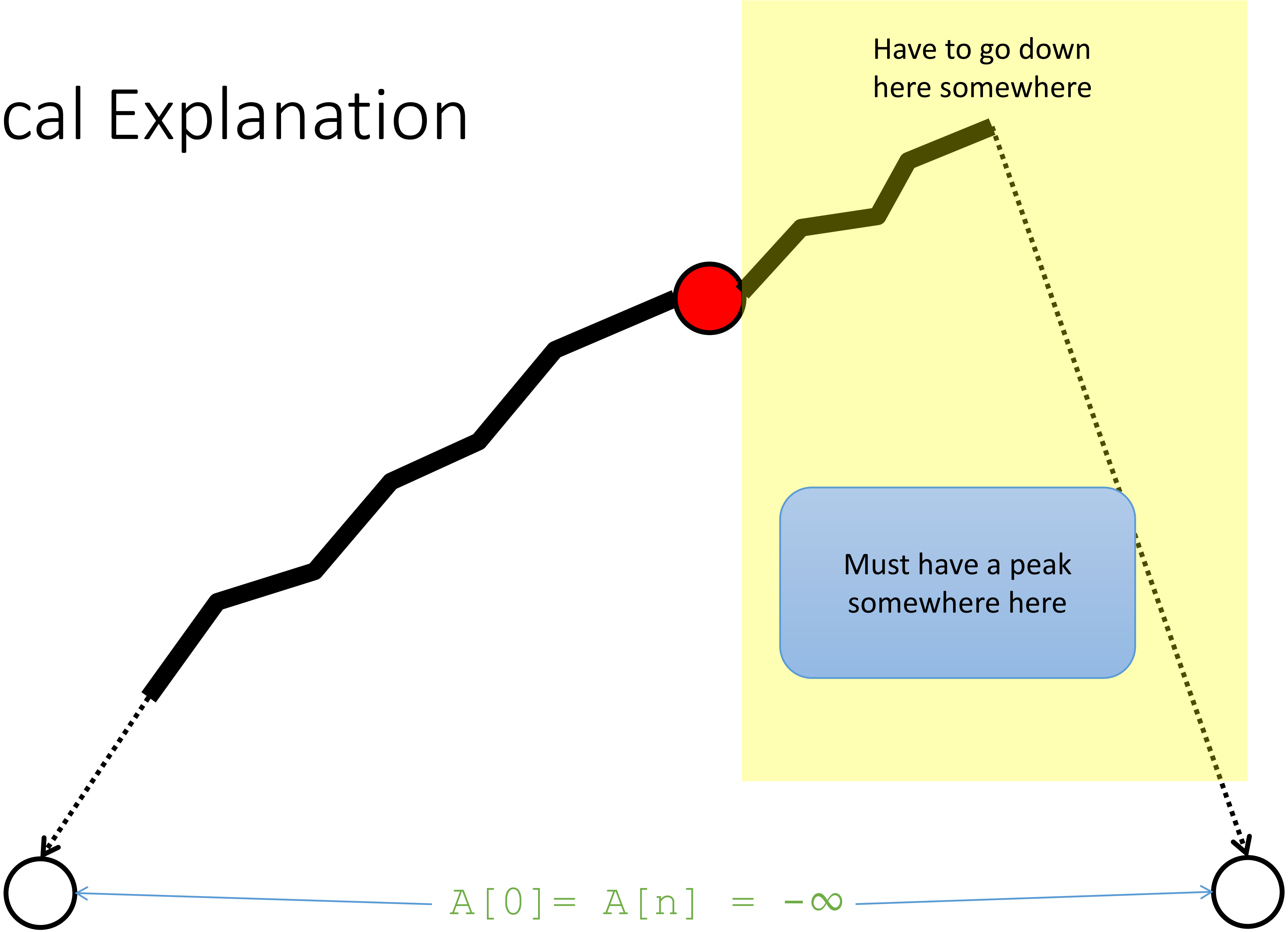


- Since `A[i+1] > A[i]`, the next `A[i+2]` must be larger than `A[i+1]` if there is no peak
  - Otherwise, `A[i+1]` is a peak!
- Following this logic, every `A[j+1]` must be larger than `A[j]` for j > i in order to avoid a peak



- But `A[n-1]` will be a peak then!

Graphical Explanation



Peak Finding: Algorithm 2

- Given some array A

2	4	9	2	5	6	23	4	6	8	17	5

- Let's say we check the middle one
  - If it is a peak, BINGO! Finished
  - Otherwise, recurse into the larger half of the array
    - There could be a chance that both sides are larger.
    - But it doesn't matter which one to go to. We just need one peak to report

Peak Finding: Algorithm 2

•

Time Complexity?

↑

```
FindPeak(A, n)
        mid = n/2
        if A[mid] is a peak then return mid
        else if A[mid+1] > A[mid] then
            Search for peak in right half.
        else if A[mid-1] > A[mid] then
            Search for peak in left half.
```

How About Peak in 2D Array?

Peak Finding in 2D

- Given a 2D array
  - Assuming there are n rows and m columns
- An element is a peak if its direct neighbors are smaller than it

10	8	5	2	1
3	2	1	5	7
17	5	1	4	1
7	9	4	6	4
8	1	1	2	6

Peak Finding in 2D: Algorithm 1

- Step 1: Find the Global maximum for each column

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
7	9	5	5

Q1: Will this work?!

Q2: Time Complexity?

Find 1D peak.

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 1

- Step 1: Find the Global maximum for each column

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
7	9	5	5

- Q1: Will this work?!
  - The output will be larger than its vertical neighbors
  - The output will be larger than the global maximums of the neighbor columns
    - Thus, larger than the immediate neighbors of the output

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 1

- Step 1: Find the Global maximum for each column

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
7	9	5	5

- Q2: Time Complexity?
- Step 1: $O(n)$ for each column and there are m columns
  - Thus $O(mn)$
- Step 2: Local peak of an array of m elements
  - $O(\log m)$
- Overall: $O(mn + \log(m))$
  - Or simply $O(n^2)$ for $n > m$

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 2

- Step 1: Find the Local maximum for each column

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
3	4	3	3

Find 1D peak.

Q1: Will this work?!

Q2: Time Complexity?

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 2

- Step 1: Find the Local maximum for each column

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3

3	4	3	3

Output: 1D peak.

- Q1: Will this work?!
  - This is already a counter example that is NOT working
- Q2: Even it's not working, what's its time complexity?

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 1

- ~~Step 1: Find the Global maximum for each column~~

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
7	9	5	5

- Q2: Time Complexity?
- Step 1: $O(n)$ for each column and there are m columns
  - Thus $O(mn)$
- Step 2: Local peak of an array of m elements
  - $O(\log m)$
- Overall: $O(mn + \log(m))$
  - Or simply $O(n^2)$ for $n > m$

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 1

- ~~Step 1: Find the Global maximum for each column~~

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
?	?	?	?

Find 1D peak.

How?

- Step 2: Find local peak in the array of max elements.

Peak Finding in 2D: Algorithm 3

- ~~Step 1: Find the Global~~

3	4	5	2
2	1	2	5
1	9	1	2
7	5	3	3
?	?	?	?

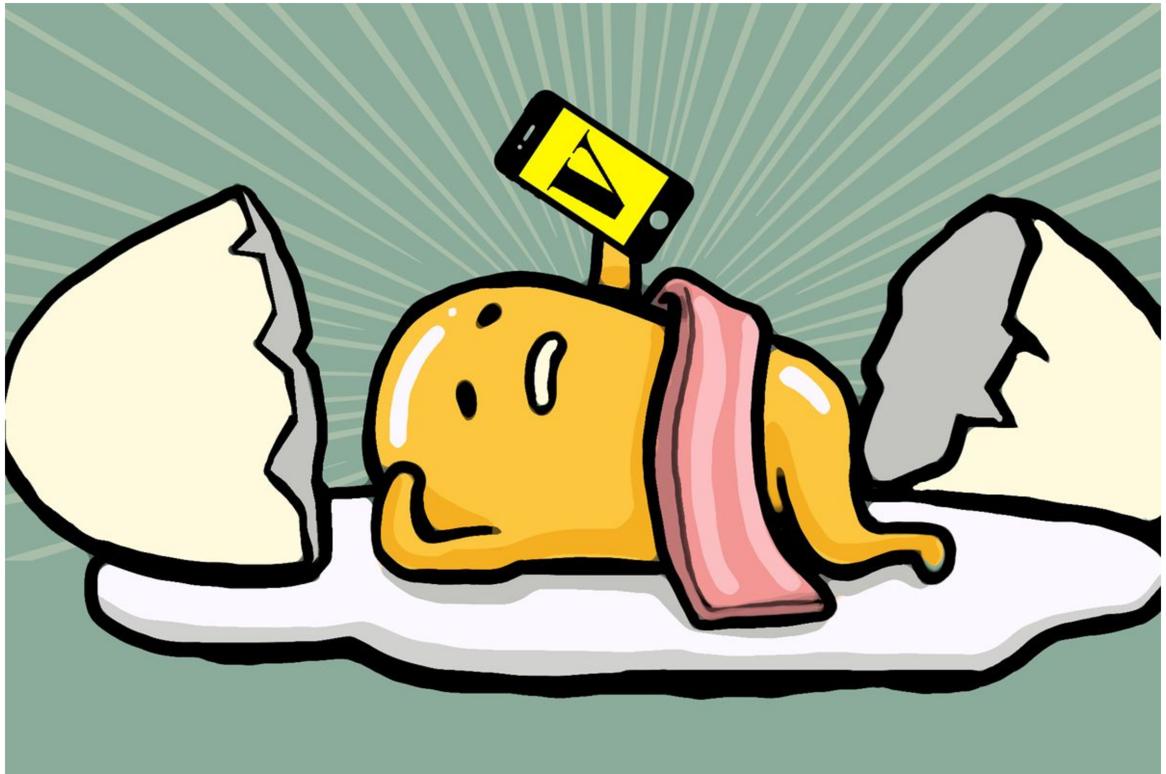
- How?
- Skipped Step 1
- Do Step 2
  - Wait until I need that global maximum of THAT column, then I compute that!

Find 1D peak.

- Step 2: Find local peak in the array of max elements.

Lazy Evaluation

- Lazy evaluation, or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed and which also avoids repeated evaluations



In Step 2

7	10	12	20	7	9	4	3	1	18	5	17	4
19	11	7	4	6	8	10	3	5	6	8	14	8
6	9	14	4	7	9	3	12	9	8	3	10	6
?	?	?	?	?	?	?	?	?	?	?	?	?

- Find the middle
  - Check if it’s a 1D Peak

In Step 2

7	10	12	20	7	9	4	3	1	18	5	17	4
19	11	7	4	6	8	10	3	5	6	8	14	8
6	9	14	4	7	9	3	12	9	8	3	10	6
?	?	?	?	?	9	10	12	?	?	?	?	?

Column i

- Find the middle
  - Check if it’s a 1D Peak
  - Now we evaluate the global maximum of the column i-1, i and i+1

- Then recurse the the “greater” half

In Step 2

7	10	12	20	7	9	4	3	1	18	5	17	4
19	11	7	4	6	8	10	3	5	6	8	14	8
6	9	14	4	7	9	3	12	9	8	3	10	6
?	?	?	?	?	9	10	12	?	18	8	14	?

Middle of the
“greater” half

- Find the middle
  - Check if it’s a 1D Peak
  - Now we evaluate the global maximum of the column i-1, i and i+1
- Then recurse to the “greater” half

- How many columns do we need to evaluate its global maximum?

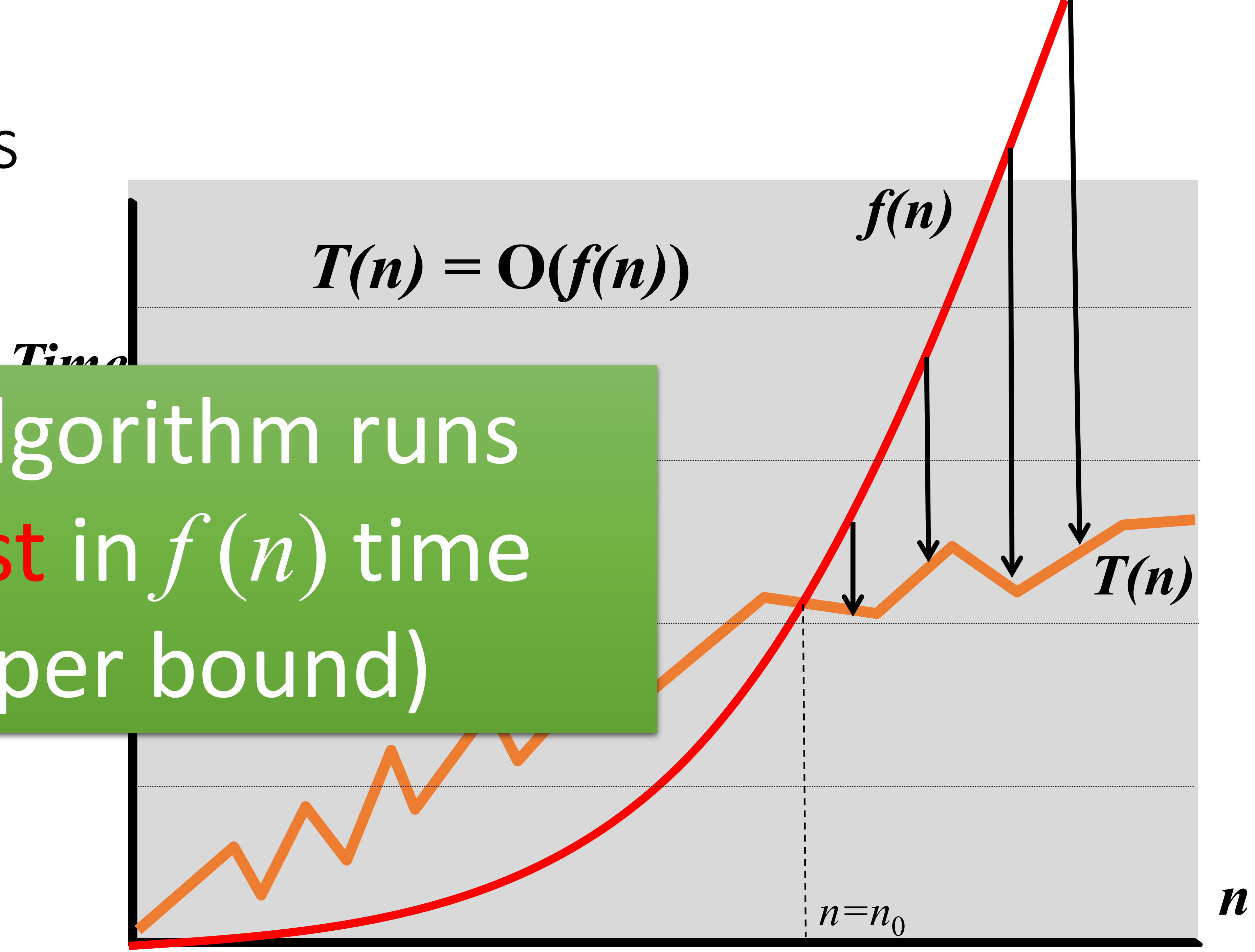
2D Peak Finding Algorithm 3

- Find peak in the array of peaks:
  - Use 1D Peak Finding algorithm
  - For each column examined by the algorithm, find the maximum element in the column.
- Running time:
  - 1D Peak Finder Examines $O(\log m)$ columns
  - Each column requires $O(n)$ time to find the global max
  - Total: $O(n \log m)$
- (Much better than $O(nm)$ of before.)

About the Big-O Notation

- Considering the worst case
- How about the best case?

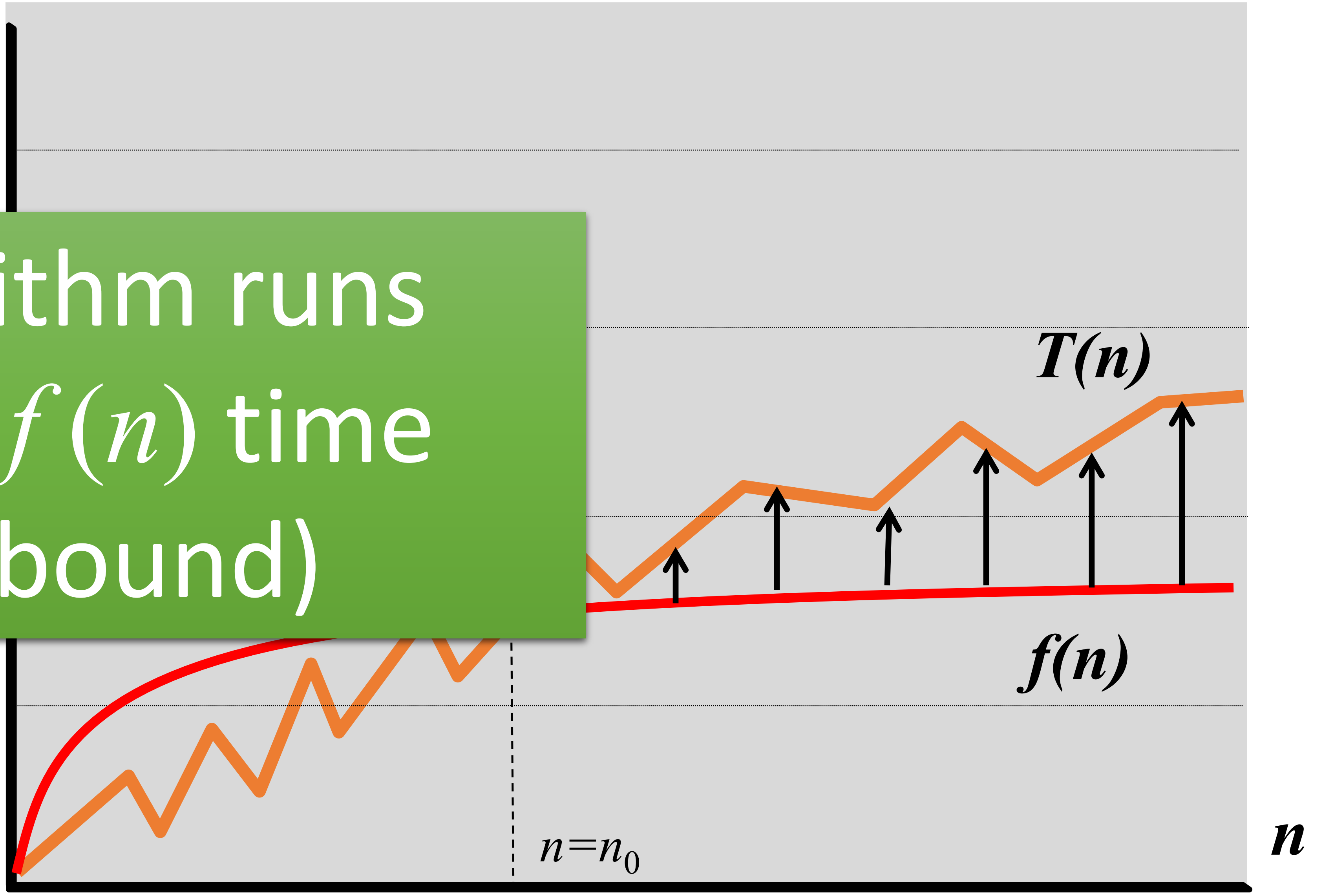
Big-O Says



This algorithm runs
at most in $f(n)$ time
(Upper bound)

How about lower bound?

This algorithm runs **at least** in $f(n)$ time (Lower bound)



Big Ω Notation Definition (Formal)

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

$T(n)$	$f(\mathrm{n})$	big-O
$T(n) = 1000n$	$f(n) = 1$	$T(n) = \Omega(1)$
$T(n) = n$	$f(n) = n$	$T(n) = \Omega(n)$
$T(n) = n^2$	$f(n) = n$	$T(n) = \Omega(n)$
$T(n) = 13n^2 + n$	$f(n) = n^2$	$T(n) = \Omega(n^2)$

Excercise

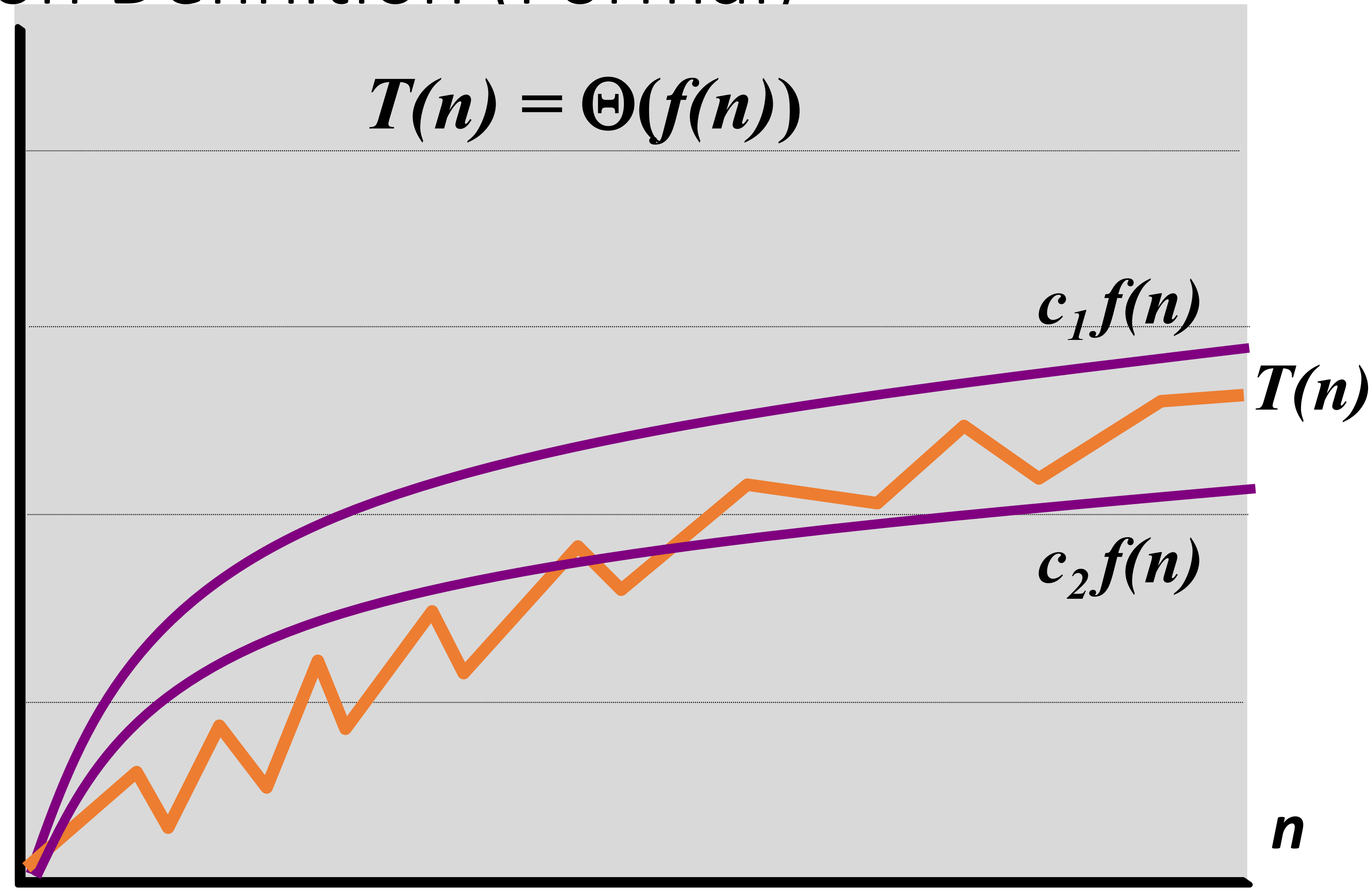
- True or false:

$$\text{“}f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))\text{”}$$

- Prove that your claim is correct using the definitions of O and Ω or by giving an example.

Big Θ Notation Definition (Formal)

- $T(n) = \Theta(f(n))$ if and only if:
  - $T(n) = O(f(n))$, and
  - $T(n) = \Omega(f(n))$



Example

$T(n)$	$f(n)$	big-O
$T(n) = 1000n$	$f(n) = n$	$T(\mathrm{n}) = \Theta(n)$
$T(n) = n$	$f(n) = 1$	$T(n) \neq \Theta(1)$
$T(n) = 13n^2 + n$	$f(n) = n^2$	$T(n) = \Theta(n^2)$
$T(n) = n^3$	$f(n) = n^2$	$T(n) \neq \Theta(n^2)$