CS2040S
Data Structures and Algorithms

Welcome!

Today: How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

Administrativia

Submit your tutorial choices by 23:59 TODAY.

Submit two sets of choices:

1. First choices.
2. All feasible choices.

More first choices ➔ more likely to get.

DO NOT select a choice that conflicts.

Administrativia

Available slots: 8am Thursday, Friday

Facts:

1. Many of you do not like 8am slots.
2. I do not like 8am slots.
3. Many of you will be in 8am slots.

SoC has limited classroom availability, so there is nothing we can do about this.

We will only consider appeals with proof of conflict.

Administrativia

Recitation appeals

Procedure:

1. Fill out (new) survey on Coursemology.
2. Only fill out survey if you have a problem.
3. We will not accept appeals changing out of 8am slot due to lack of sleep. Sorry. :)

Today: How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

Algorithm Analysis

Which takes longer?

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

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

$100k$ push operations

k^2 push operations

Which grows faster?

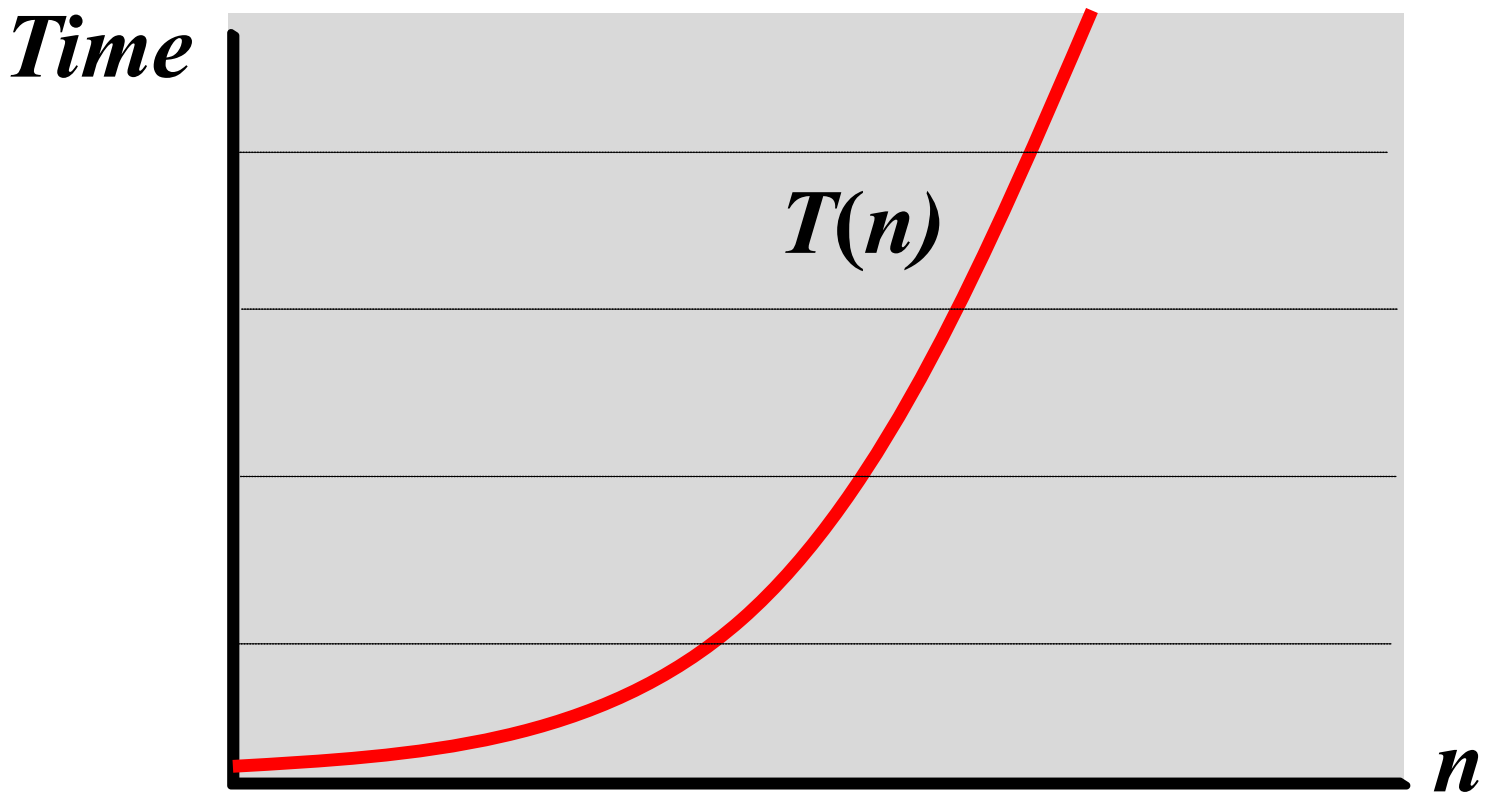
$T(k) = 100k$	$T(k) = k^2$
$T(0) = 0$	$T(0) = 0$
$T(1) = 100$	$T(1) = 1$
$T(100) = 10{,}000$	$T(100) = 10{,}000$
$T(1000) = 100{,}000$	$T(1000) = 1{,}000{,}000$

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

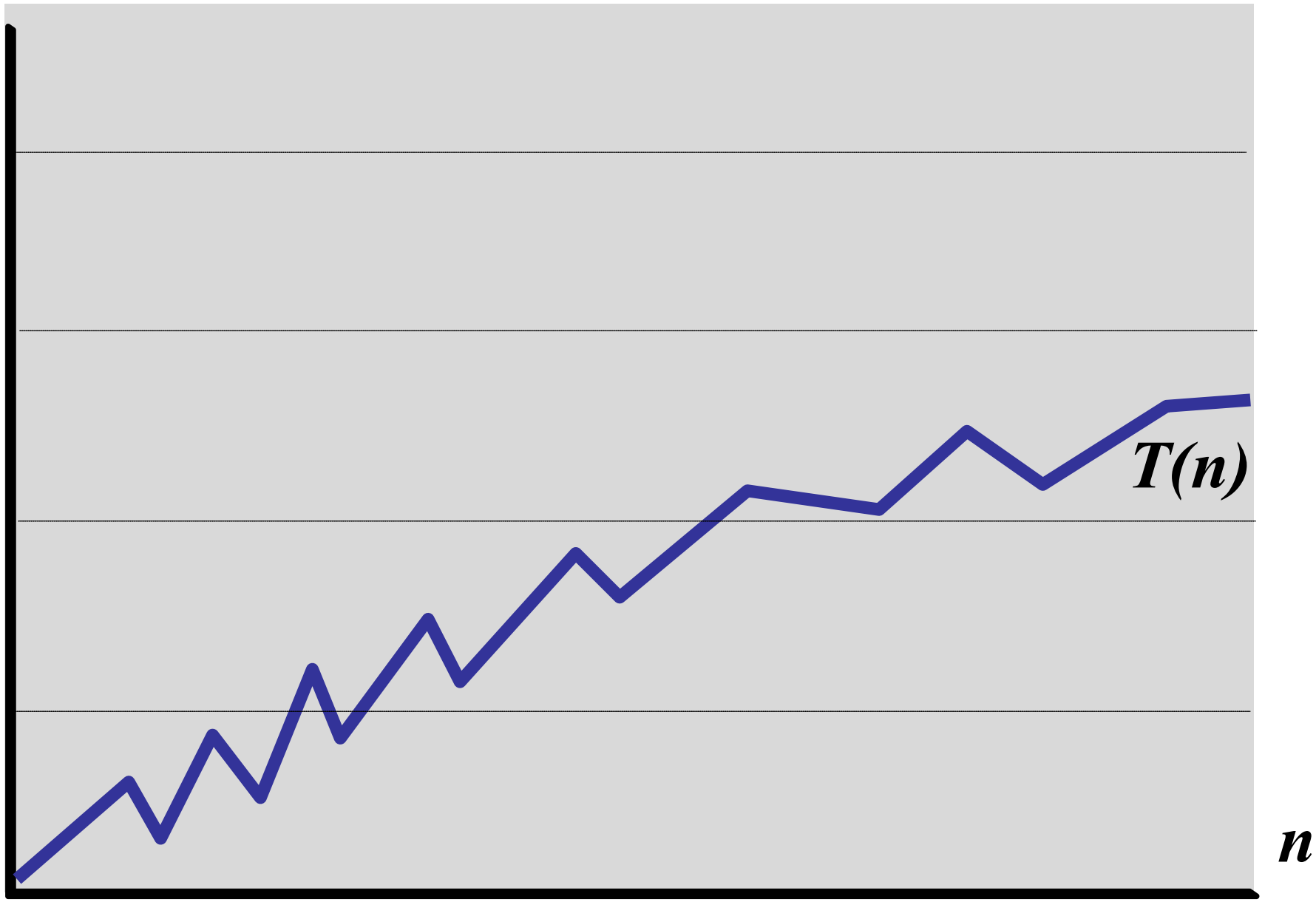
$T(n) = O(f(n))$ if:

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

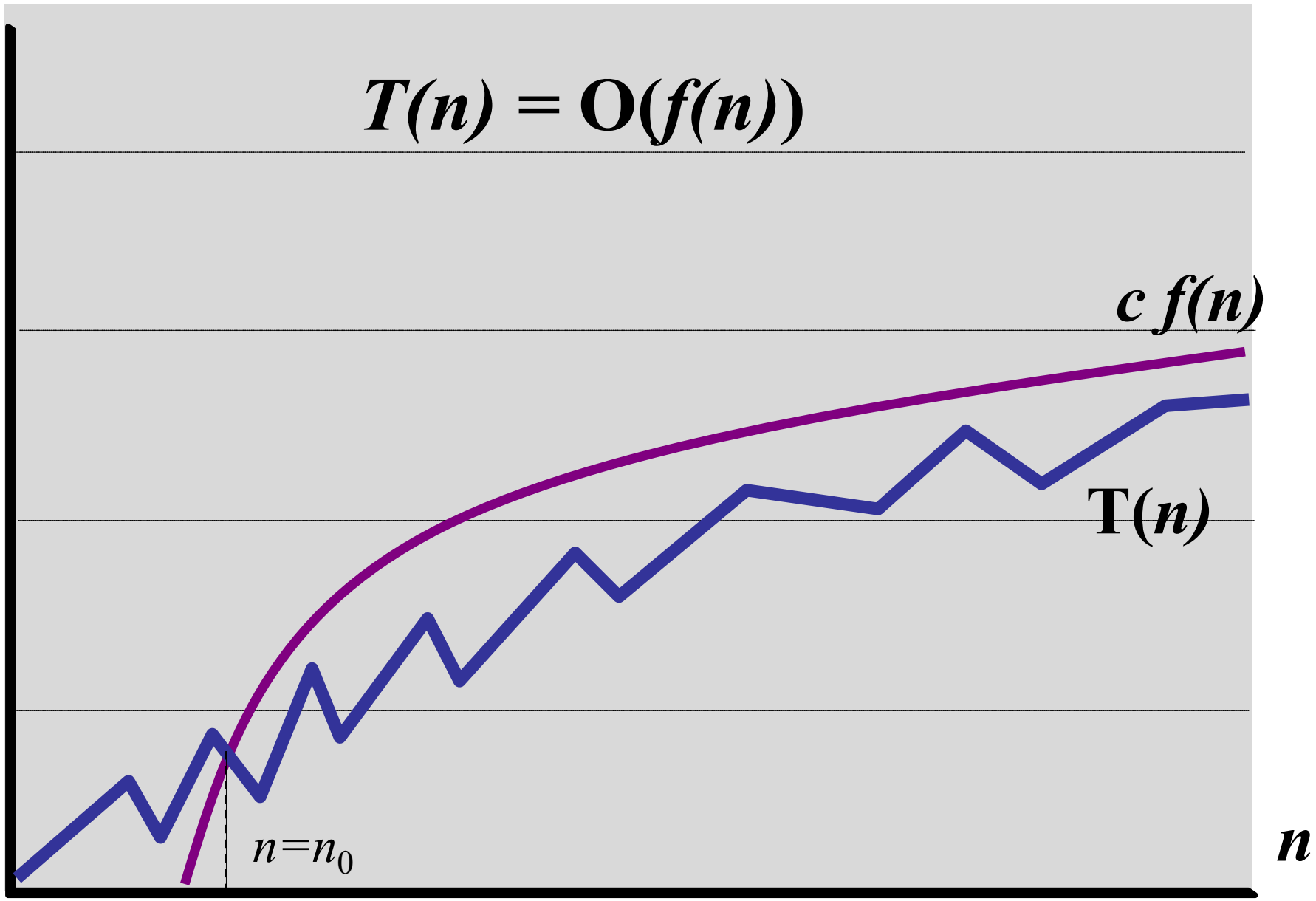
such that for all $n > n_0$:

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

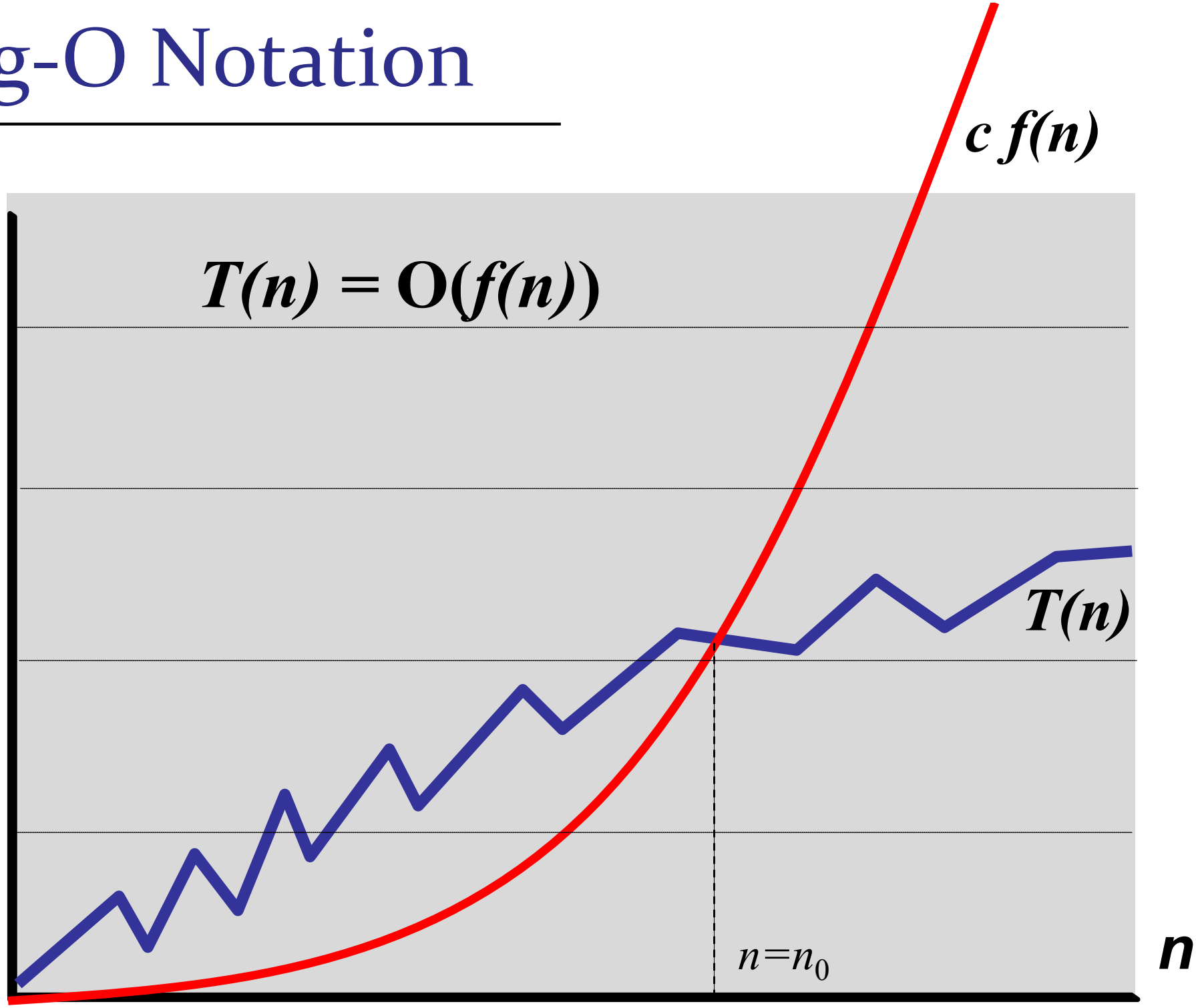
Big-O Notation



Big-O Notation



Big-O Notation



Big-O Notation

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

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

Example

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

Not tight

Big-O Notation

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

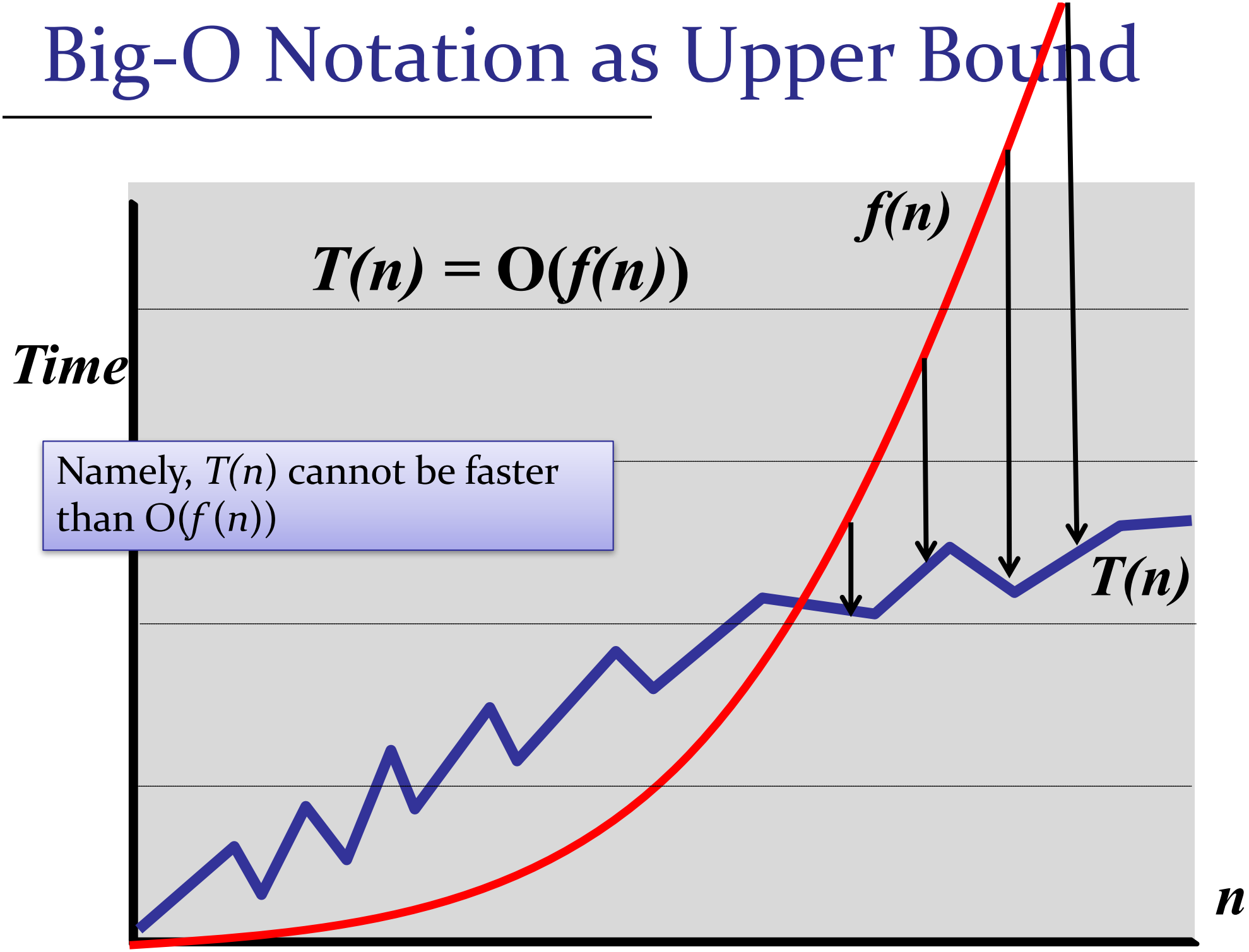
$T(n) = O(f(n))$ if:

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

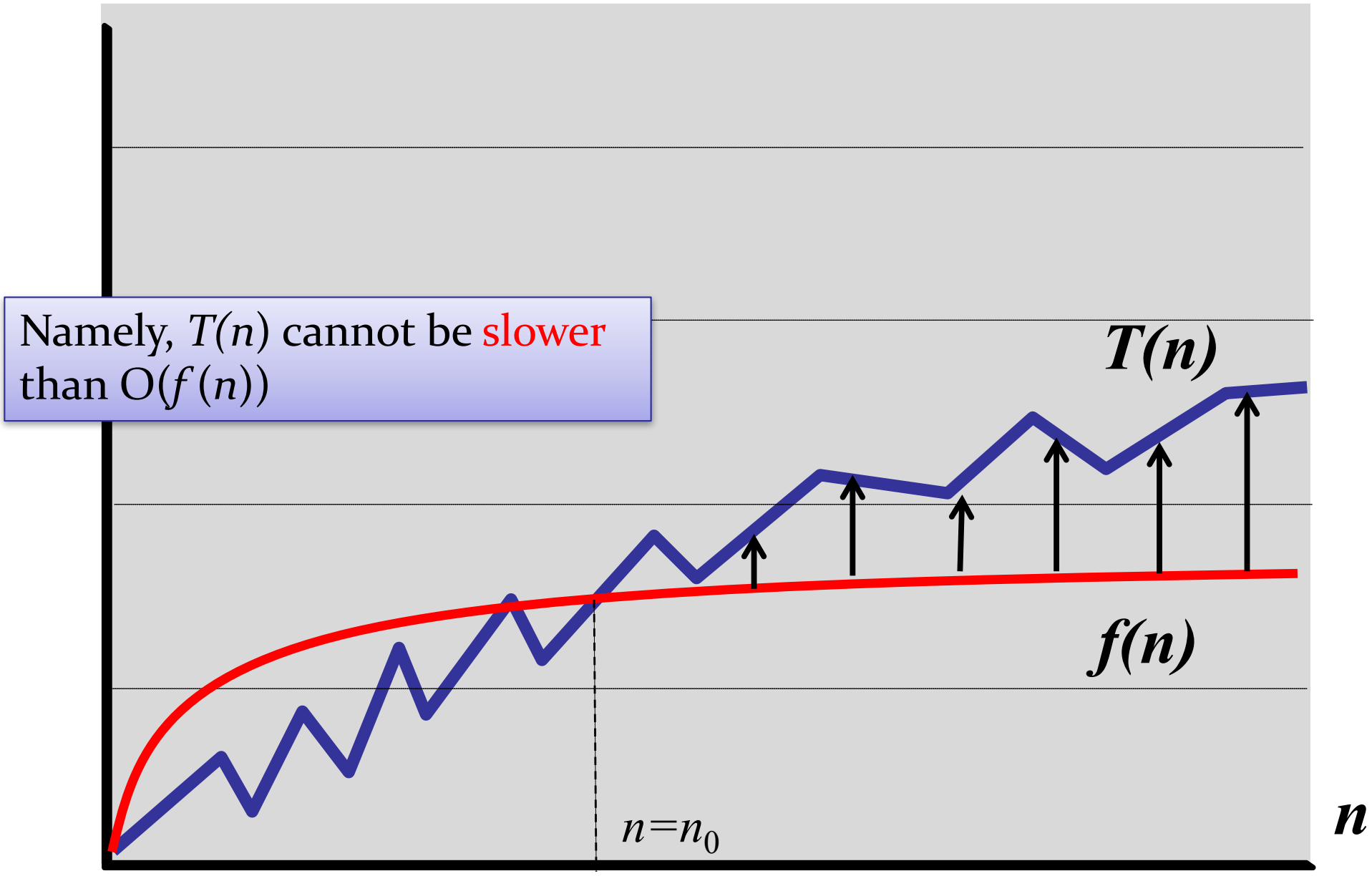
such that for all $n > n_0$:

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

Big-O Notation as Upper Bound



How about Lower bound?



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

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

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 Ω or by giving an example.

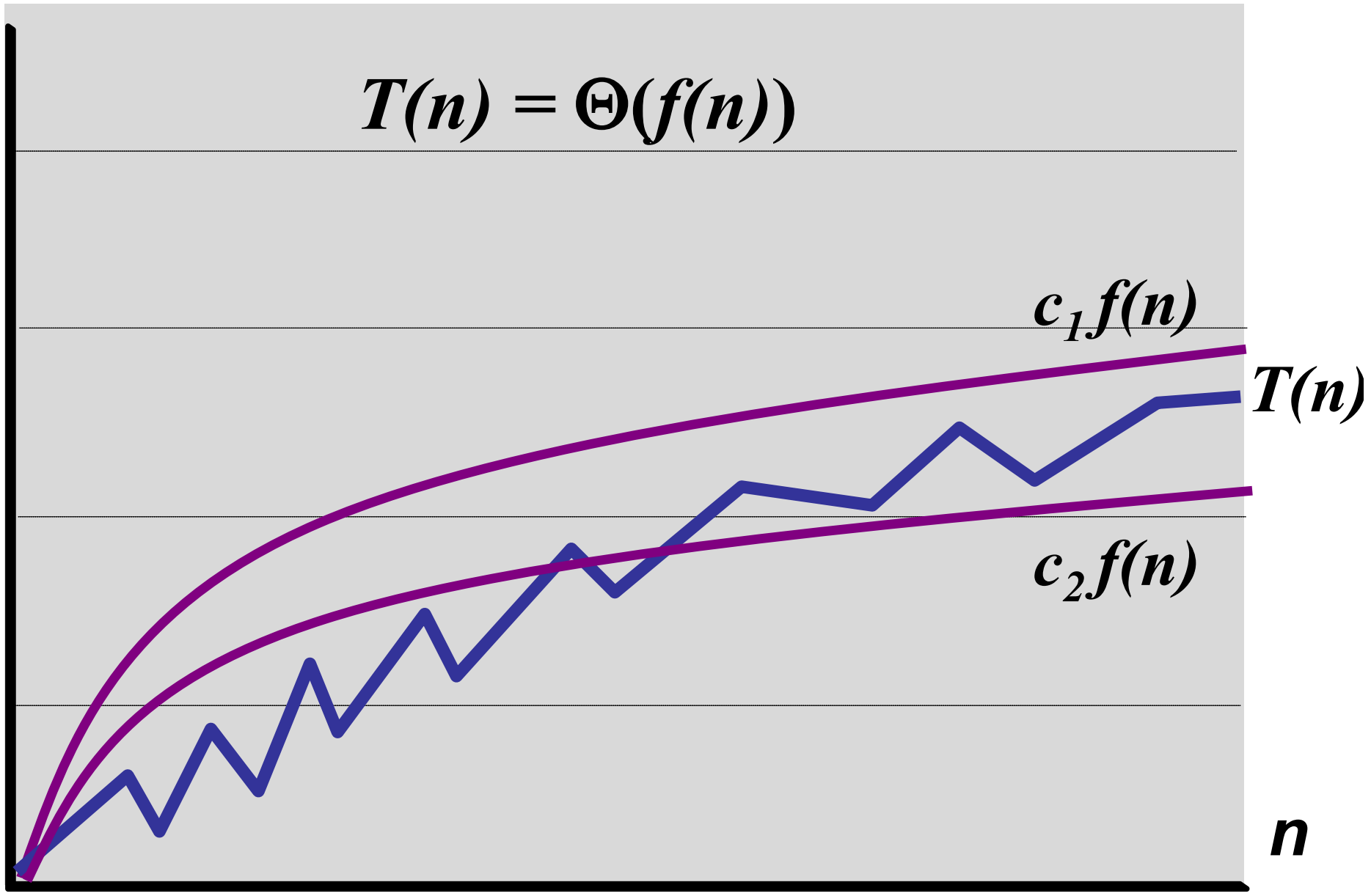
Big-O Notation

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

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

Big-O Notation

Rules:

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

$$T(n) = \mathrm{O}(n^k)$$

Example:

$$10n^5 + 50n^3 + 10n + 17 = \mathrm{O}(n^5)$$

Big-O Notation

Rules:

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

$$\boldsymbol{T(n) + S(n) = O(f(n) + g(n))}$$

Example:

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

$5n = O(n)$

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

Big-O Notation

Rules:

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

$$\boldsymbol{T(n)*S(n) = O(f(n)*g(n))}$$

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. $O(n^4)$

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

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. $O(n^n)$

$\log(n!) =$

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

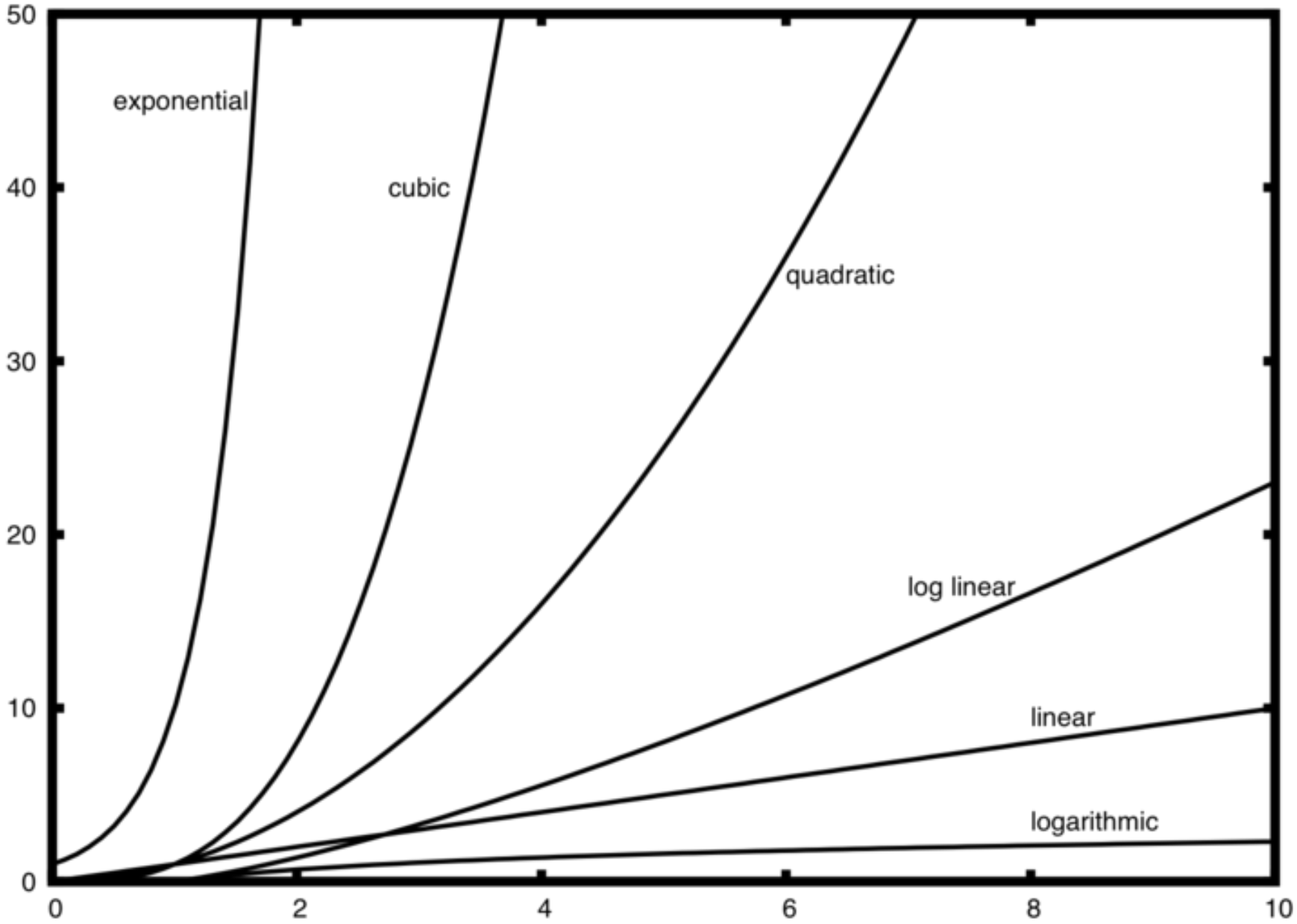
$\log(n!) =$

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


50
40
30
20
10
0
0
2
4
6
8
10
exponential
cubic
quadratic
log linear
linear
logarithmic

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

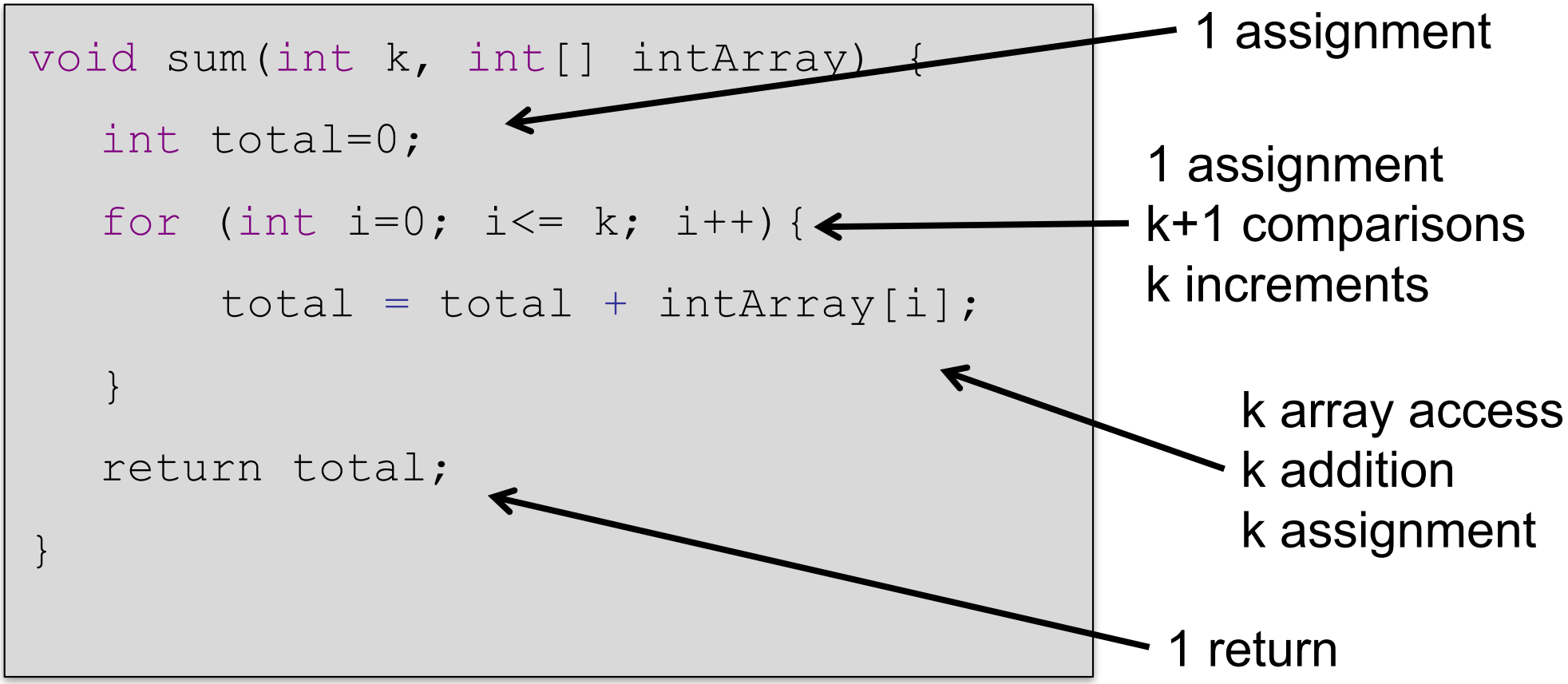
- One thing at a time

- All operations take constant time

  Addition, subtraction, multiplication, comparison

Algorithm Analysis

Example:



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

Algorithm Analysis

Example:

What is the cost of this operation?

```
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

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)(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)

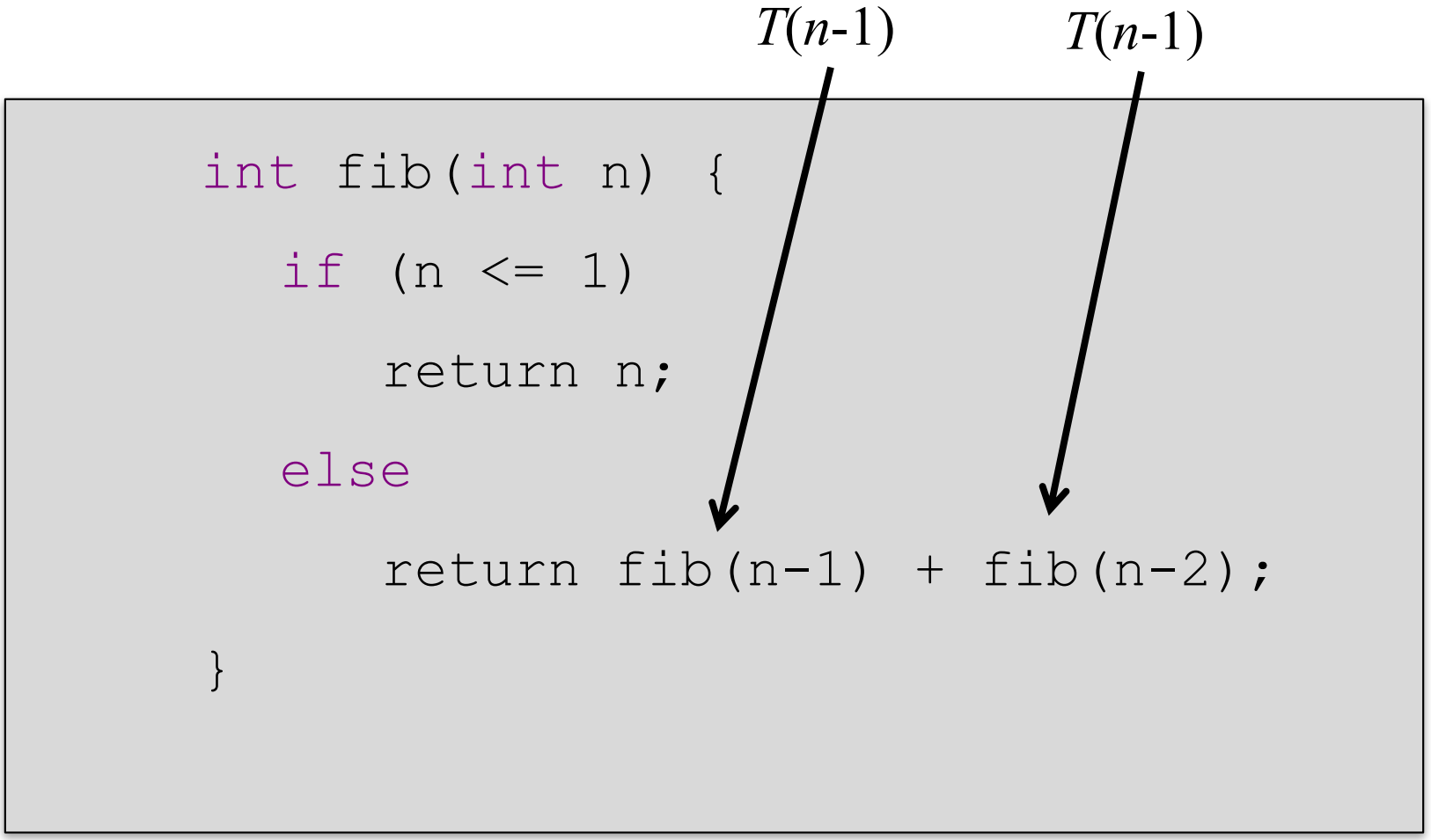
```
void sum(int k, int[] intArray) {
   if (k > 100)
       doExpensiveOperation();
   else
       doCheapOperation();
   return;
}
```

Rules

For recursive function calls.....

Recurrences

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



What is the running time?

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

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

Today: Divide and Conquer!

Algorithm Analysis

- Big-O Notation
- Model of computation

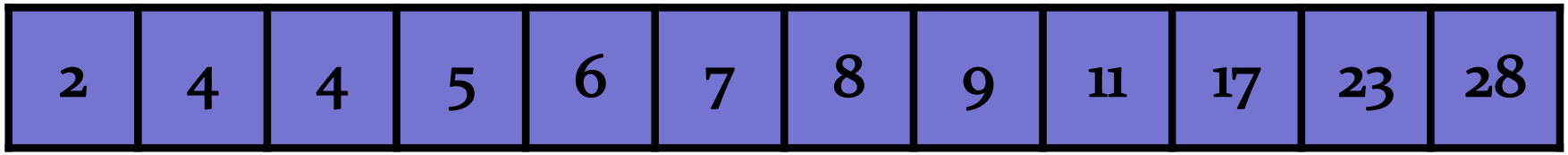
Searching

Peak Finding

- 1-dimension
- 2-dimensions

Binary Search

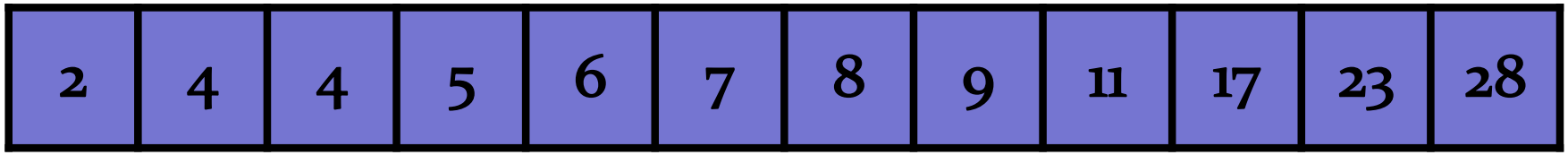
Sorted array: `A[1..n]`



Search for *k* in array A.

Binary Search

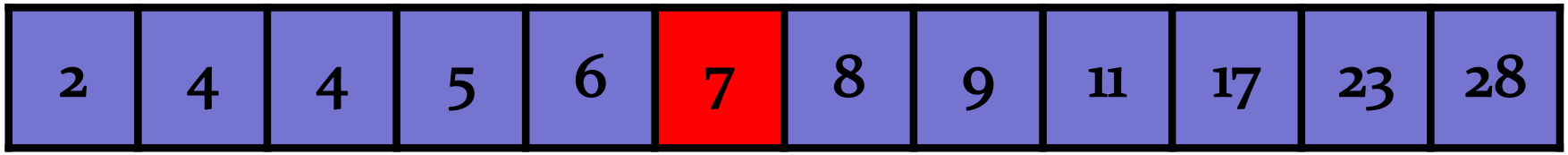
Sorted array: `A[1..n]`



Search for 17 in array A.

Binary Search

Sorted array: `A[1..n]`

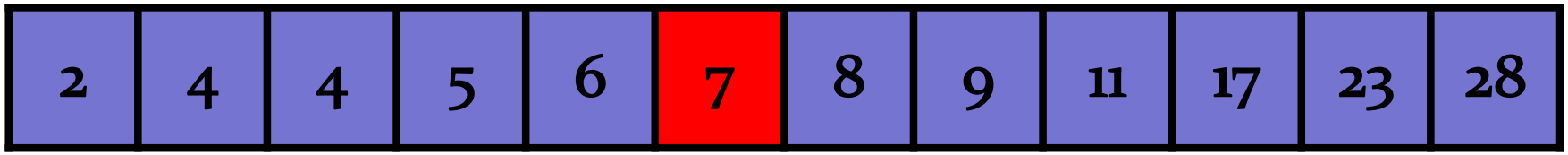


Search for 17 in array A.

- Find middle element: 7

Binary Search

Sorted array: `A[1..n]`

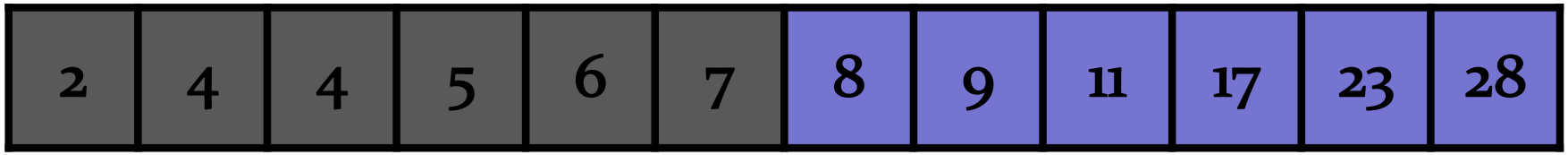


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

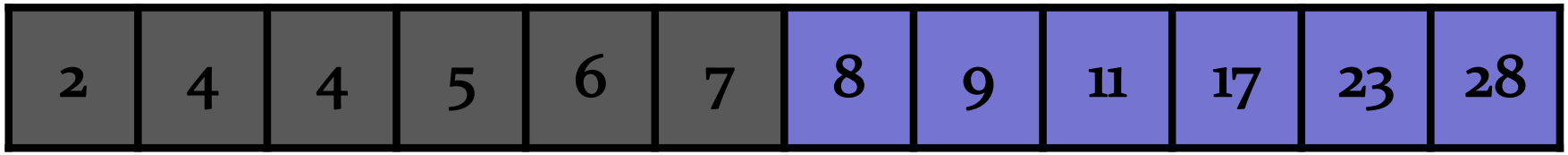


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

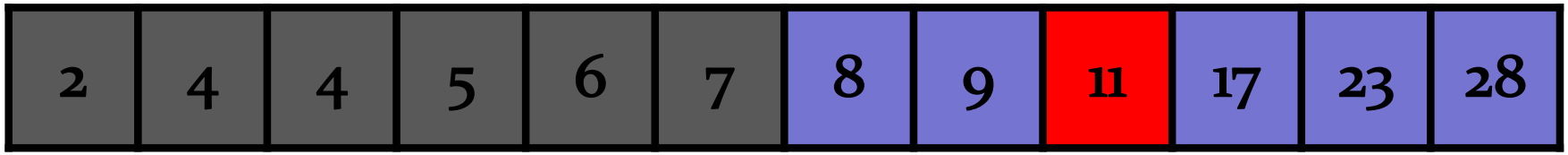


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

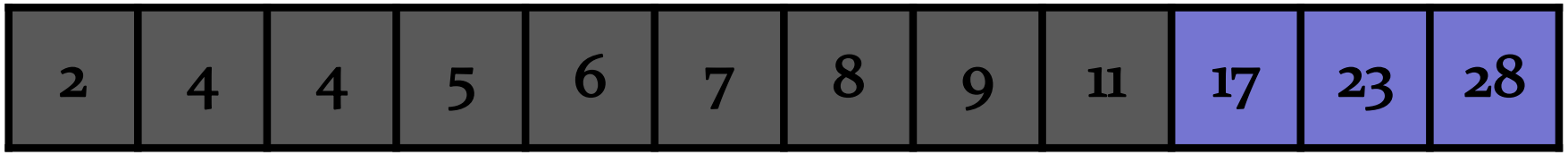


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

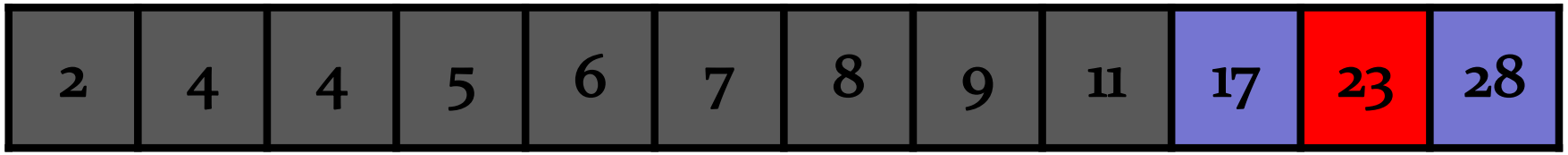


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

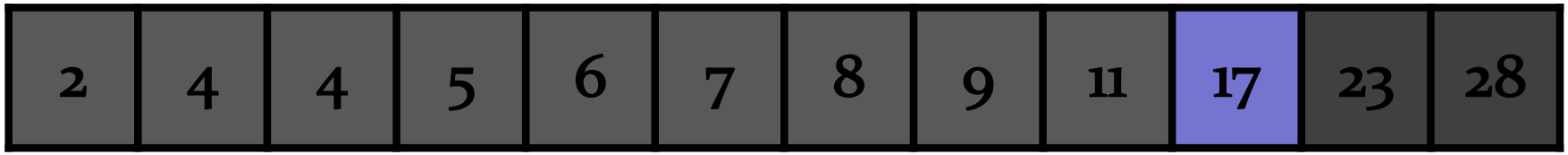


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

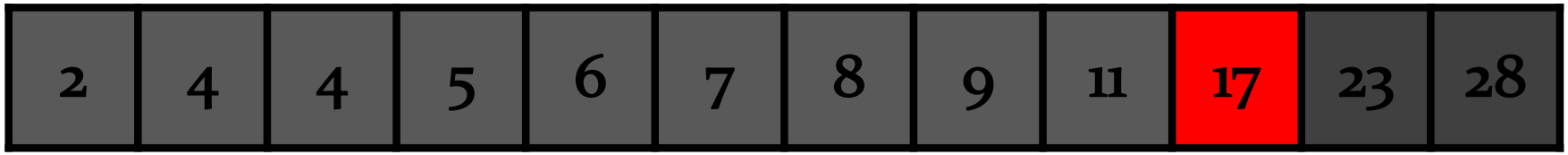


Search for 17 in array A.

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

Binary Search

Sorted array: `A[1..n]`

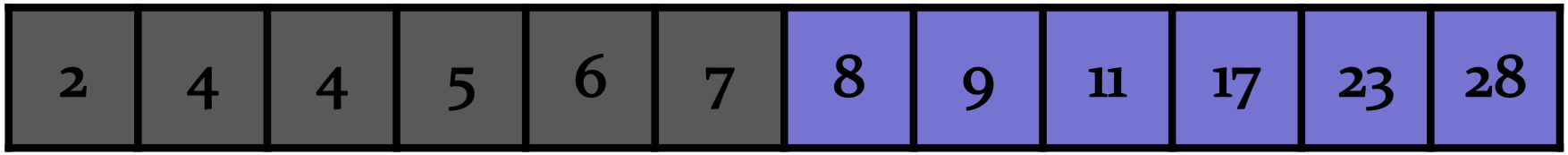


Search for 17 in array A.

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

Problem Solving: Reduce the Problem

Sorted array: `A[1..n]`



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

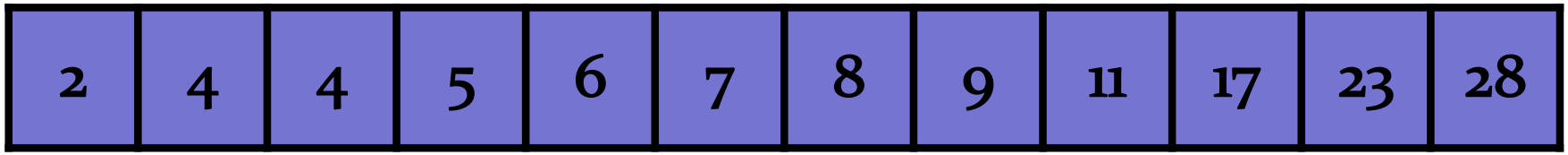
I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudo-code was fine. At the end of the specified time, almost all 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)

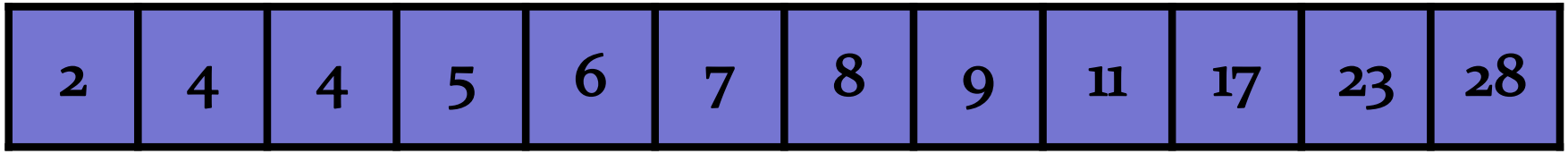
Sorted array: `A[1..n]`




```
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[begin]
```

Binary Search (buggy)

Sorted array: `A[1..n]`




```
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]
```

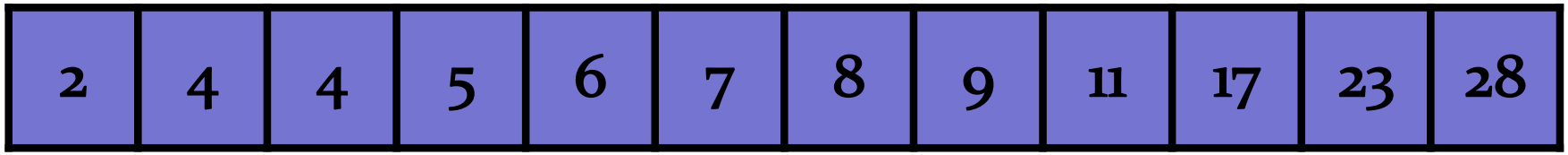
May not terminate!

Round down?

Reduce end < begin?

Binary Search (buggy)

Sorted array: `A[1..n]`

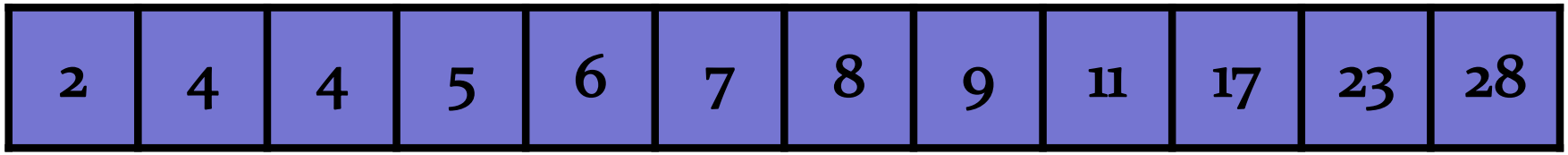



```
Search(A, key, n)
    begin = 0
    end = n
    while begin != end do:
        if  key < A[(begin+end)/2] then
            end = (begin+end)/2 - 1
        else begin = (begin+end)/2
    return A[begin]
```

A[begin] == key?

Binary Search (buggy)

Sorted array: `A[1..n]`




```
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]
```

array out of bounds

Binary Search

Specification:

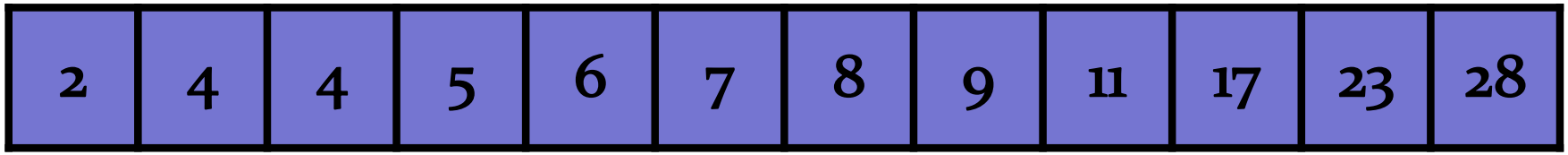
- Returns element if it is in the array.
- Returns “null” if it is not in the array.

Alternate Specification:

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

Binary Search

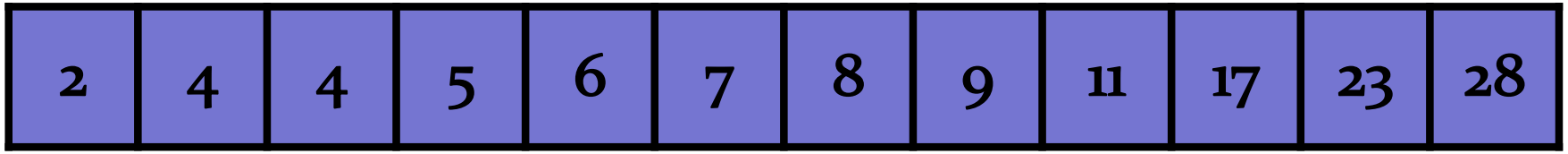
Sorted array: `A[1..n]`




```
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]
```

Binary Search

Sorted array: `A[1..n]`

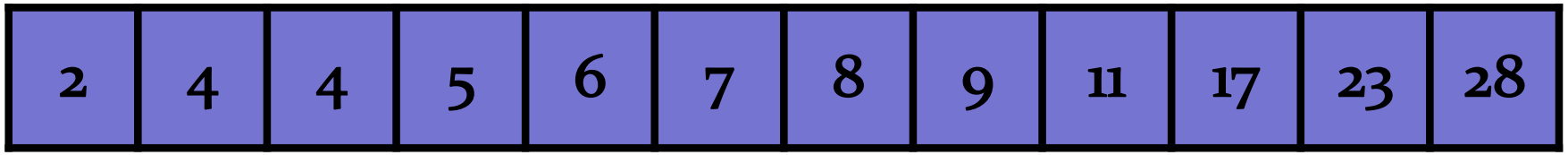



```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return A[begin]
```

less-than-or-equal

Binary Search

Sorted array: `A[1..n]`

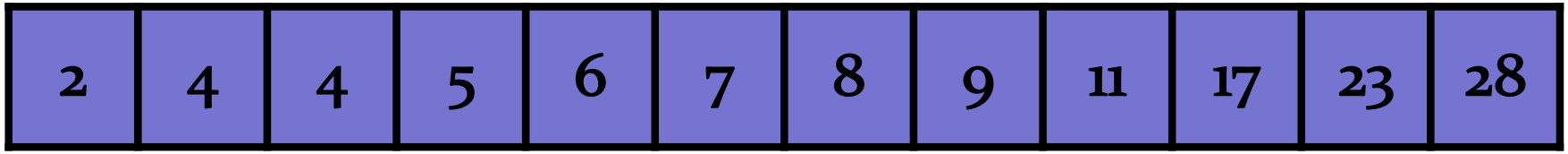



```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return A[begin]
```

strictly greater than

Binary Search

Sorted array: `A[1..n]`

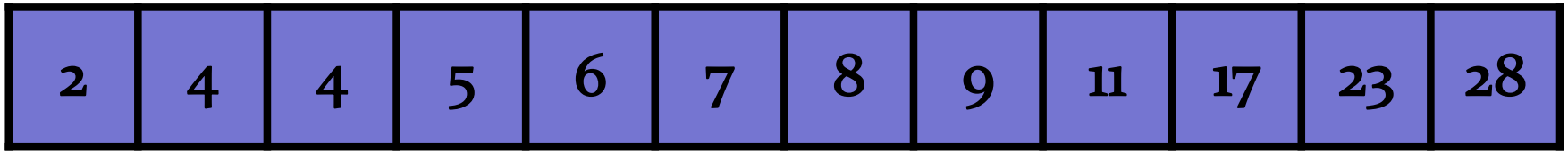



```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return A[begin]
```

Array of out bounds?

Binary Search

Sorted array: `A[1..n]`

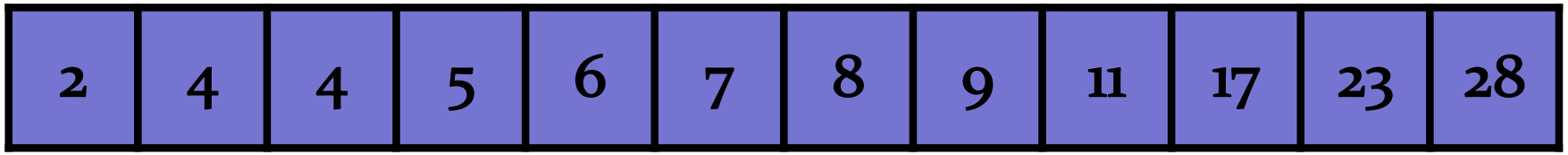



```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return A[begin]
```

Array of out bounds?
No: division rounds down.

Binary Search

Sorted array: `A[1..n]`

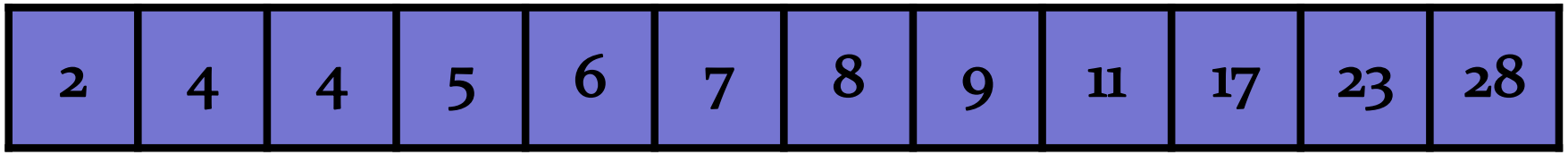



```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(beg
    return A[begin]
```

Oops, still has a bug…

Binary Search

Sorted array: `A[1..n]`

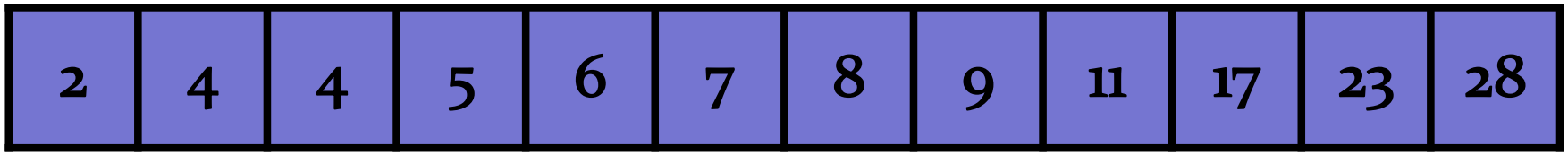



```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return A[begin]
```

What if begin > MAX_INT/2?

Binary Search

Sorted array: `A[1..n]`



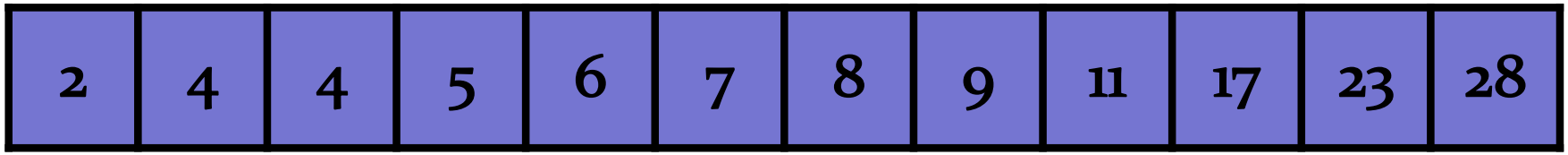

```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        if  key <= A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = 1+(begin+end)/2
    return A[begin]
```

What if begin > MAX_INT/2?

Overflow error: begin+end > MAX_INT

Binary Search

Sorted array: A[1..n]




```
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]
```

Precondition and Postcondition

Precondition:

- Fact that is true when the loop/method begins.

Postcondition:

- Fact that is true when the loop/method ends.

Loop Invariants

Invariant:

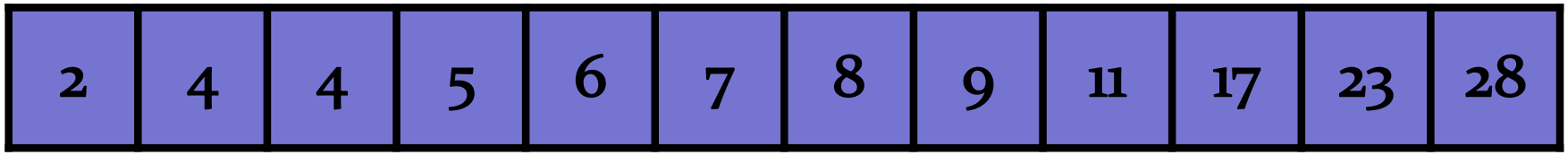
- relationship between variables that is always true.

Loop Invariant:

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

Binary Search

Sorted array: `A[1..n]`




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

Binary Search

Functionality:

- If element is in the array, return it.

Preconditions:

- Array is of size n
- Array is sorted

Postcondition:

- `A[begin] = key`

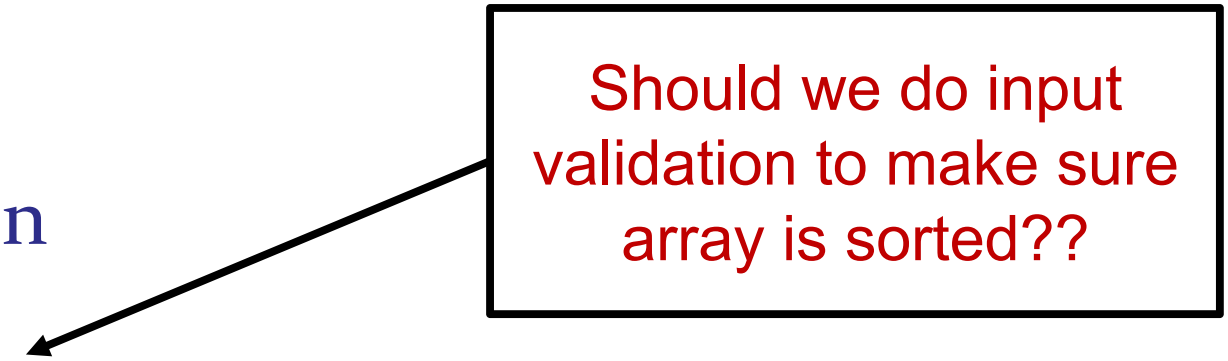
Binary Search

Functionality:

- If element is in the array, return it.

Preconditions:

- Array is of size n
- Array is sorted



Postcondition:

- `A[begin] = key`

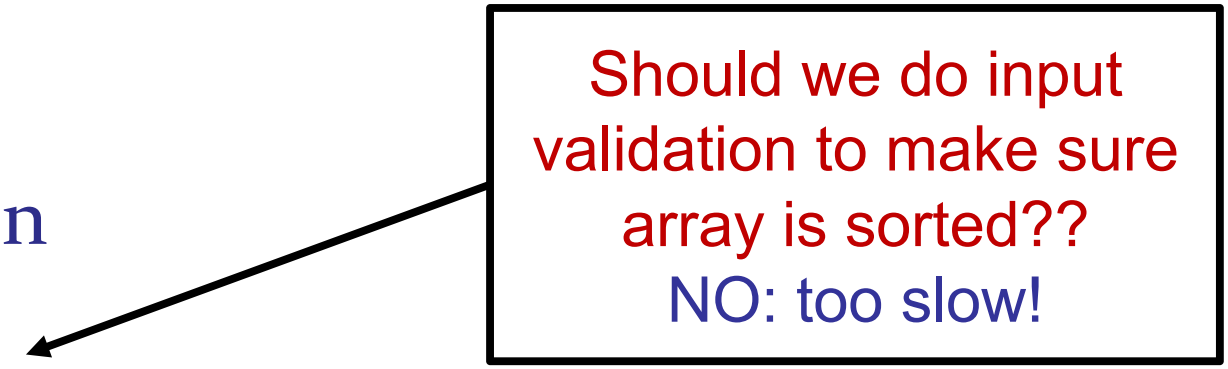
Binary Search

Functionality:

- If element is in the array, return it.

Preconditions:

- Array is of size n
- Array is sorted

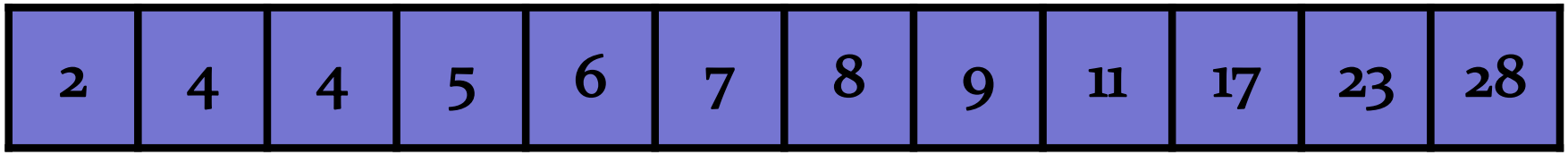


Postcondition:

- `A[begin] = key`

Binary Search

Sorted array: A[1..n]




```
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]
```

Binary Search

Loop invariant:

- `A[begin] ≤ key ≤ A[end]`

Interpretation:

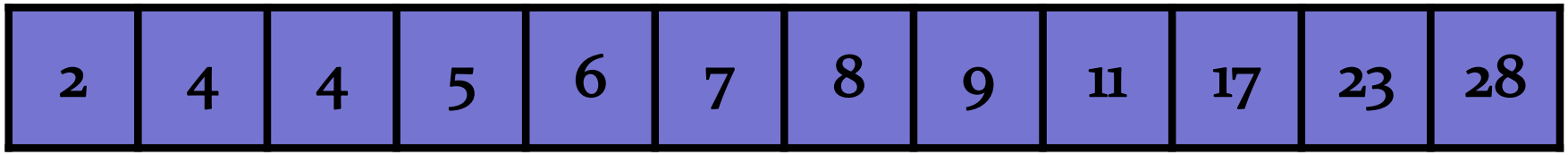
- The key is in the range of the array

Error checking:

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

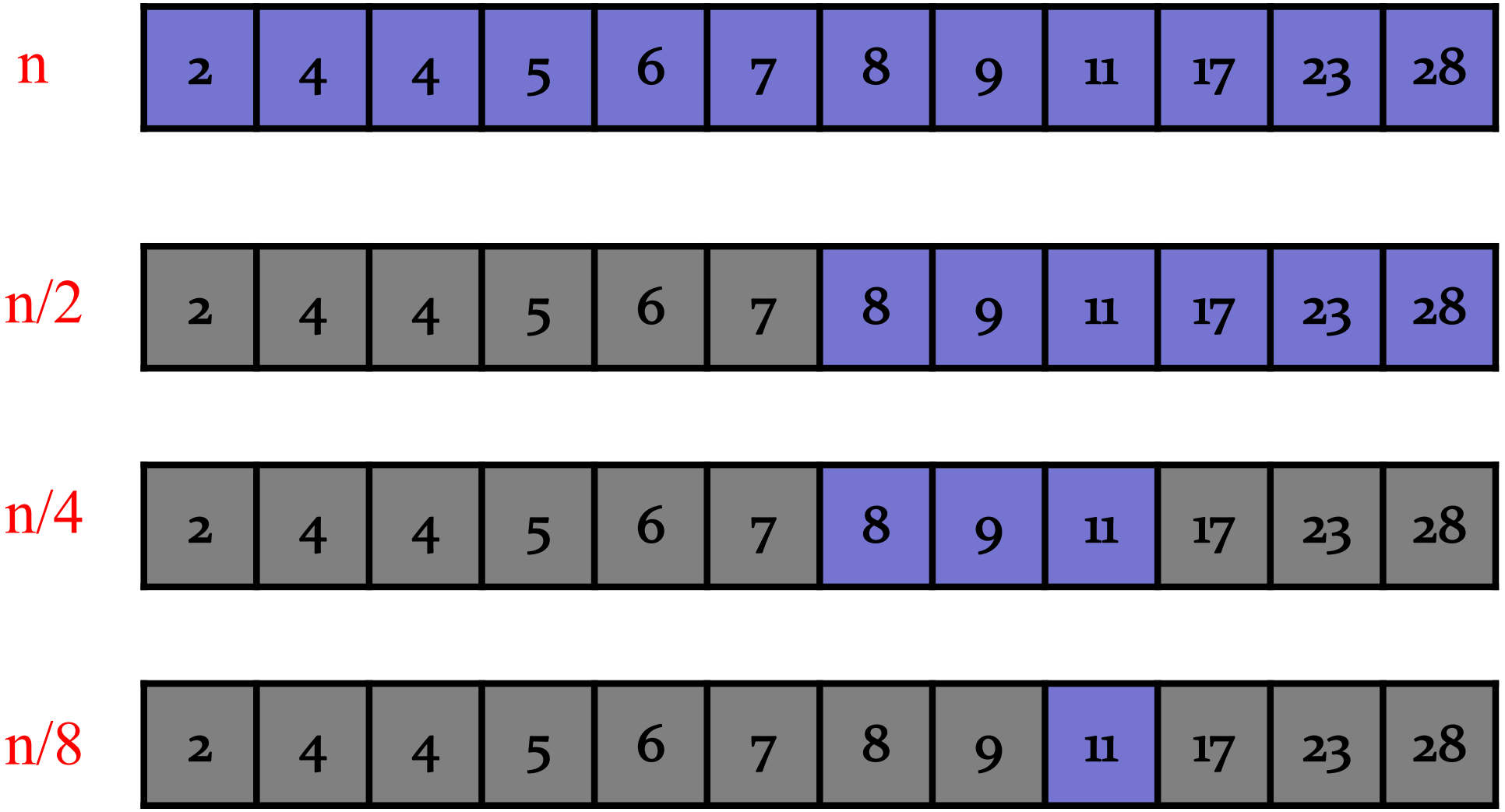
Binary Search

Sorted array: A[1..n]



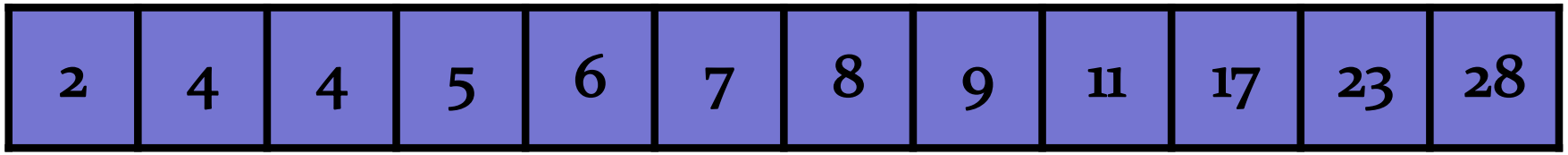

```
Search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if  key <= A[mid] then
            end = mid
        else begin = mid+1
    return A[begin]
```

Binary Search



Binary Search

Sorted array: `A[1..n]`



Iteration 1: (end – begin) = n

Iteration 2: (end – begin) = $n/2$

Iteration 3: (end – begin) = $n/4$

…

Iteration k: (end – begin) = $n/2^k$

Another invariant!

$n/2^k = 1$ ➔ $k = \log(n)$

Key Invariants:

Correctness:

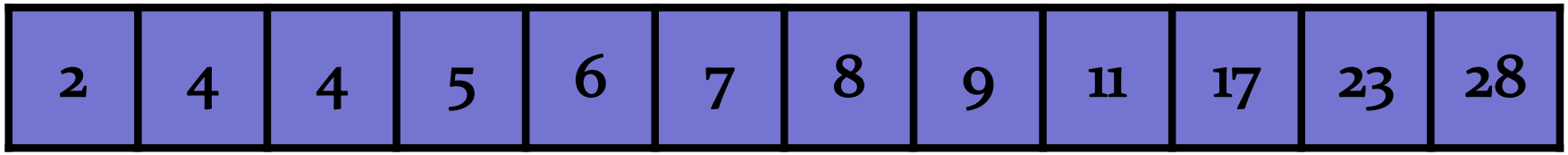
- `A[begin] ≤ key ≤ A[end]`

Performance:

- `(end-begin) <= n/2^k` in iteration k.

Binary Search

Sorted array: A[1..n]



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

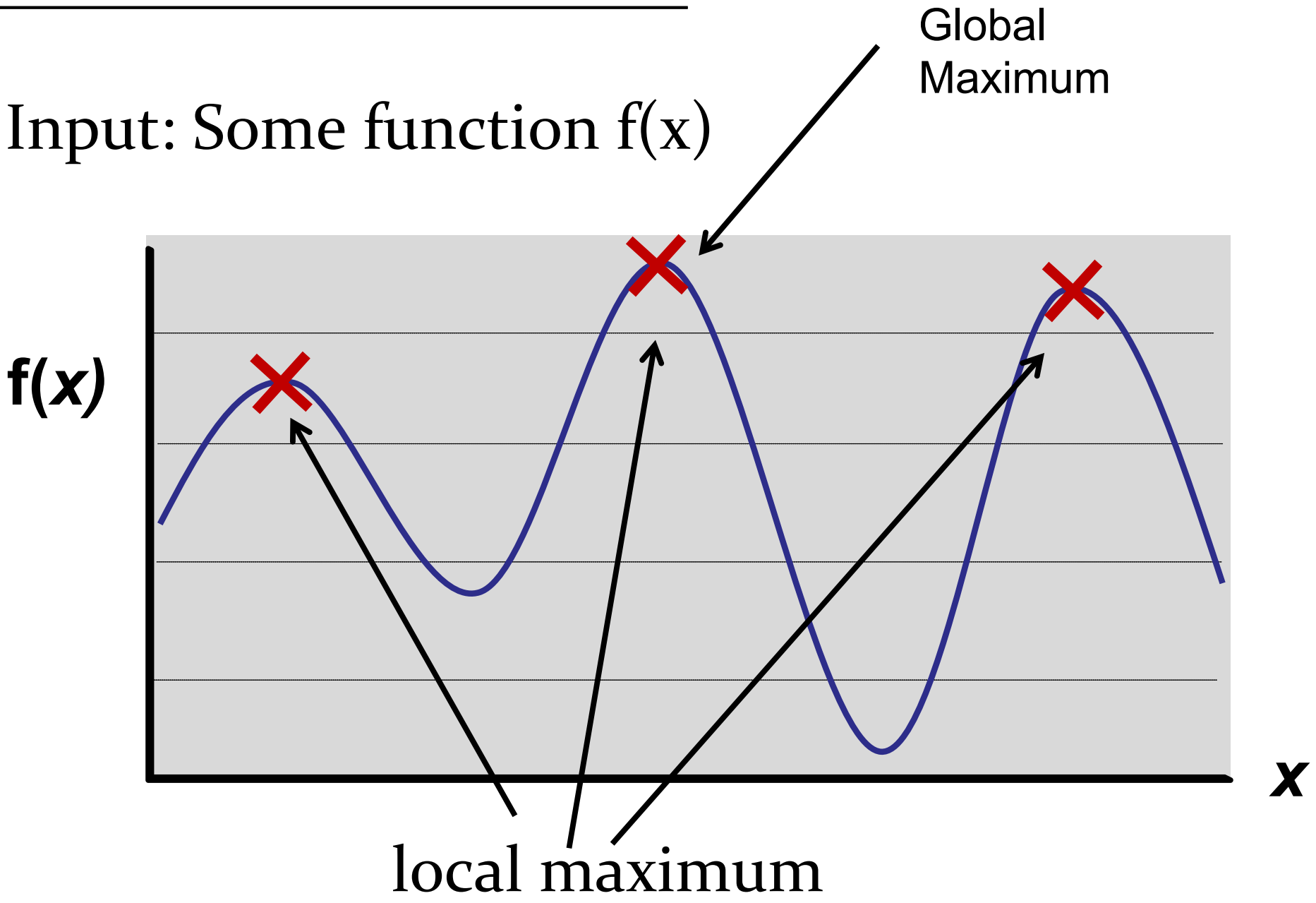
Searching

Peak Finding

- 1-dimension
- 2-dimensions

Peak Finding

Input: Some function f(x)



Peak Finding

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?

- Finds a *good enough* solution.
- Local maxima are close to the global maximum?
- Much, much faster.

Global Maximum

Input: Array $A[1..n]$

Output: global maximum element in A

How long to find a global maximum?

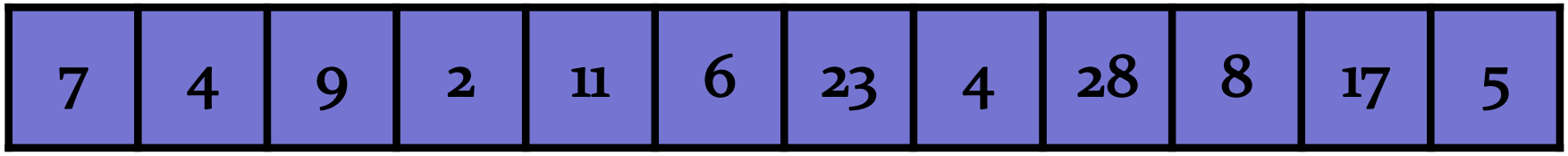
Input: Array $A[1..n]$

Output: maximum element in A

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

Global Maximum

Unsorted array: `A[1..n]`



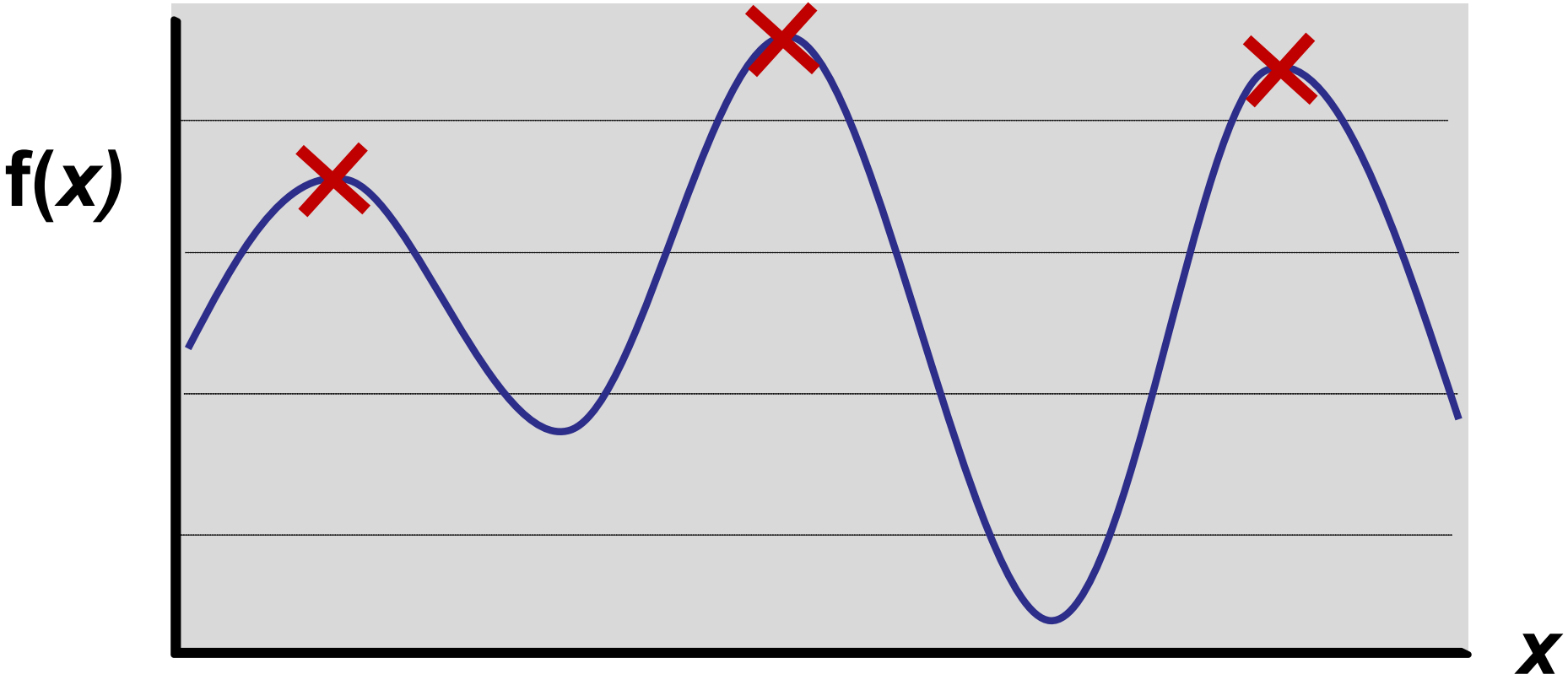

```
FindMax(A,n)
    max = A[1]
    for i = 1 to n do:
        if (A[i]>max) then max=A[i]
```

Too slow!

Time Complexity: $O(n)$

Peak (Local Maximum) Finding

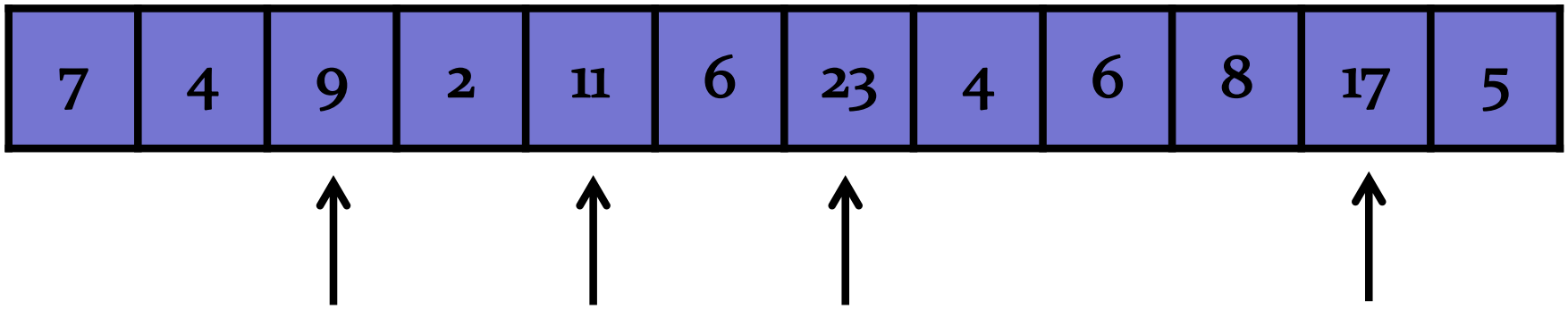
Input: Some function f(x)



Output: A local maximum

Peak Finding

Input: Some ~~function~~ array $A[1..n]$



Output: a local maximum in A

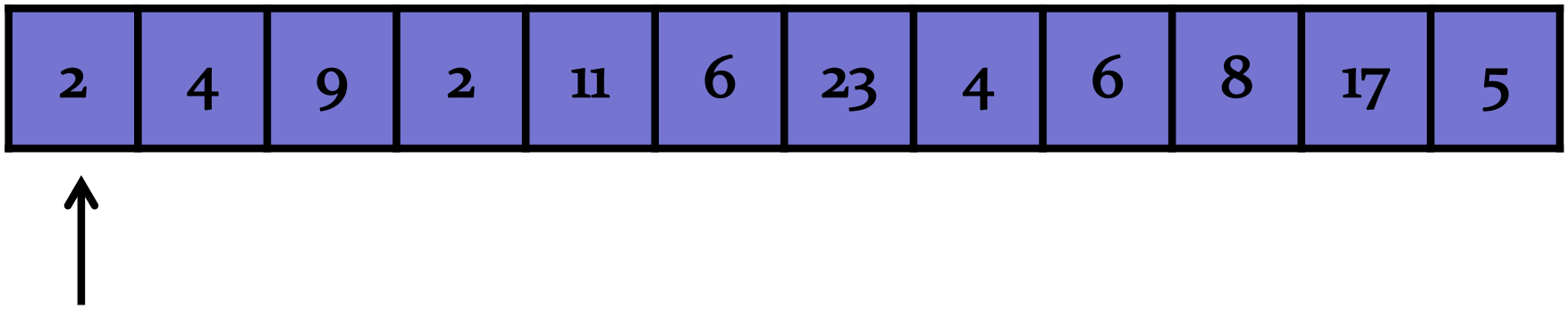
$A[i-1] \leq A[i]$ **and** $A[i+1] \leq A[i]$

Assume that

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

Peak Finding: Algorithm 1

Input: Some array `A[1..n]`

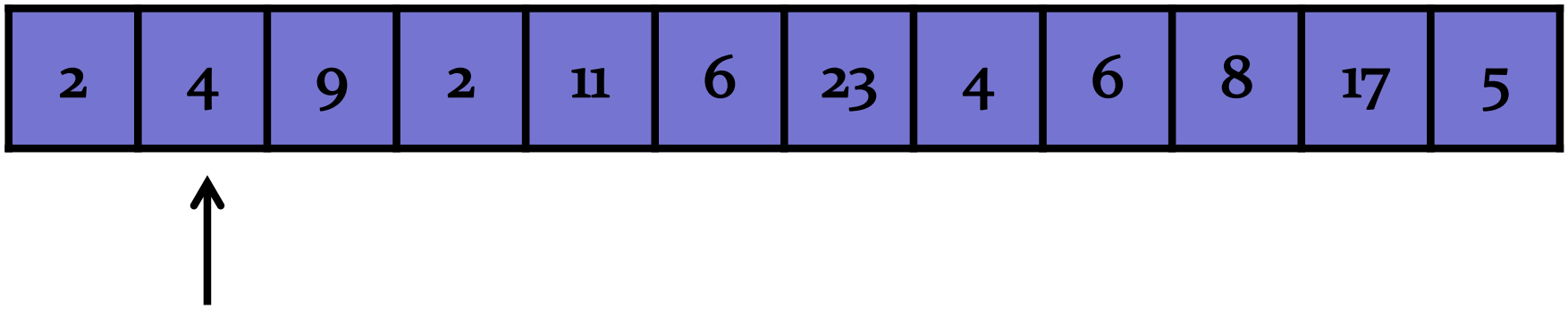


FindPeak

- Start from `A[1]`
- Examine every element
- Stop when you find a peak.

Peak Finding: Algorithm 1

Input: Some array `A[1..n]`

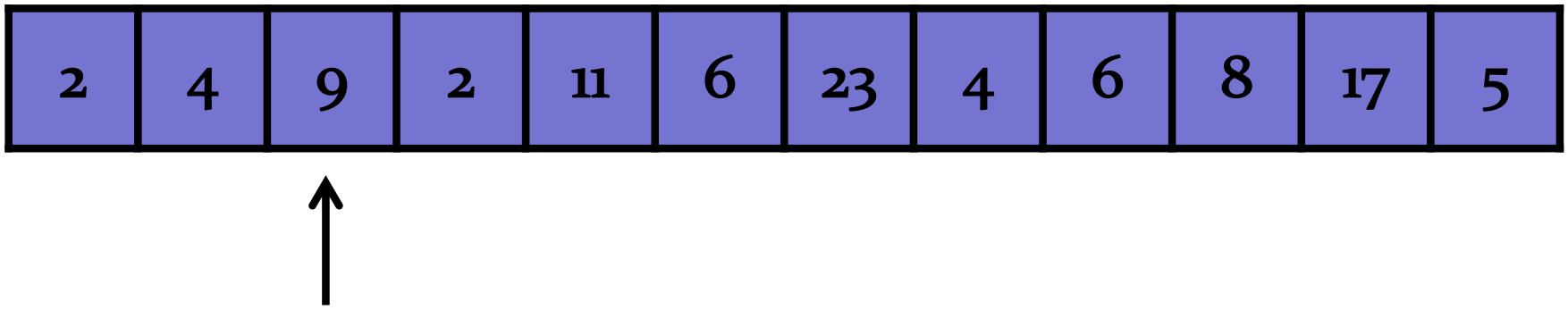


FindPeak

- Start from A[1]
- Examine every element
- Stop when you find a peak.

Peak Finding: Algorithm 1

Input: Some array A[1..n]

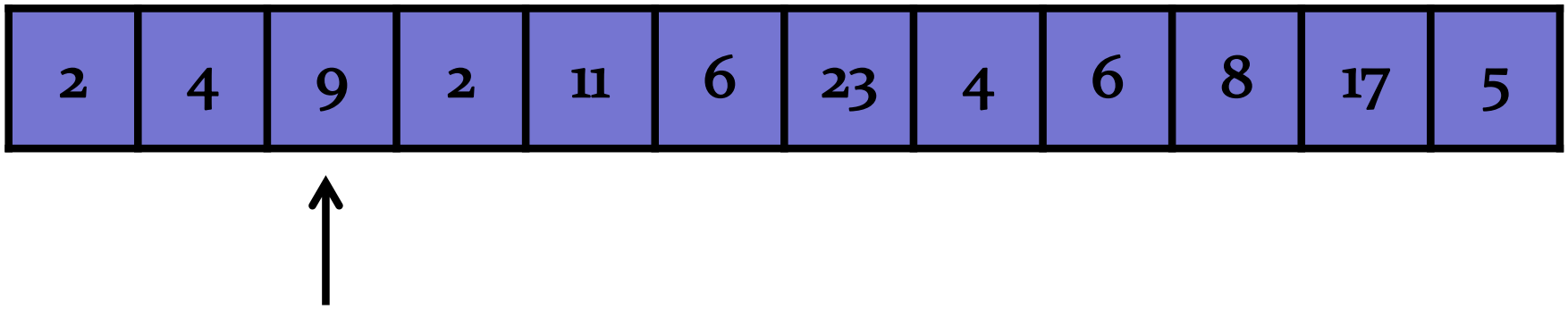


FindPeak

- Start from A[1]
- Examine every element
- Stop when you find a peak.

Peak Finding: Algorithm 1

Input: Some array A[1..n]

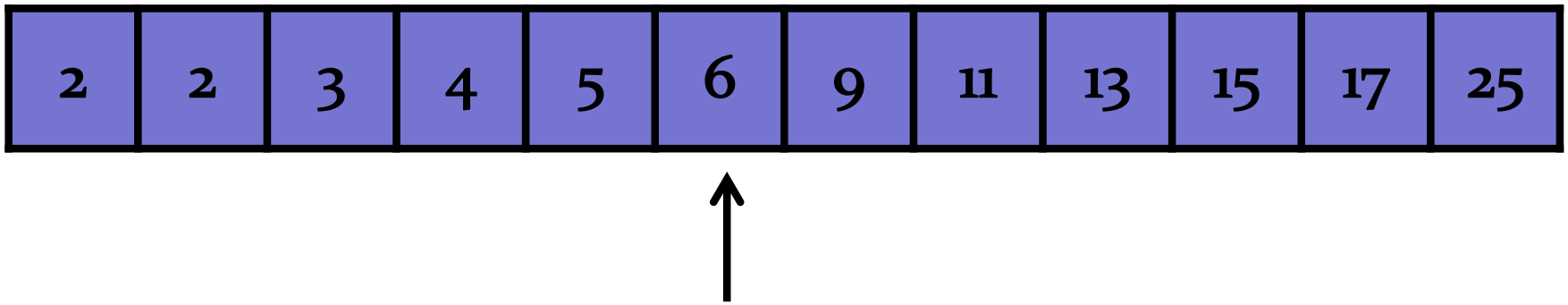


Running time: n

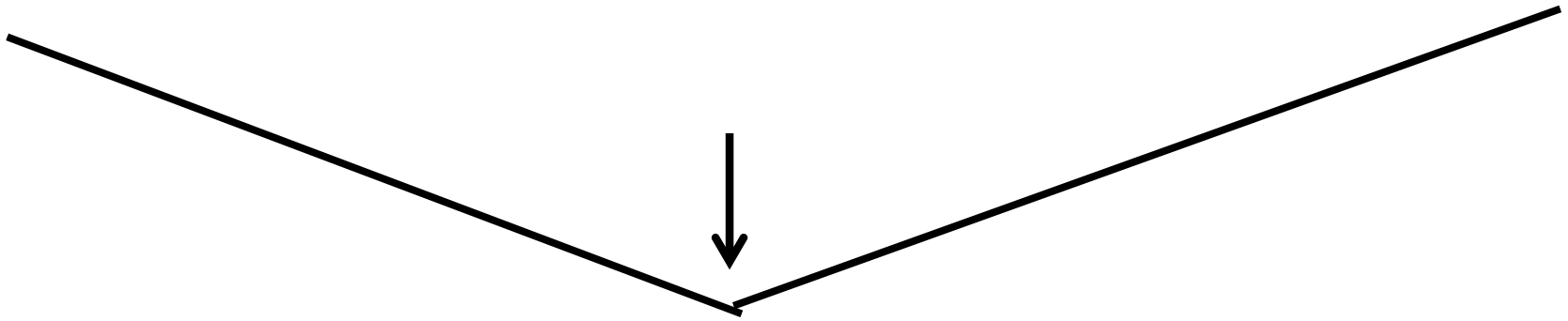
Simple improvement?

Peak Finding: Algorithm 1

Input: Some array A[1..n]



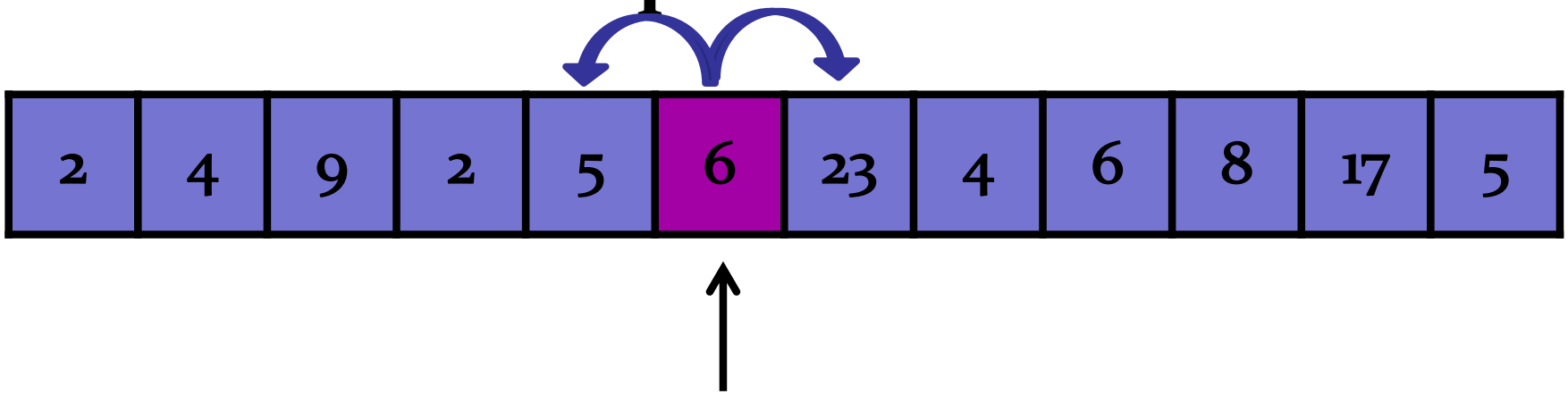
Start in the middle!



Worst-case: n/2

Peak Finding: Algorithm 2

Divide-and-Conquer

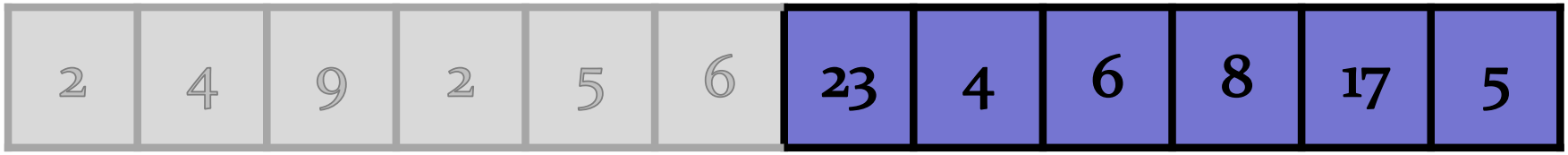


Start in the middle

5 < 6? ← **OK**

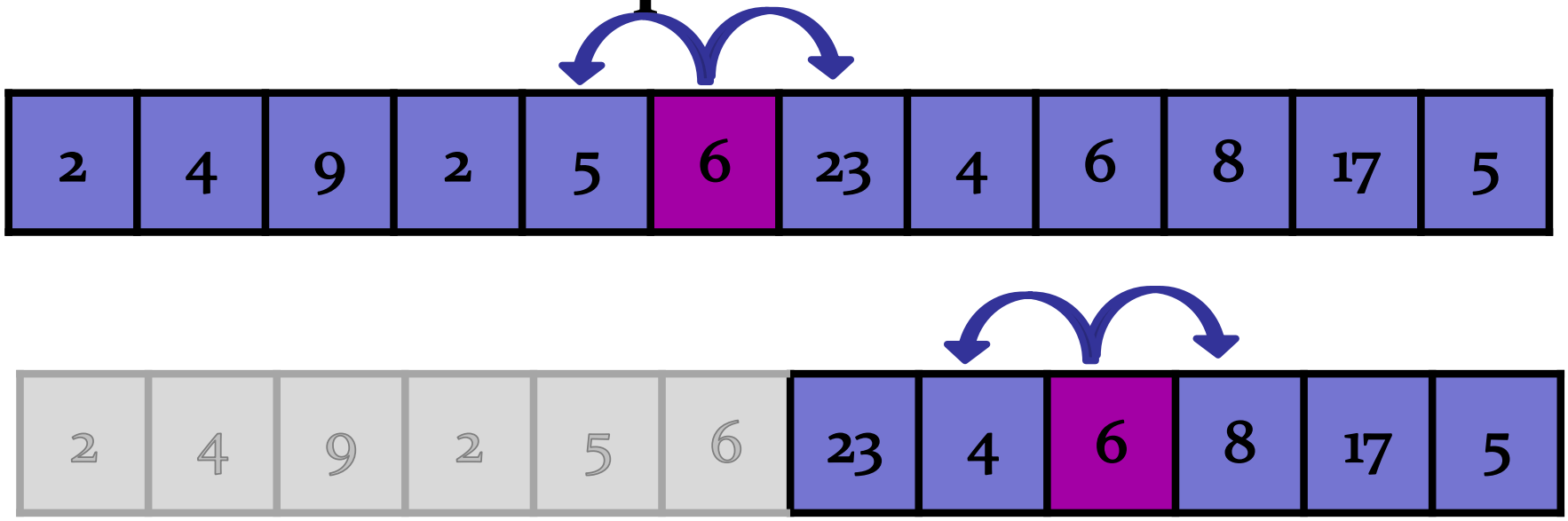
6 > 23? ← **NO**

Recurse!



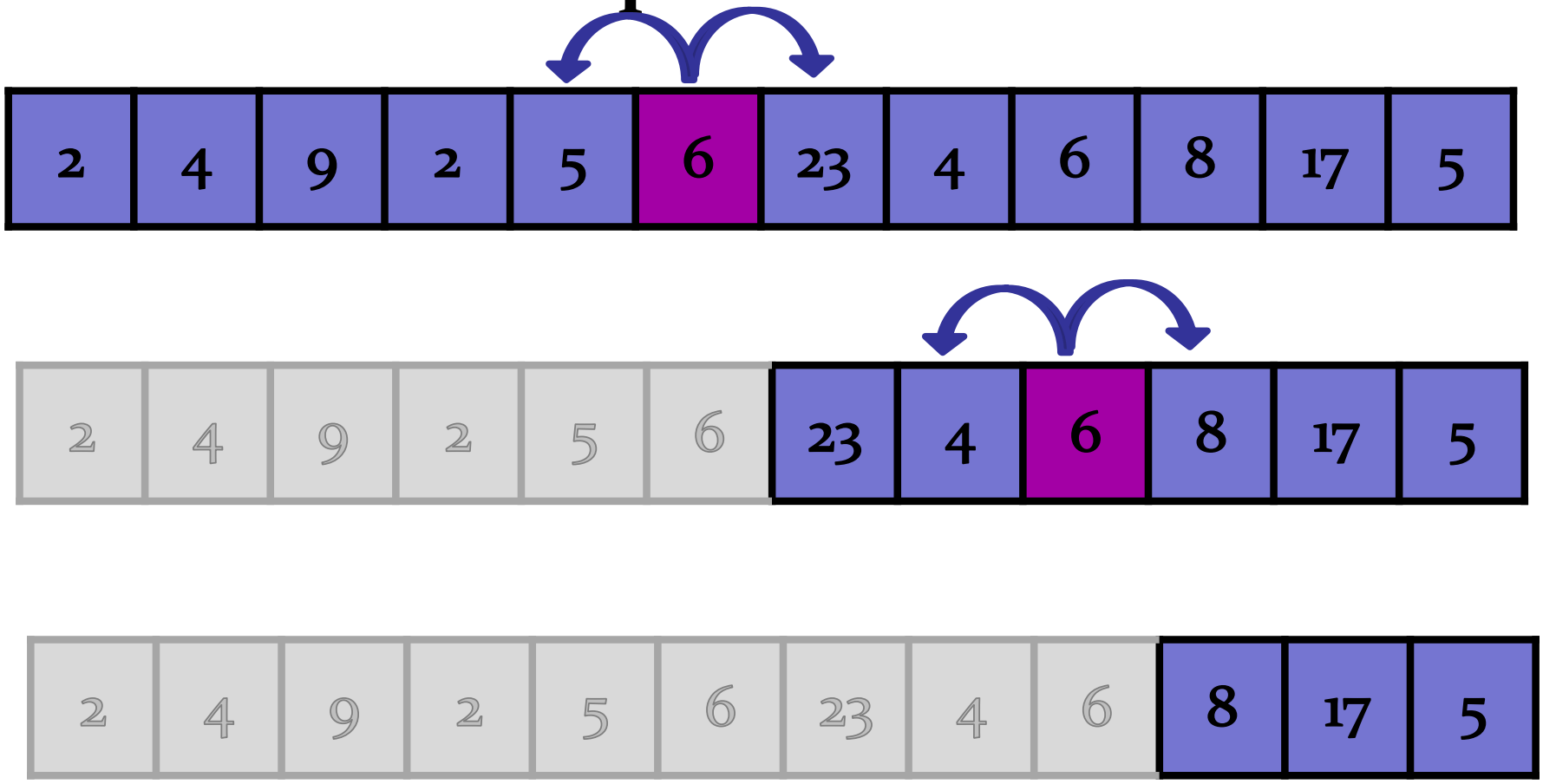
Peak Finding: Algorithm 2

Divide-and-Conquer



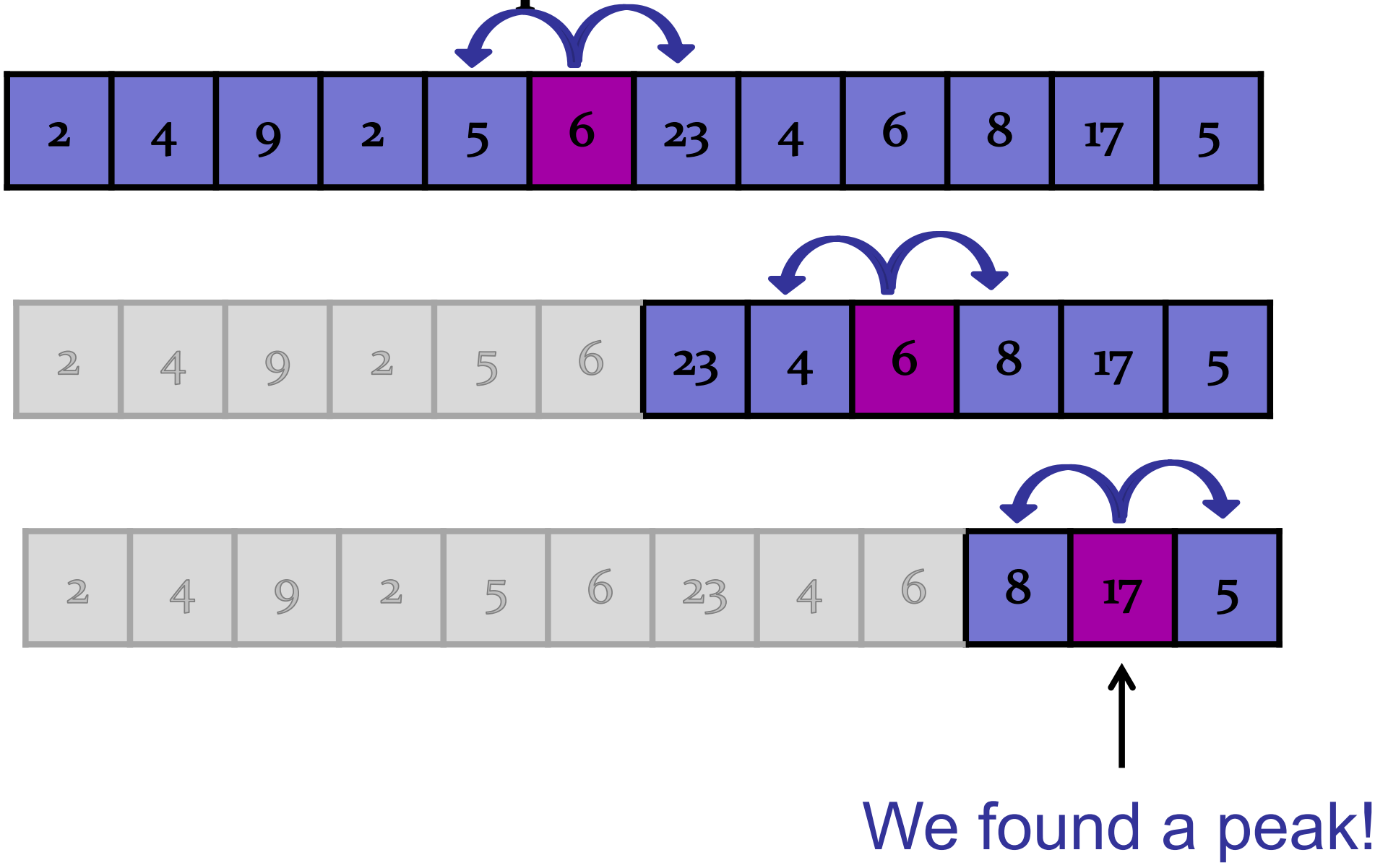
Peak Finding: Algorithm 2

Divide-and-Conquer



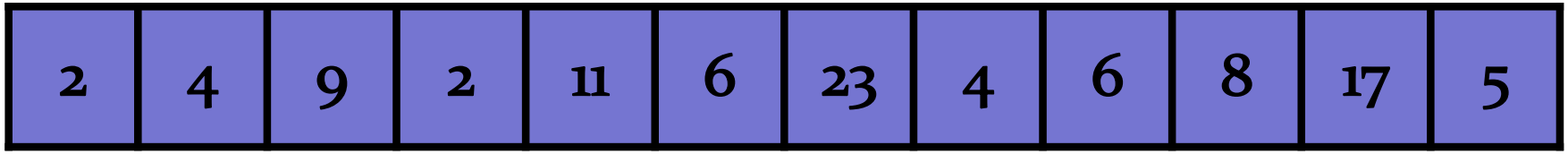
Peak Finding: Algorithm 2

Divide-and-Conquer



Peak Finding: Algorithm 2

Input: Some array A[1..n]



FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

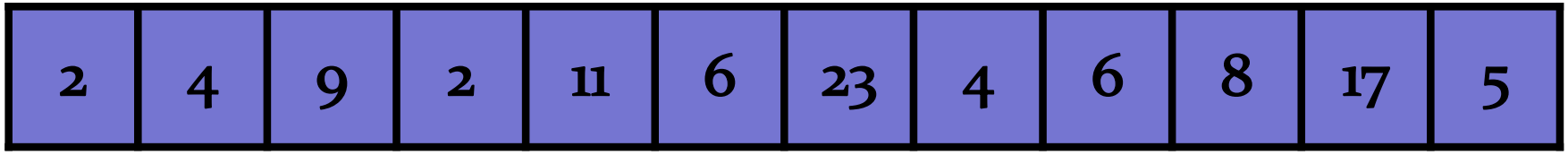
Search for peak in right half.

else if A[n/2–1] > A[n/2] **then**

Search for peak in left half.

Peak Finding: Algorithm 2

Input: Some array A[1..n]

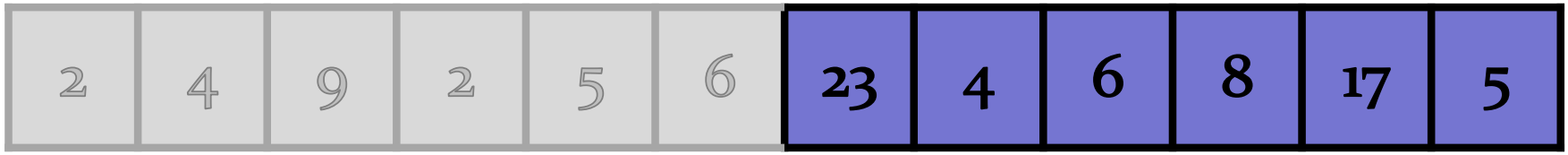



```
FindPeak(A, n)
    if A[n/2] is a peak then return n/2
    else if A[n/2+1] > A[n/2] then
        FindPeak (A[n/2+1..n], n/2)
    else if A[n/2–1] > A[n/2] then
        FindPeak (A[1..n/2-1], n/2)
```

Peak Finding: Algorithm 2

Key property:

- If we recurse in the right half, then there exists a peak in the right half.



Peak Finding: Algorithm 2

Key property:

- If we recurse in the "higher" half, then there exists a peak in the right half.

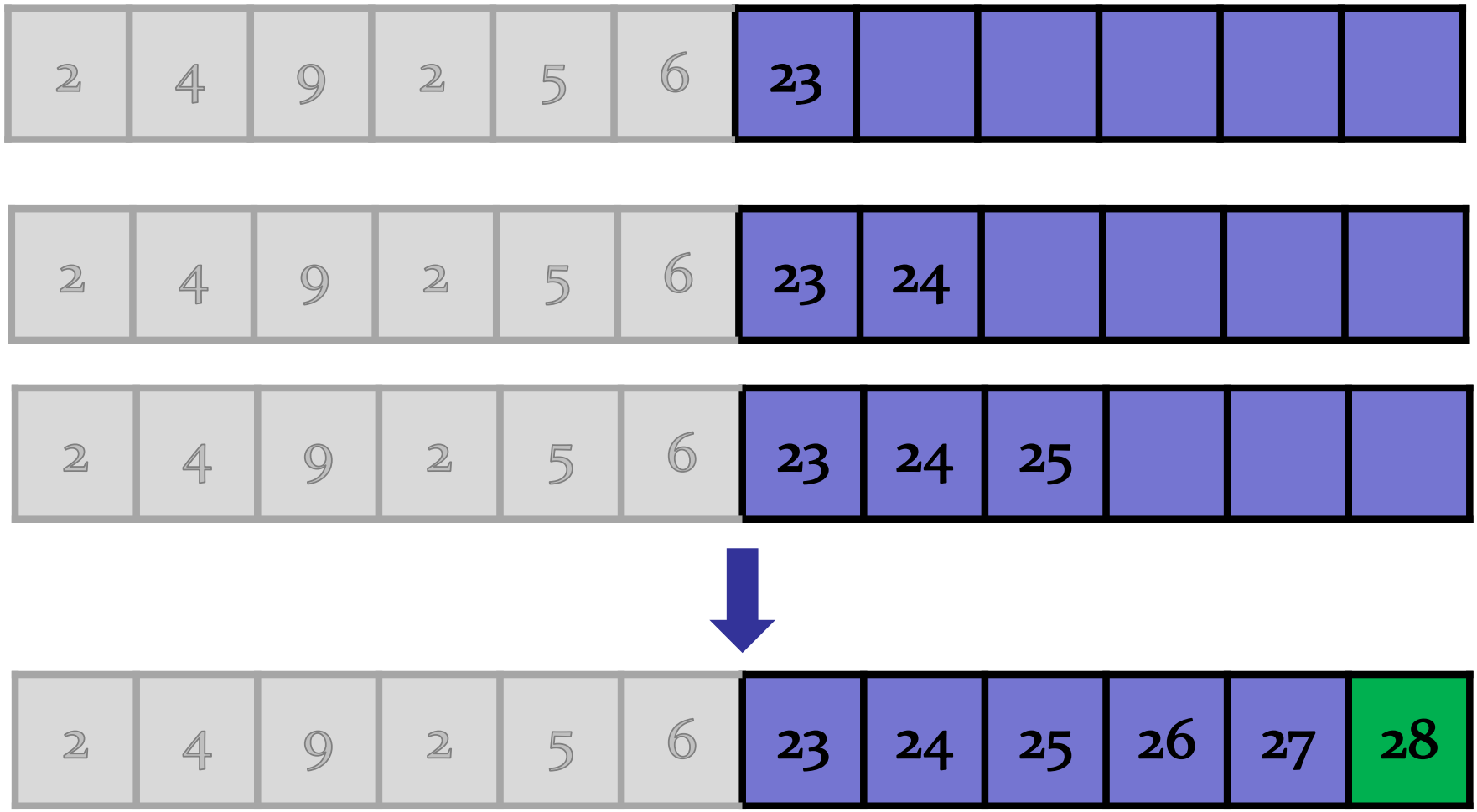
Explanation: Induction!

- Even though there is "no peak" in the right half.
- Given: A[middle] < A[middle + 1]
- Since no peaks, A[middle+1] < A[middle+2]
- Since no peaks, A[middle+2] < A[middle+3]
- …
- Since no peaks, A[n-1] < A[n] ⟵ PEAK!!

Peak Finding: Algorithm 2

Recurse on right half, since 23 > 6.

- Assume no peaks in right half.



Key Invariants:

Correctness:

There exists a peak in the range [begin, end].

Key Invariants:

Correctness:

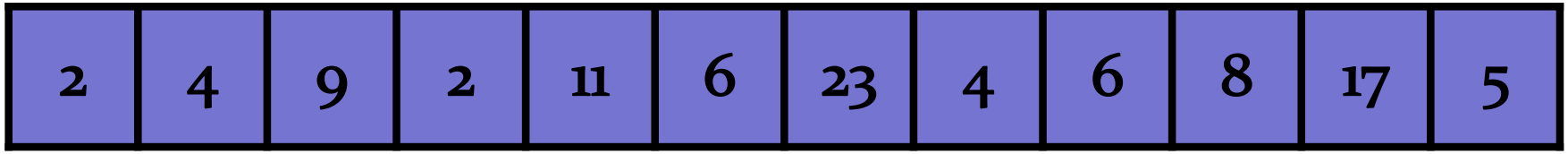
There exists a peak in the range [begin, end].

Every peak in[begin, end] is a valid peak in [1, n].

If the recursive call finds a peak, is it valid?

Peak Finding: Algorithm 2

Running time?



FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

Search for peak in right half.

else if A[n/2–1] > A[n/2] **then**

Search for peak in left half.

Peak Finding: Algorithm 2

Running time:

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

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

Unrolling the recurrence:

$$T(n) = \theta(1) + \theta(1) + \ldots + \theta(1) = O(\log n)$$

Peak Finding: Algorithm 2

Unrolling the recurrence:

Rule:
T(X) = T(X/2) + O(1)

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

$$= T(n/4) + \theta(1) + \theta(1)$$

$$= T(n/8) + \theta(1) + \theta(1) + \theta(1)$$

$$\ldots$$

$$\ldots$$

$$= T(1) + \theta(1) + \ldots + \theta(1) =$$

$$= \theta(1) + \theta(1) + \ldots + \theta(1) =$$

How many times can you divide a number ***n*** in half before you reach 1?

1. $n/4$
2. $\sqrt{n}$
3. ✓ $\log_2(n)$
4. $\arctan(1+\sqrt{5}/2n)$
5. I don't know.

Peak Finding: Algorithm 2

How many times can you divide a number ***n*** in half before you reach 1?

$$\underbrace{2 \times 2 \times \ldots \times 2}_{\log(n)} = 2^{\log(n)} = n$$

Note: I always assume $\log = \log_2$

$$O(\log_2 n) = O(\log n)$$

Peak Finding: Algorithm 2

Running time:

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

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

Unrolling the recurrence:

$$T(n) = \underbrace{\theta(1) + \theta(1) + \ldots + \theta(1)}_{\log(n)} = O(\log n)$$

Summary

Peak finding algorithm:

Key idea: Binary Search

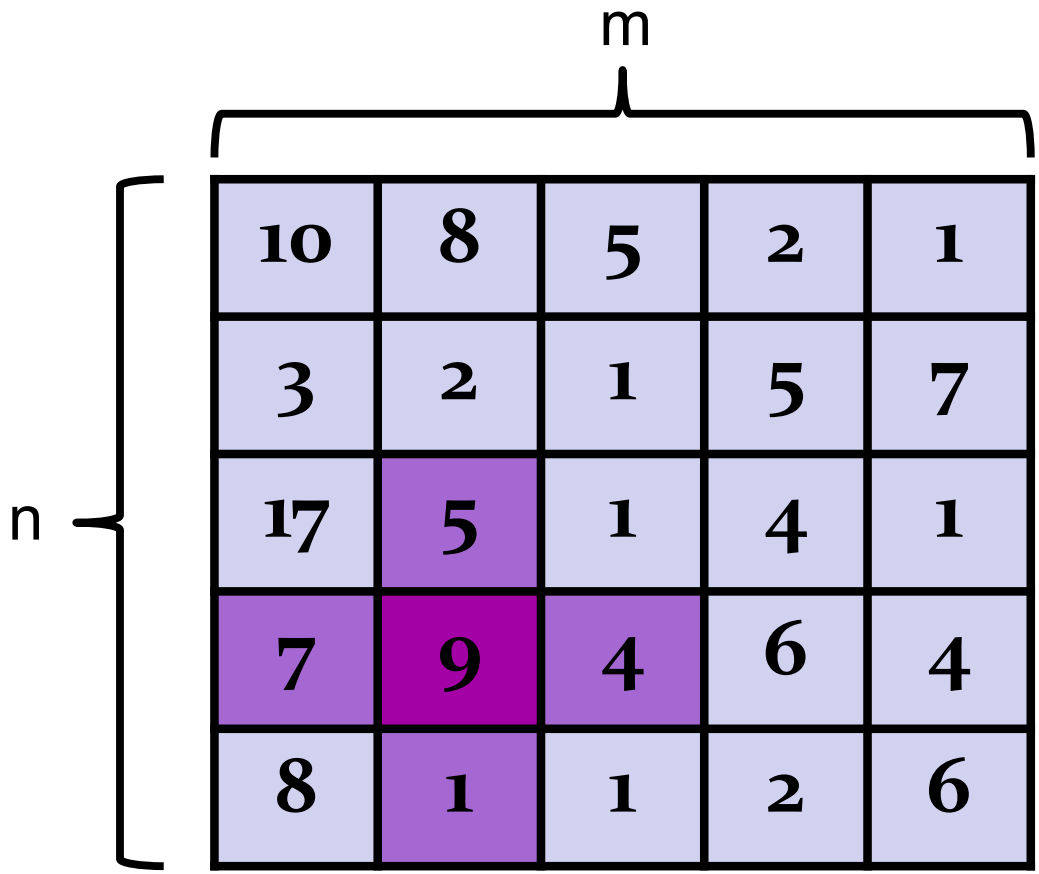
Running time: O(log n)

Onwards...

The 2nd dimension!

Peak Finding 2D (the sequel)

Given: 2D array A[1..n, 1..m]



Output: a peak that is not smaller than the (at most) 4 neighbors.

2D: Algorithm 1

Step 1: Find global max for each column

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

7 9 5 5 ← Find 1D peak.

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

Algorithm 1-2D

Step 1: Find global max for each column.

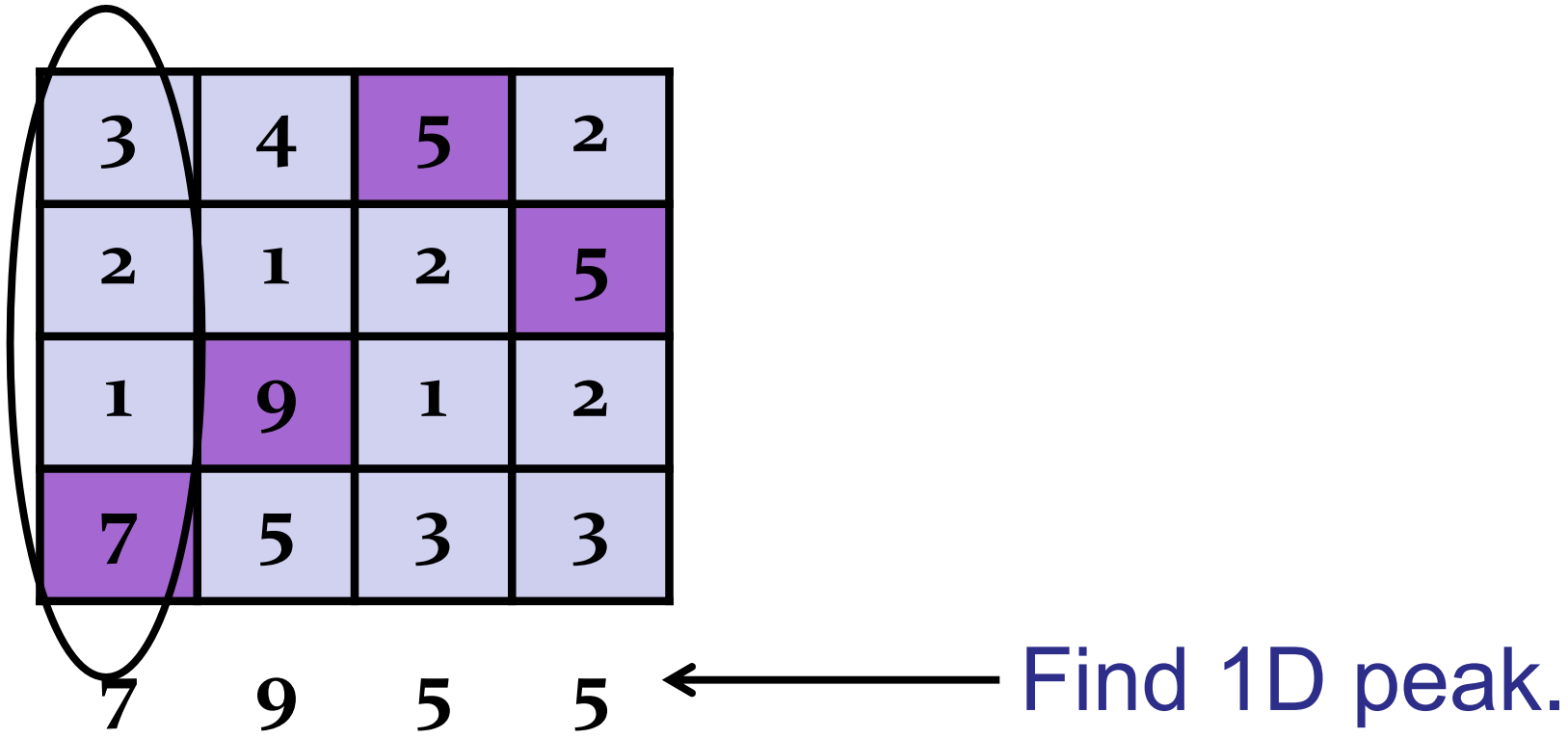
Step 2: Find peak in the max array.

Is this algorithm correct?

1. Yes
2. No

2D: Algorithm 1

Step 1: Find global max for each column



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

Running time: O(mn + log(m))

2D: Algorithm 2

Step 1: Find a (local) peak for each column

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

7 9 5 5 ← Find 1D peak.

Step 2: Find peak in the array of peaks.

Algorithm 2-2D

Step 1: Find 1D-peak for each column.

Step 2: Find peak in the max array.

Is this algorithm correct?

1. Yes
2. No

2D: Algorithm 2 (Counter Example)

Step 1: Find a (local) peak 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.

Step 2: Find peak in the array of peaks.

2D: Algorithm 1

~~Step 1: Find global max for each column~~

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

7 9 5 5 ← Find 1D peak.

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

Running time: O(mn + log(m))

2D: Algorithm 2

~~Step 1: Find a global max for each column~~

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

? ? ? ? ← Find 1D peak.

Step 2: Find peak in the array of peaks by lazy evaluation.

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 1D Peak:

Step 1: Check middle element.

Step 2: Recurse left/right half.

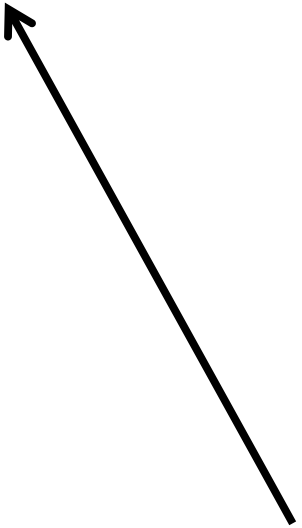
7	10	12	20	7	9	4	3	1	18	5	17	4
19	11	7	4	6	8	10	3	5	6	8	14	8
6	9	14	4	7	9	3	12	9	8	3	10	6

? ? ? ? ? 9 10 12 ? ? ? ? ?

Find 1D Peak:

Step 1: Check middle element.

Step 2: Recurse left/right half.



Column
Max Array

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 ?

Find 1D Peak:

Step 1: Check middle element.

Step 2: Recurse left/right half.

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

? ? ? ? ? 8 10 12 ? 18 8 14 8

Find 1D Peak:

Step 1: Check middle element.

Step 2: Recurse left/right half.

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

? ? ? ? ? 8 10 12 ? 18 8 14 8

How many columns do we need to examine?

1. O(m)
2. O(√m)
3. O(log m)
4. O(1)

2D: Algorithm 2

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 max
- Total: **O(n log m)**

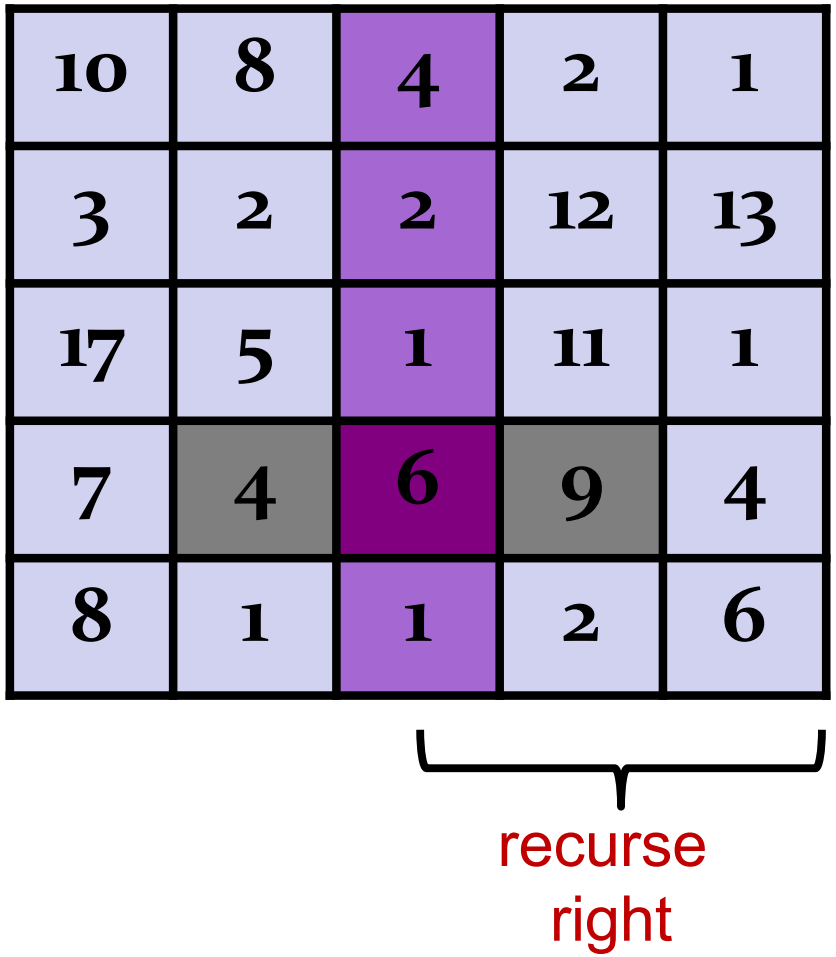
(Much better than O(nm) of before.)

2D Algorithm 3

2D Algorithm 3

Divide-and-Conquer

1. Find MAX element of middle column.
2. If found a peak, DONE.
3. Else:
   - If left neighbor is larger, then recurse on left half.
   - If right neighbor is larger, then recurse on right half.



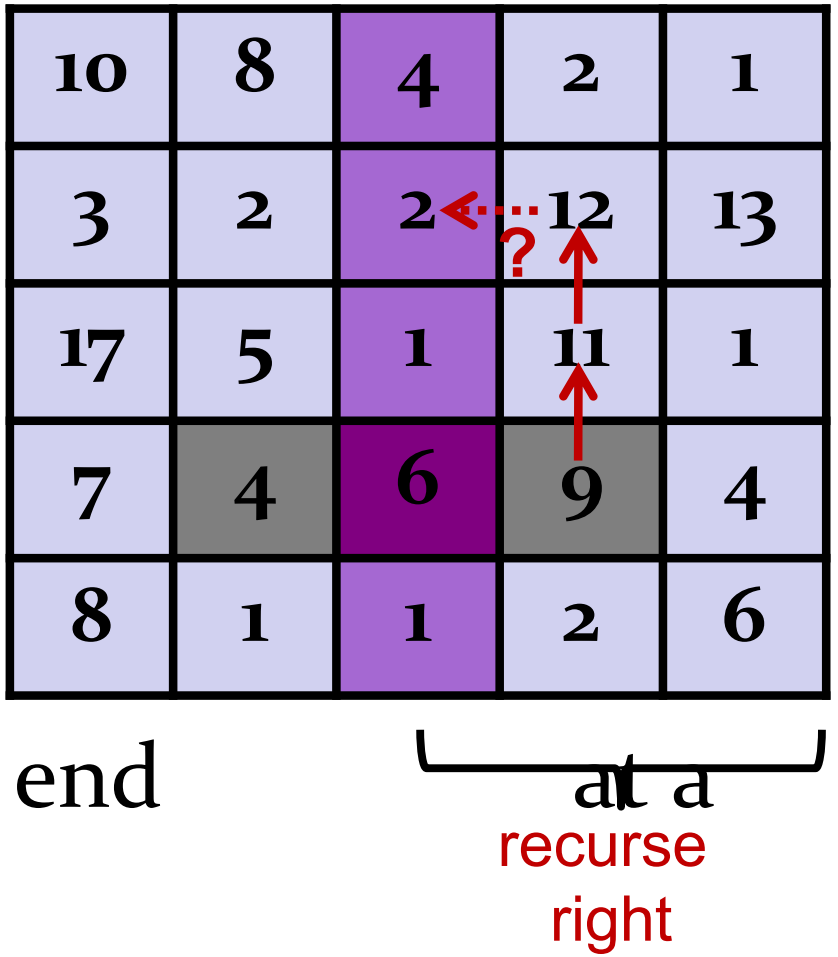
2D Algorithm 3

Correctness

1. Assume no peak on right half.
2. Then, there is some increasing path:

   $9 \rightarrow 11 \rightarrow 12 \rightarrow \ldots$

3. Eventually, the path must end at a max.
4. If there is no max in the right half, then it must cross to the left half… Impossible!



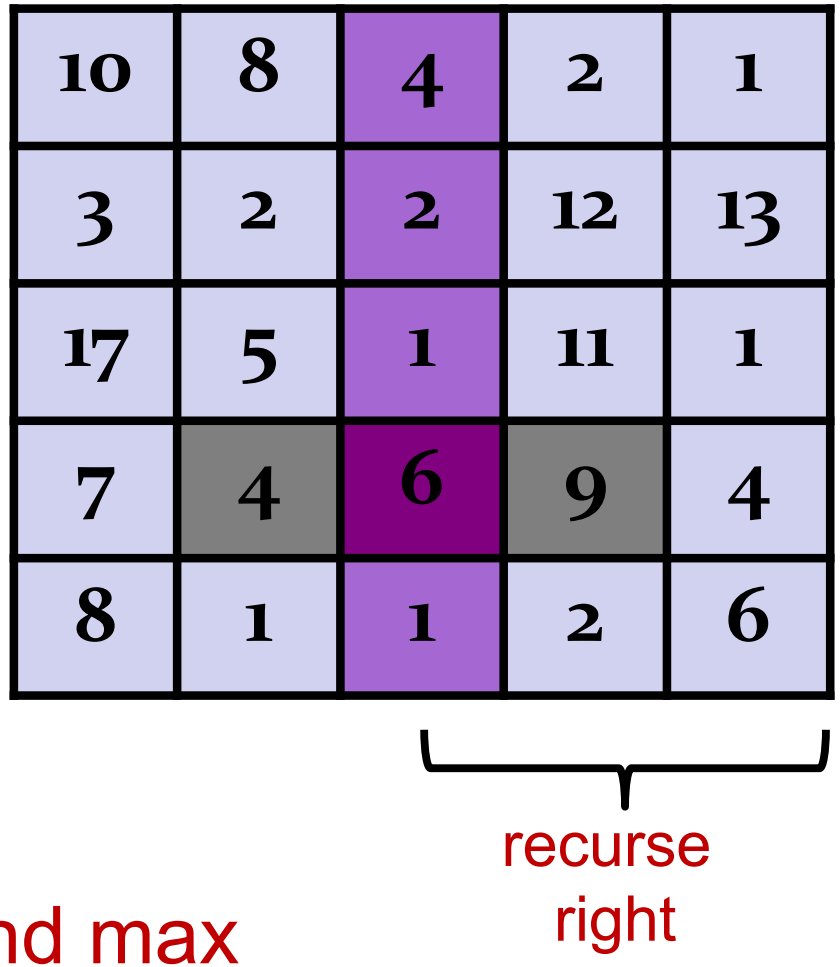
2D Algorithm 3

Divide-and-Conquer

$$T(n,m) = T(n,m/2) + O(n)$$

Recurse *once* on array of size [n, m/2]

Do n work to find max element in column.



Recurrence Analysis

$$
\begin{aligned}
T(n, m) &= T(n, m/2) + n \\
&= T(n, m/4) + n + n \\
&= T(n, m/8) + n + n + n \\
&= T(n, m/16) + n + n + n + n \\
&= \ldots
\end{aligned}
$$

Recurrence Analysis

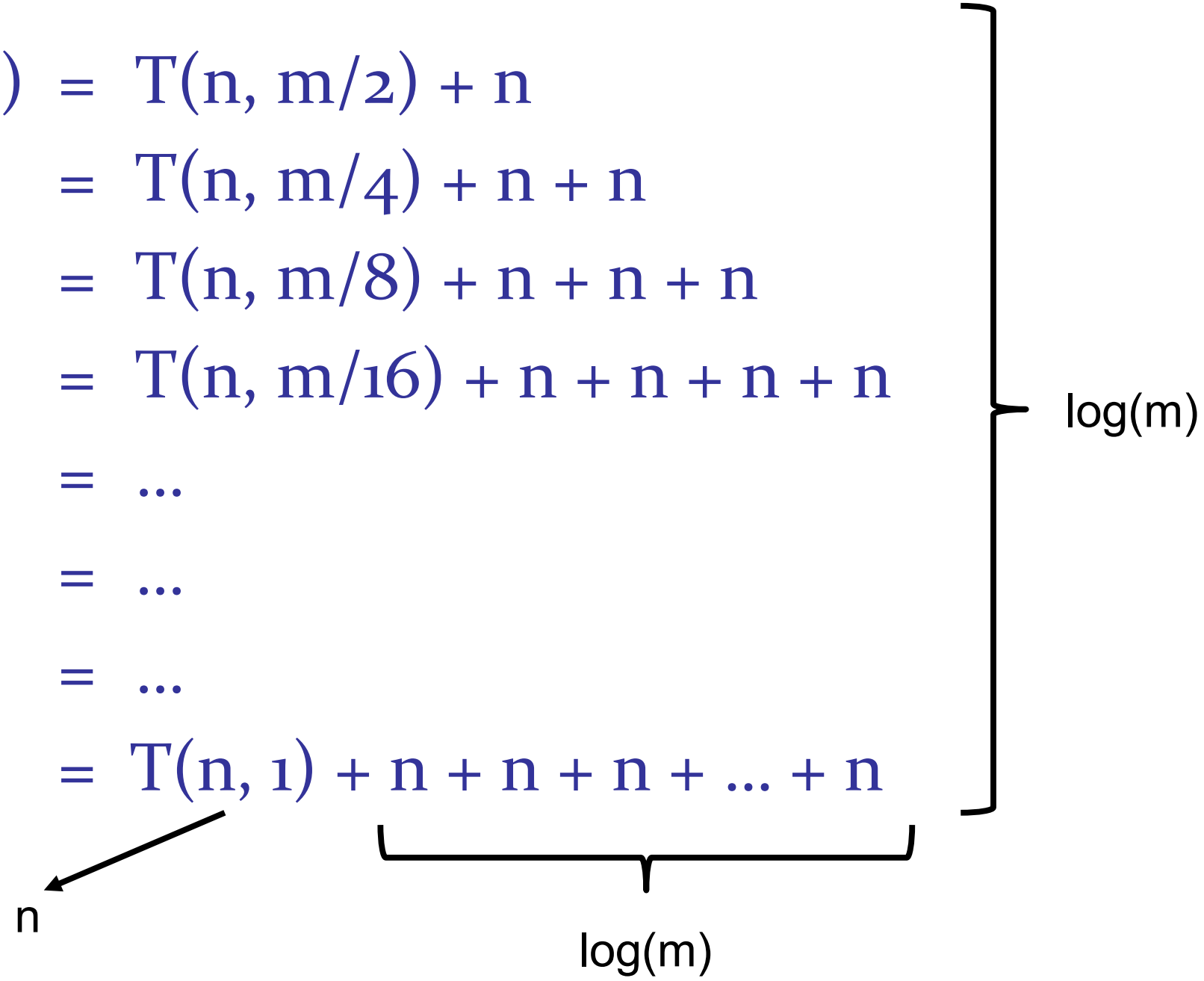
$$T(n, m) = T(n, m/2) + n$$

T(n,m) = ??

1. O(log m)
2. O(nm)
3. O(n log m)
4. O(m log n)

Recurrence Analysis

$$
\begin{aligned}
T(n, m) &= T(n, m/2) + n \\
&= T(n, m/4) + n + n \\
&= T(n, m/8) + n + n + n \\
&= T(n, m/16) + n + n + n + n \\
&= \ldots \\
&= \ldots \\
&= \ldots \\
&= T(n, 1) + n + n + n + \ldots + n
\end{aligned}
$$

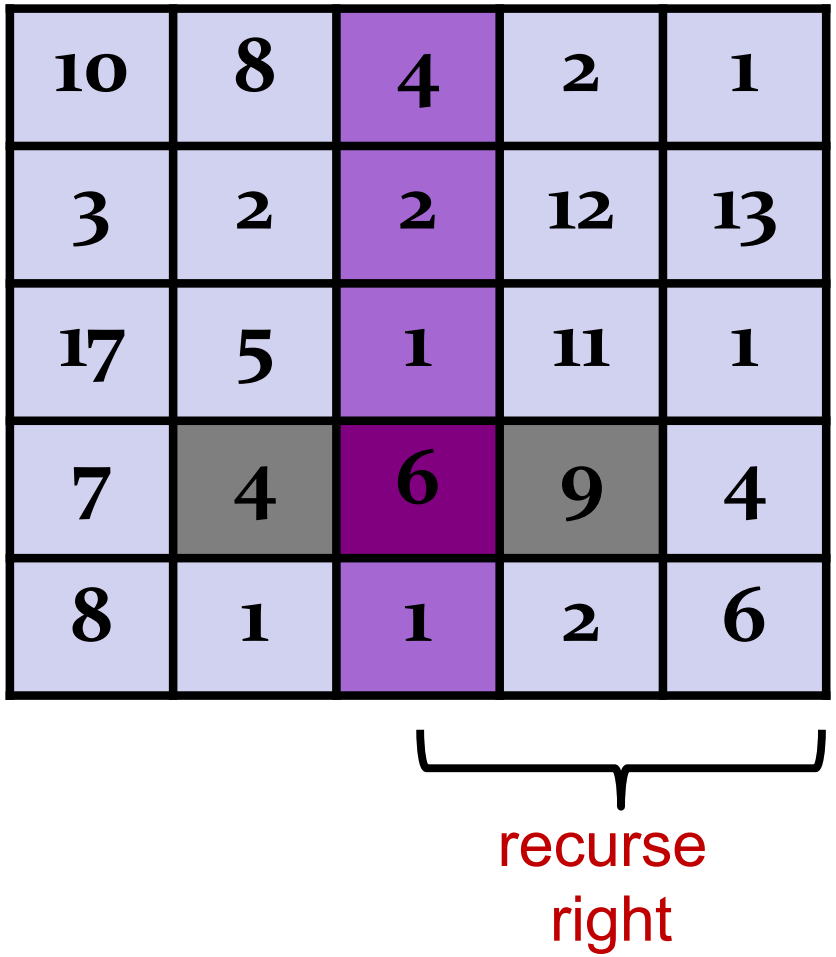


2D Algorithm 3

Divide-and-Conquer

1. Find MAX element of middle column.
2. If found a peak, DONE.
3. Else:
   - If left neighbor is larger, then recurse on left half.
   - If right neighbor is larger, then recurse on right half.

T(n) = O(n log m)



2D Algorithm 4

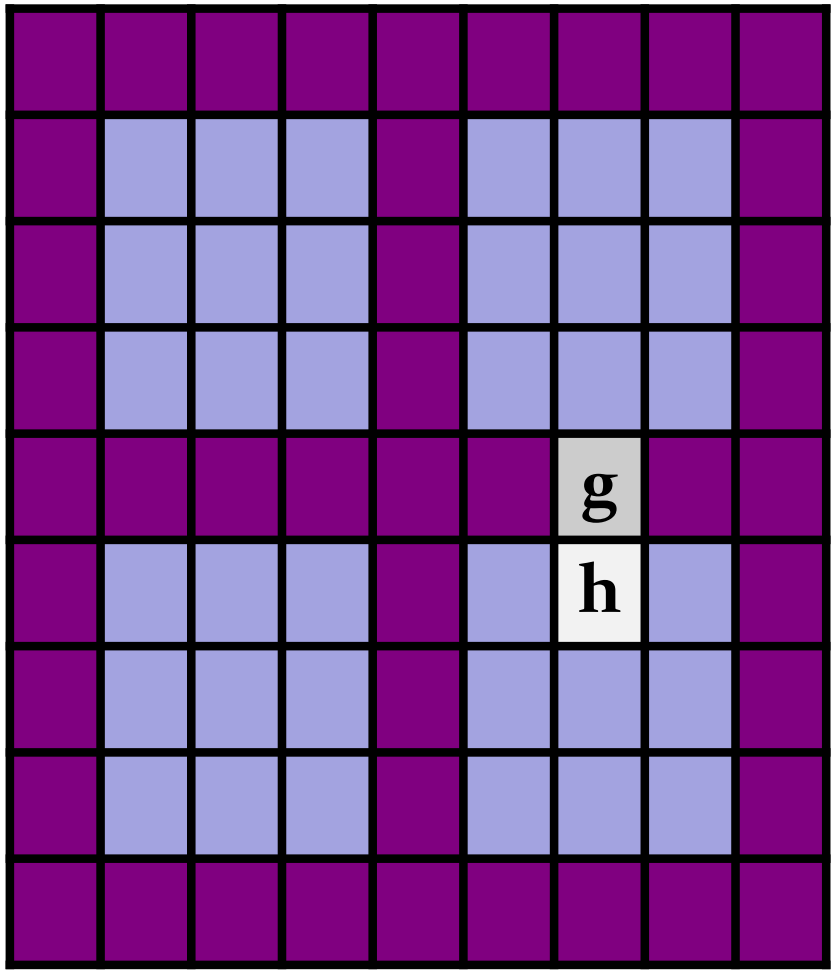
Can we do better than O(n log m)?

2D Algorithm 4

Divide-and-Conquer

1. Find MAX element on border + cross.
2. If found a peak, DONE.
3. Else:

   Recurse on quadrant containing element bigger than MAX.



Example: MAX = g

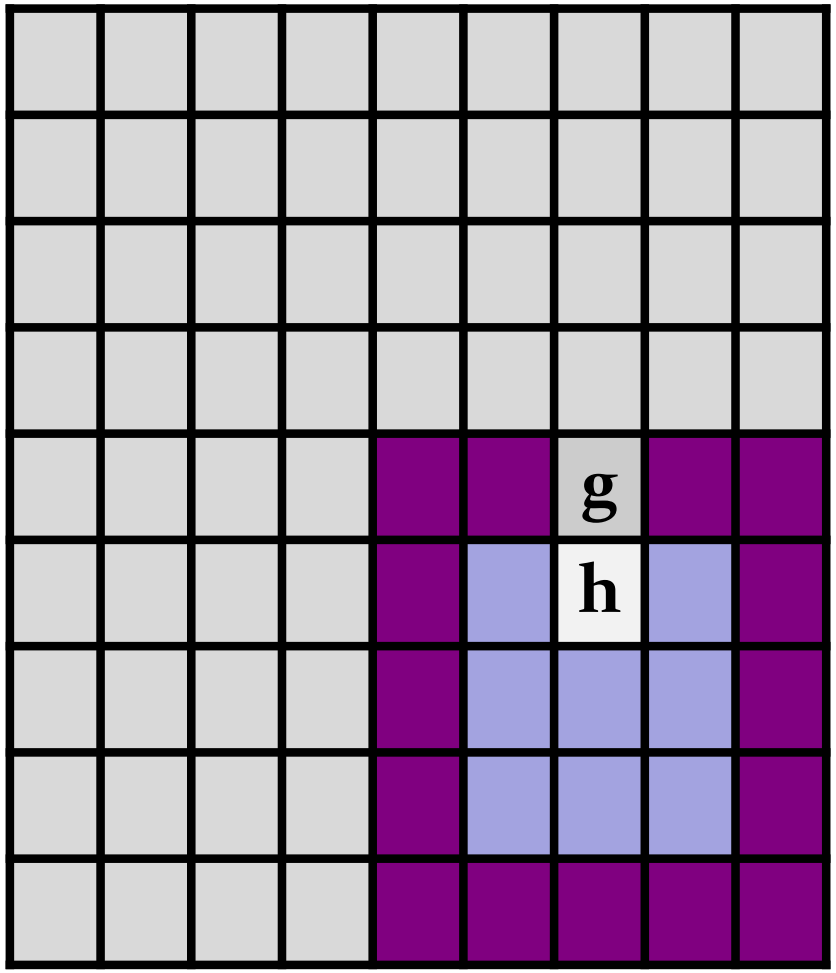
h > g

2D Algorithm 4

Divide-and-Conquer

1. Find MAX element on border + cross.
2. If found a peak, DONE.
3. Else:

   Recurse on quadrant containing element bigger than MAX.



Example: MAX = g

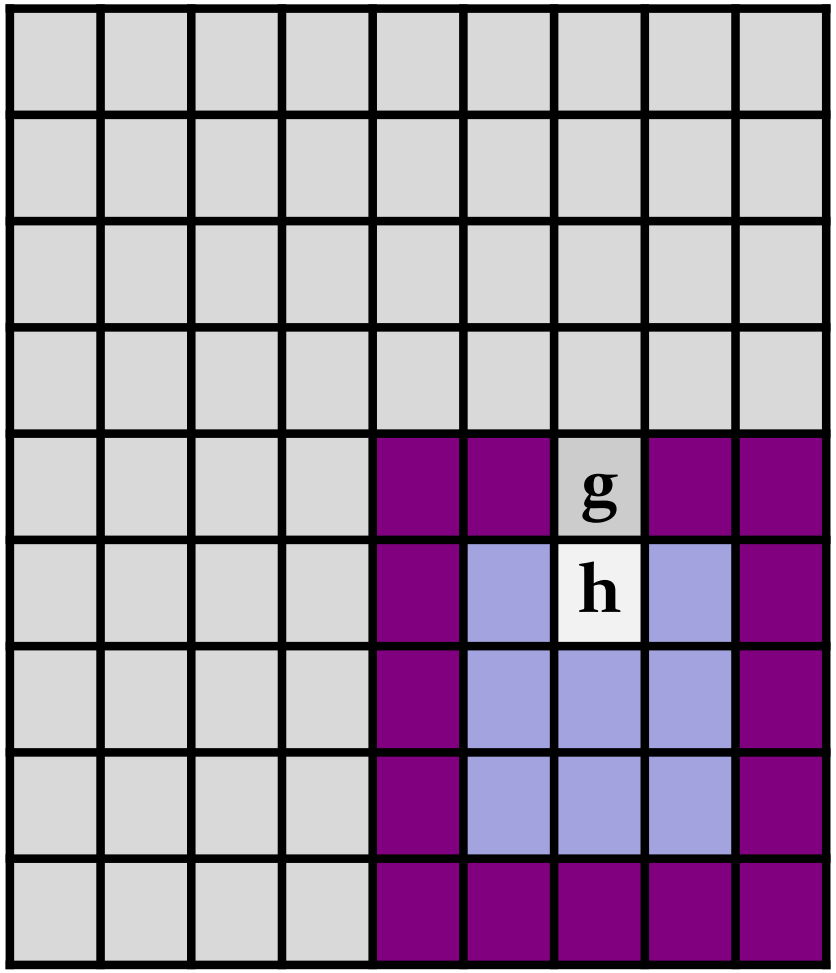
h > g

2D Algorithm 4

Correctness

1. The quadrant contains a peak.

   Proof: as before.



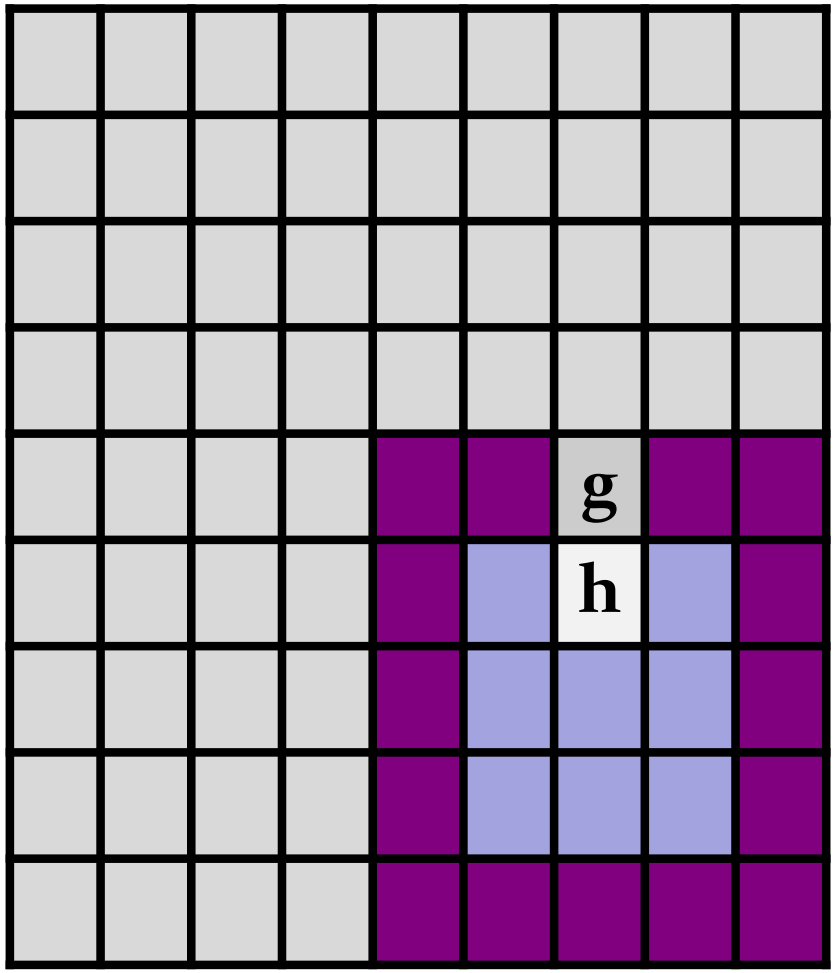
2D Algorithm 4

Correctness

1. The quadrant contains a peak.

   Proof: as before.

2. Every peak in the quadrant is NOT a peak in the matrix.



2D Algorithm 4

Correctness

1. The quadrant contains a peak.

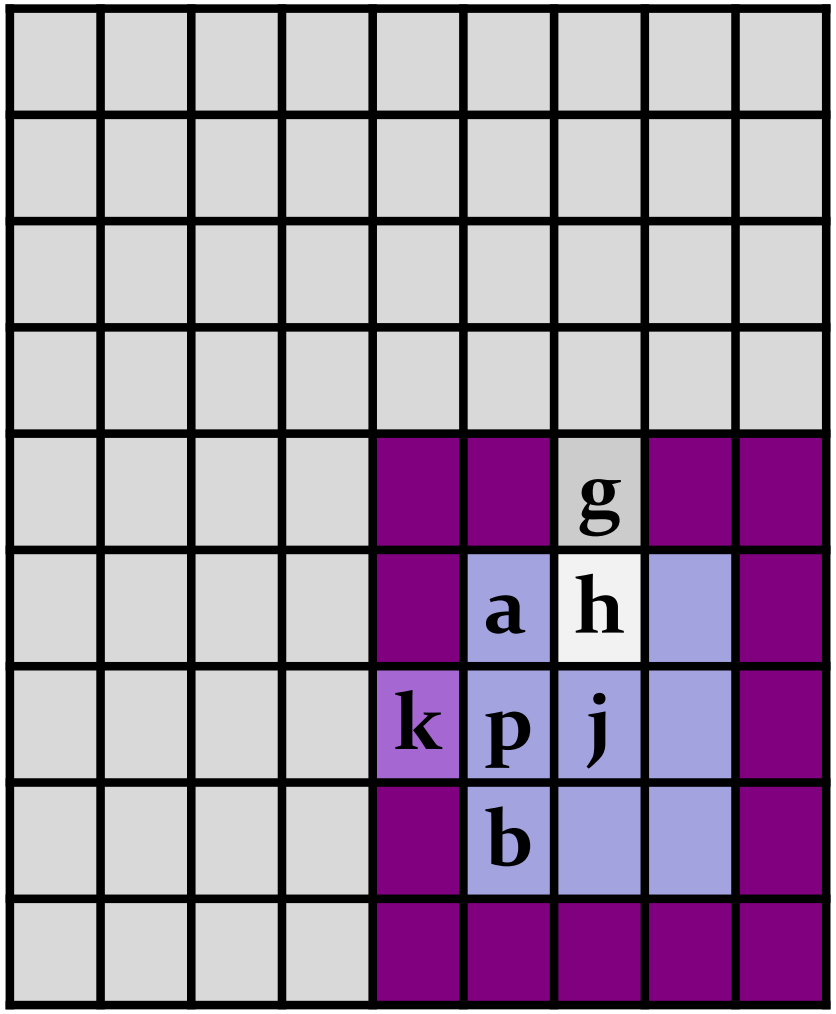
   Proof: as before.

2. Every peak in the quadrant is NOT a peak in the matrix.

   Example: k > p > j

   p > a

   p > b



2D Algorithm 4

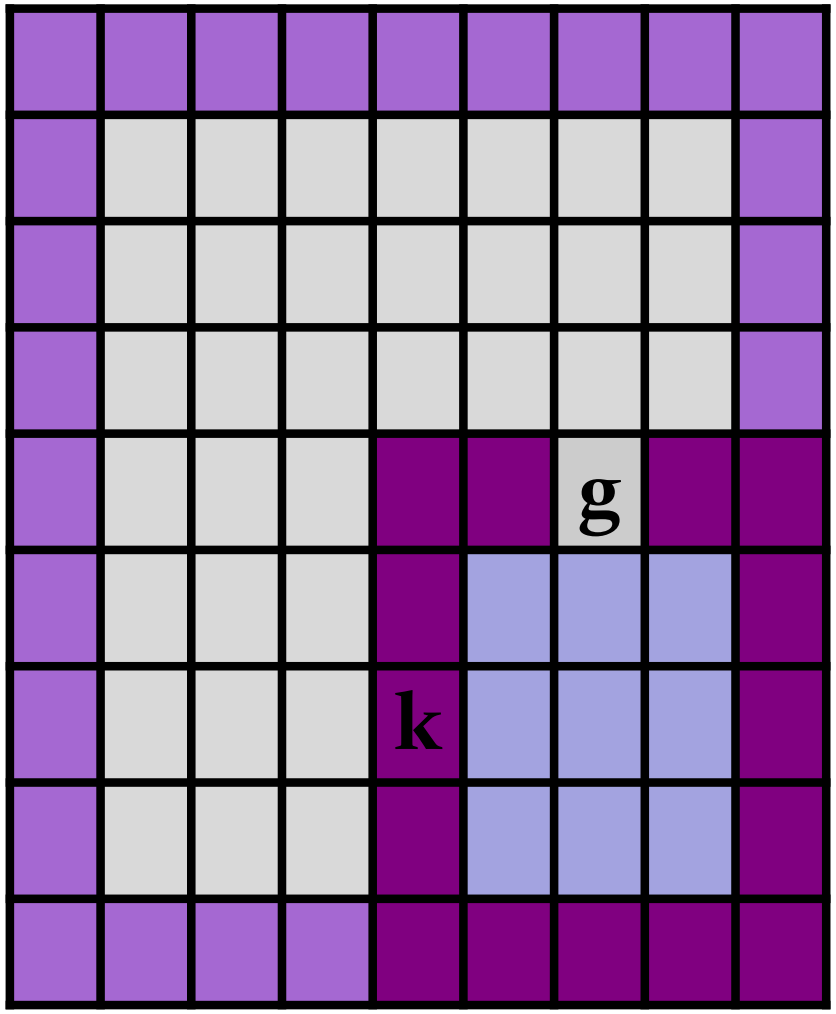
Correctness

Key property:

Find a peak at least as large as every element on the boundary.

Proof:

If recursing finds an element at least as large as g, and g is as big as the biggest element on the boundary, then the peak is as large as every element on the boundary.



2D Algorithm 4

Divide-and-Conquer

$$T(n,m) = T(n/2, m/2) + O(n + m)$$

Recurse *once* on array of size [n/2, m/2]

Do 6(n+m) work to find max element.

Recurrence Analysis

$$
\begin{aligned}
T(n, m) &= T(n/2, m/2) + c(n+m) \\
&= T(n/4, m/4) + c(n/2 + m/2) + c(n + m) \\
&= T(n/8, m/8) + c(n/4 + m/4) + \ldots \\
&= \ldots
\end{aligned}
$$

Recurrence Analysis

$$T(n, m) = T(n/2, m/2) + cn + cm$$

T(n, m) = ??

1. O(nm)
2. O(n log m)
3. O(m log n)
4. O(n+m)

Recurrence Analysis

$$
\begin{aligned}
T(n, m) &= T(n/2, m/2) + c(n+m) \\
&= T(n/4, m/4) + c(n/2 + m/2 + n + m) \\
&= T(n/8, m/8) + c(n/4 + m/4 + \ldots \\
&= \ldots \\
\\
&= cn(1 + \tfrac{1}{2} + \tfrac{1}{4} + \ldots) + \\
&\quad\ cm(1+ \tfrac{1}{2} + \tfrac{1}{4} + \ldots) \\
&< 2cn + 2cm \\
&= O(n +m)
\end{aligned}
$$

Summary

1D Peak Finding

- Divide-and-Conquer
- O(log n) time

2D Peak Finding

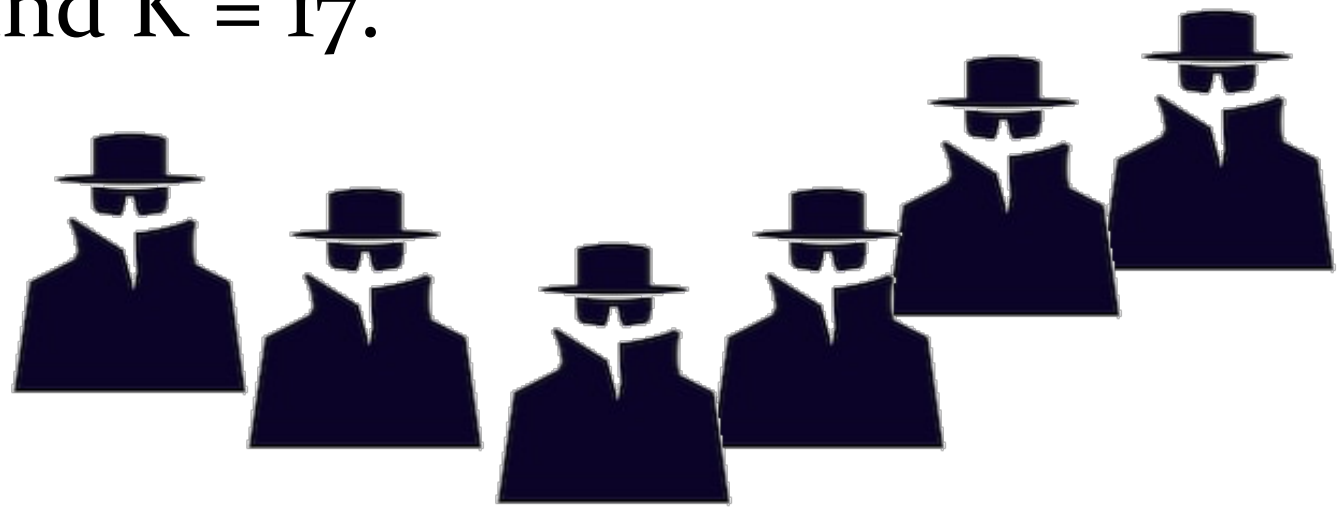
- Simple algorithms: O(n log m)
- Careful Divide-and-Conquer: O(n + m)

Puzzle of the Week

There are N students in CS2040S and K of them are spies. Your job is to identify all the spies.

Let's assume:

N = 1,024 and K = 17.



Puzzle of the Week


To catch a spy:

You can send some students on a mission.

If all K spies are on the mission, they will meet.

You learn if the meeting occurred or not.


You learn nothing else.

Puzzle of the Week

N = 1024
K = 17

Prove:

You need at least 122 missions to identify all the spies.

Find:

The best strategy you can to catch all the spies. (You might write a program!)