# CS2040S
# Data Structures and Algorithms

## Welcome!

Puzzle of the day:

10 coins are placed in front of you. You are blindfolded, and cannot feel whether the coins are heads-side up or tails-side up. You may only move the coins around, or flip them. You are promised 5 of the coins are head-side up. 5 of the coins are tail-side up.

How do you make 2 piles of coins where the number of heads-up coins in the first pile is the same as the second

# How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

# Problem Set 2

Released on Monday

Due Sunday night. (Click “finalize” when done!)

FAQ:

- If you have questions, ask on the Coursemology forum.
- Think carefully about the different possible inputs.
- Think carefully about the strange corner cases.
- Private test cases are for the purpose of evaluation (i.e., we will not release them or tell you what they are); they may include hints. It may be the same of them are testing very hard cases.

# Problem Set Policies

1. No resubmission.
    - Tutors only have time to grade once!
2. Almost no unsubmission.
    - Please do not submit until you are ready to have it graded.
    - In extreme cases, can ask tutor for unsubmission, with very good reason.
    - If tutor deems that you have entirely misunderstood the question, they may unsubmit for you.
3. As much feedback as you want.
    - Tutors will help you to understand what you got wrong, look at any fixes that you make, and help you to learn.
4. Tutors can grant rare *short* extensions.
    - Ask your tutor if you need an extension for a very good reason.

# How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

# Last time…

## Binary Search

- Simple, ubiquitous algorithm.
- Surprisingly easy to add bugs.
- Some ideas for avoiding bugs:
  - Problem specification
  - Preconditions
  - Postconditions
  - Invariants / loop invariants
  - Validate (when feasible)

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
```

# 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

# How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

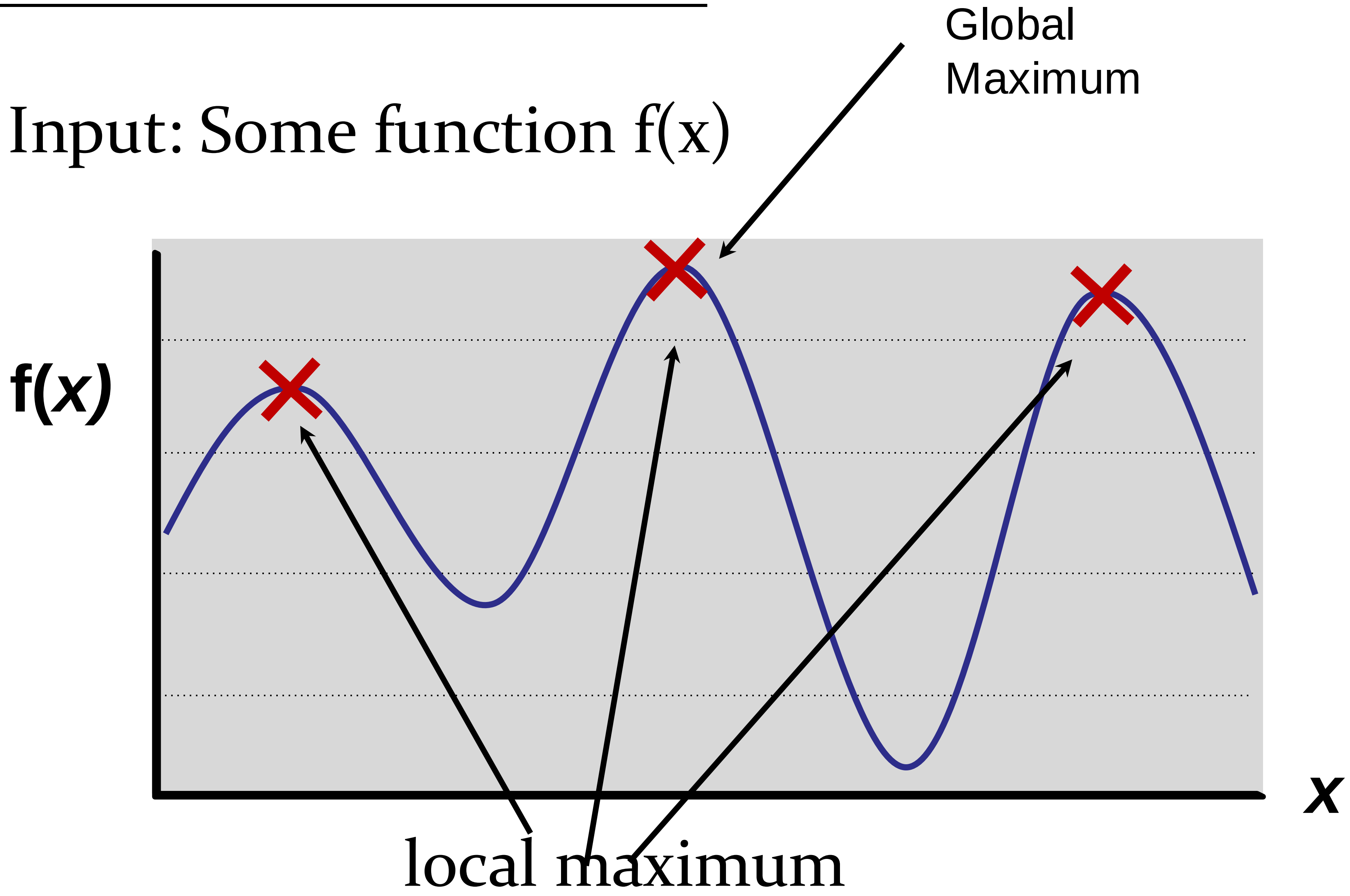
Searching

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[0..n-1]

Output: global maximum element in A

# How long to find a global maximum?

Input: Arbitrary array $A[0..n-1]$

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[0..n-1]`

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

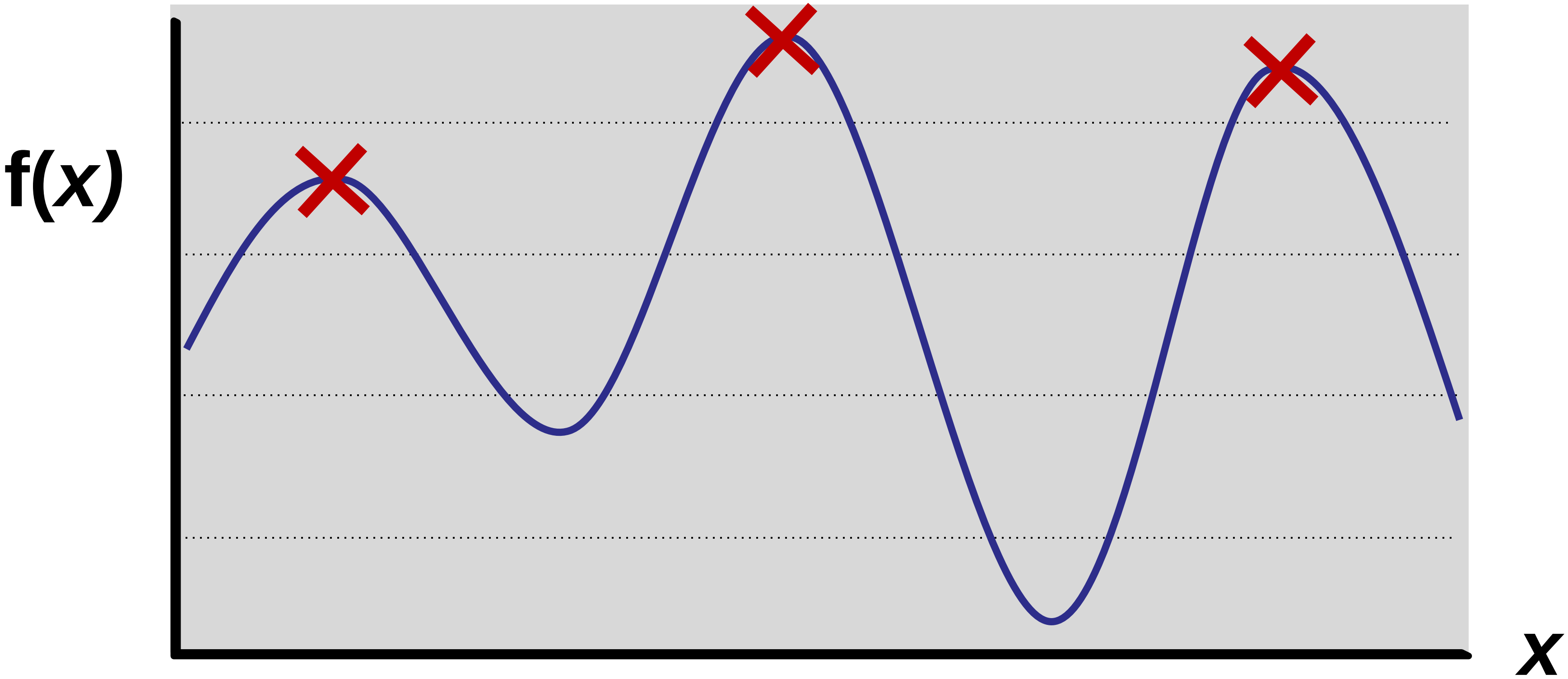
```
FindMax(A,n)
   max = A[1]
   for i = 1 to n-1 do:
       if (A[i]> max) then max=A[i]
```

Time Complexity: $O(n)$

Too slow!

# Peak (Local Maximum) Finding

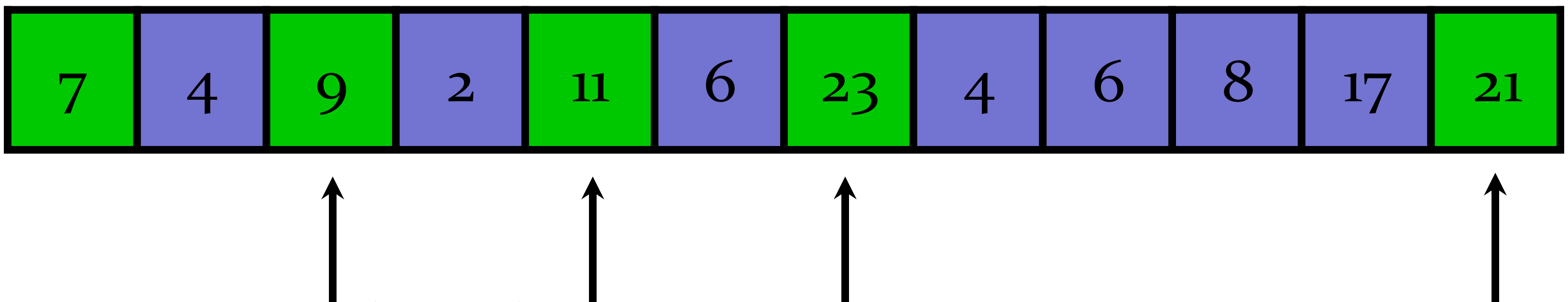
Input: Some function f(x)




Output: A local maximum

# Peak Finding

Input: Some ~~function~~ array A[0..n-1]




Output: a local maximum in A

A[i-1] ≤ A[i] **and** A[i+1] ≤ A[i]

Assume that

A[-1] = A[n] = -MAX_INT

# Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

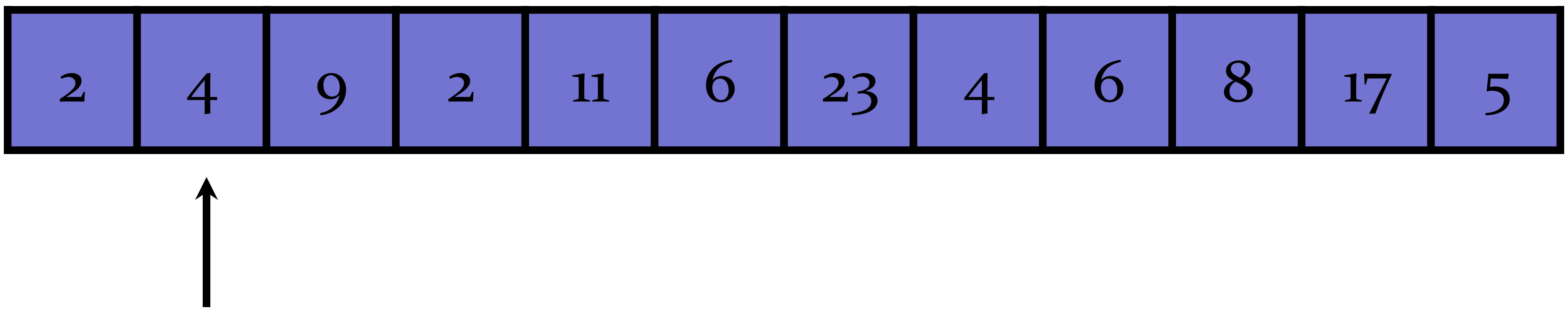
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

FindPeak

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

# Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

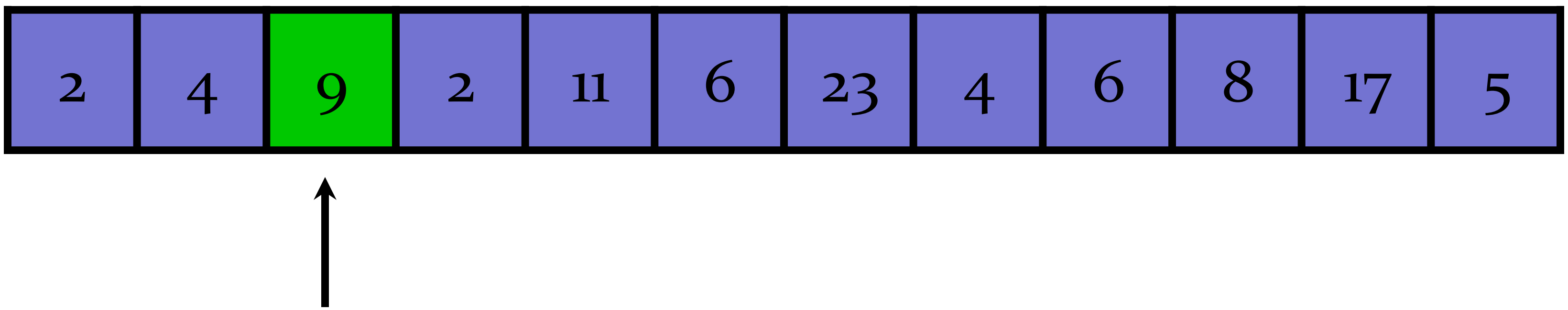



FindPeak

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

# Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`




FindPeak

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

# Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

Running time: n

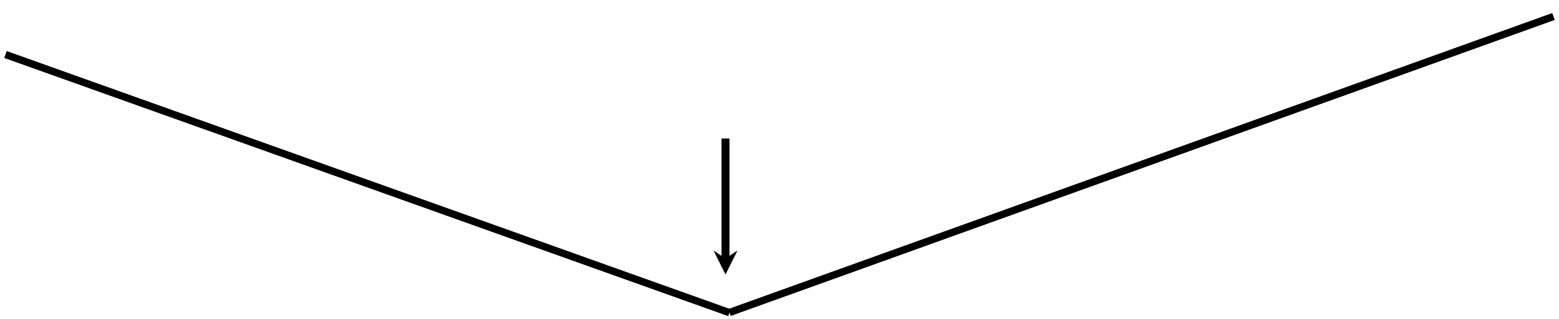
Simple improvement?

# Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 4 | 5 | 6 | 9 | 11 | 13 | 15 | 17 | 25 |

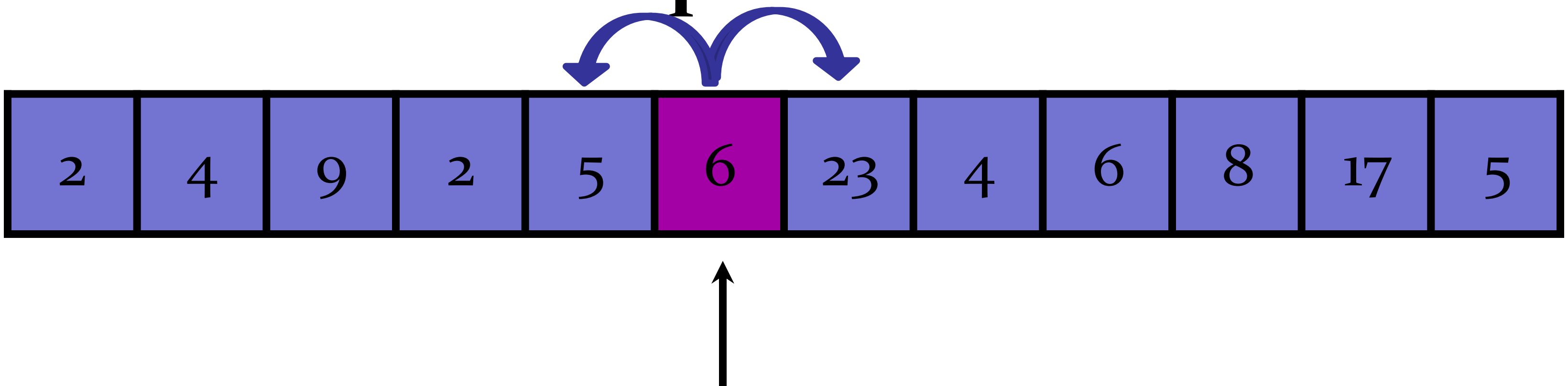
**Start in the middle!**



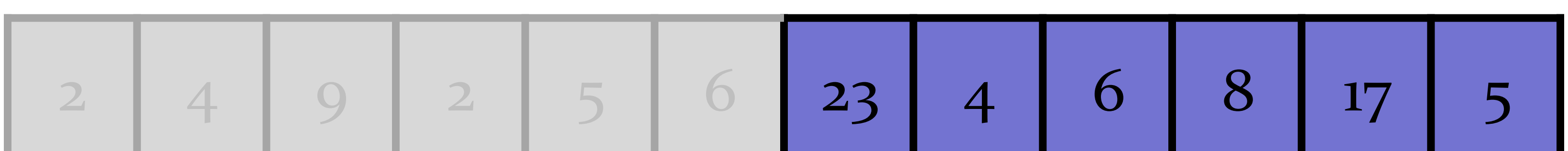
Worst-case: n/2

# Peak Finding: Algorithm 2

**Reduce-and-Conquer**

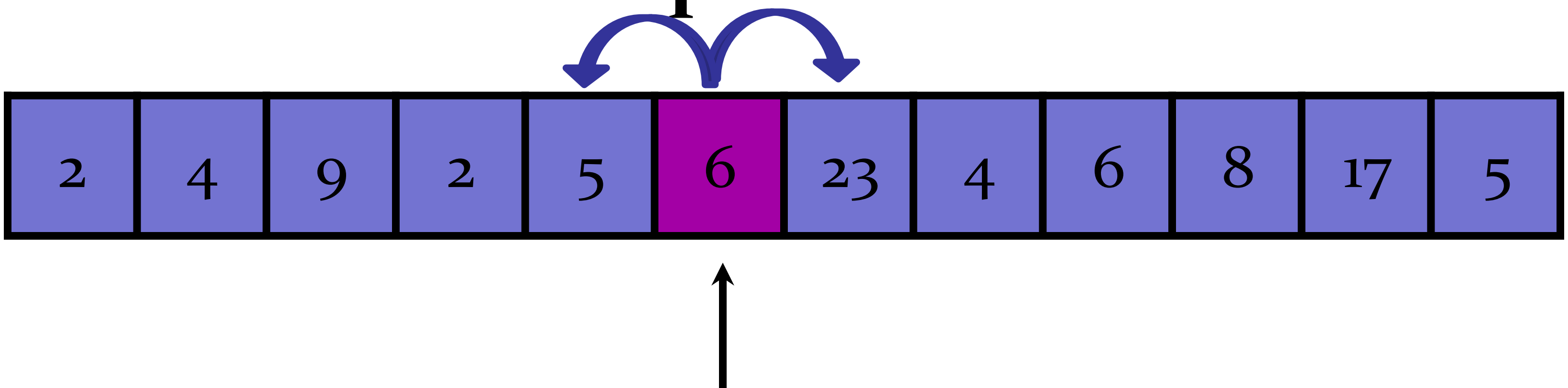



Think: Which side should we recurse on?



# Peak Finding: Algorithm 2

**Reduce-and-Conquer**

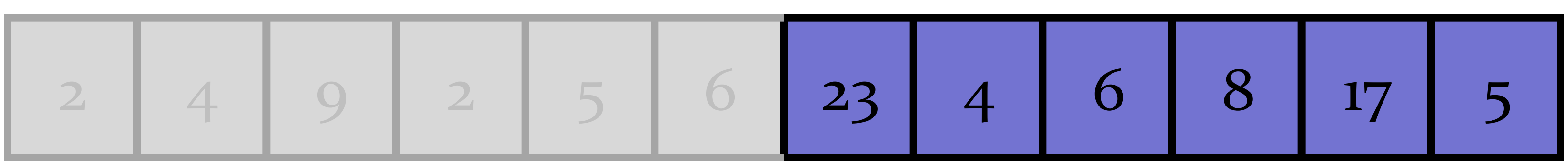



Start in the middle

**5 < 6?**

**6 > 23?** ← **NOT PEAK**

Recurse on which side?



# Peak Finding: Algorithm 2

**Reduce-and-Conquer**

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

Start in the middle

**5 < 6?**

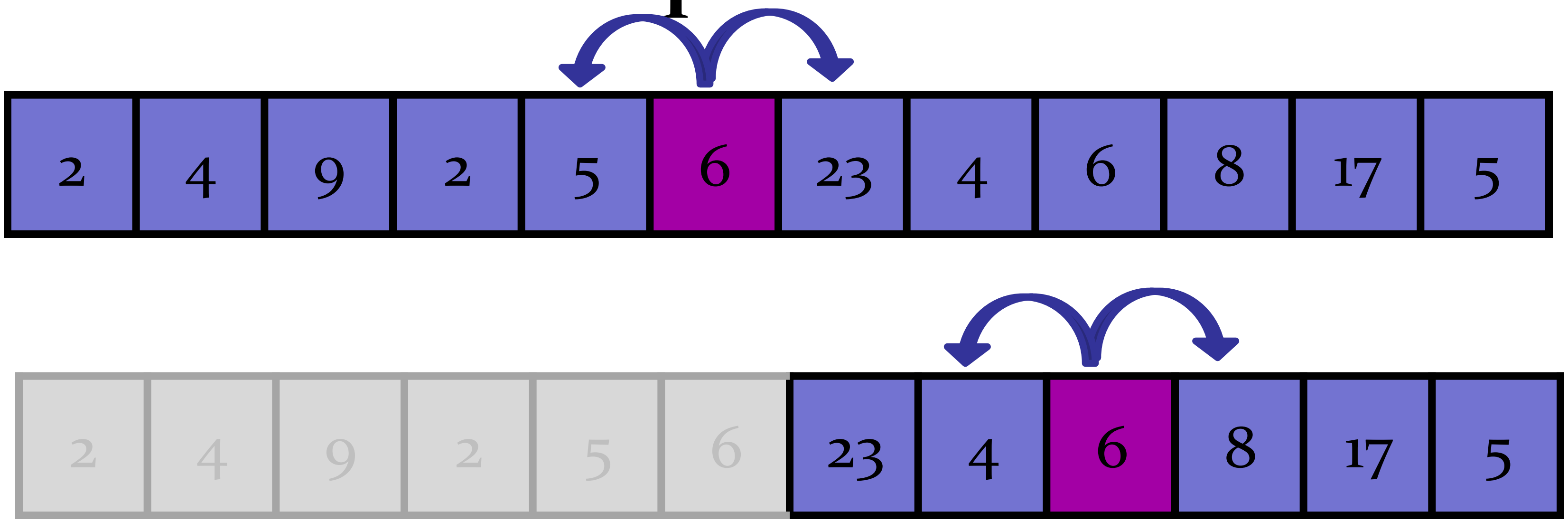
**6 > 23?** **NOT PEAK**

Recurse on the right side!

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

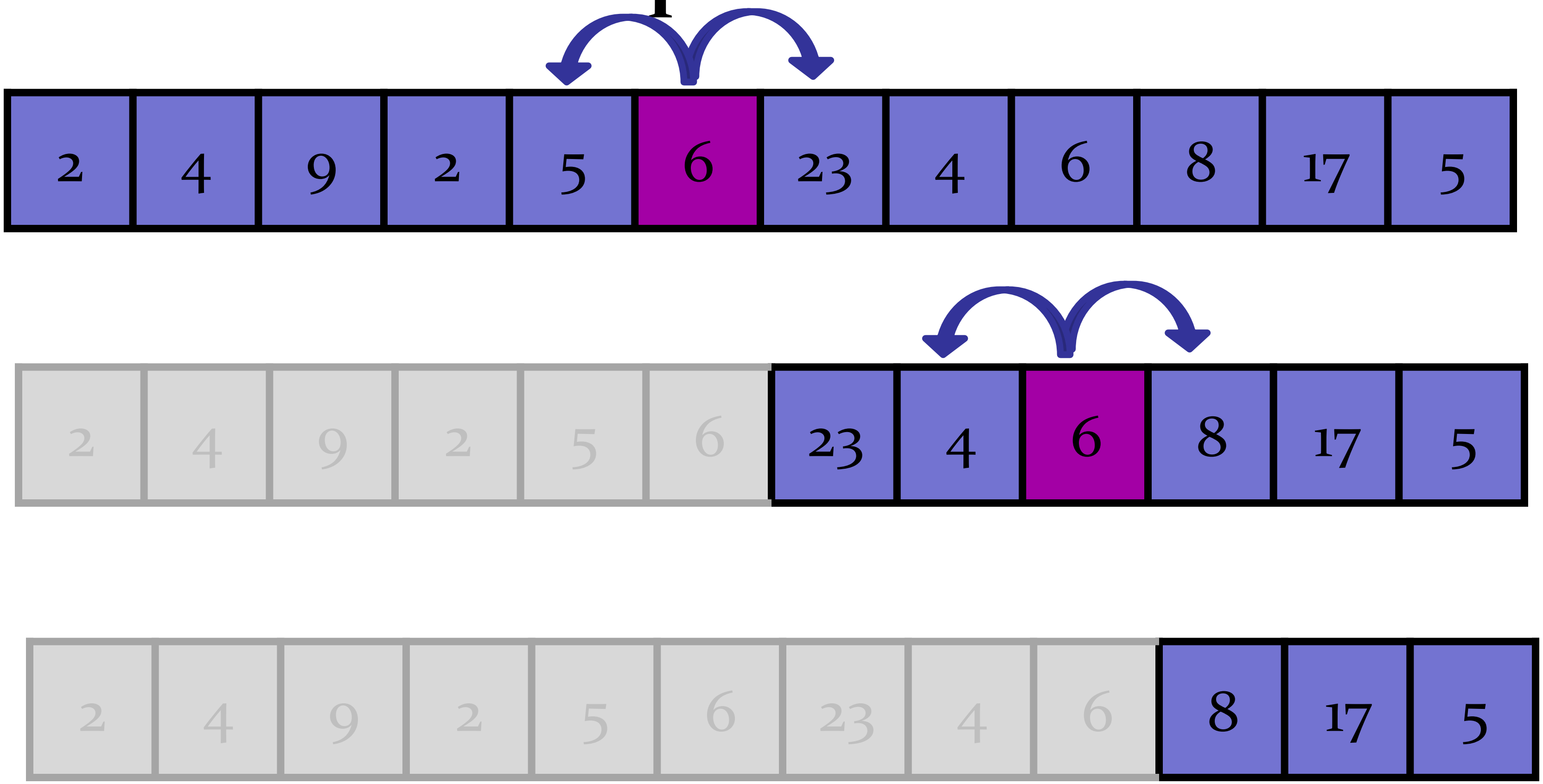
# Peak Finding: Algorithm 2

## Reduce-and-Conquer



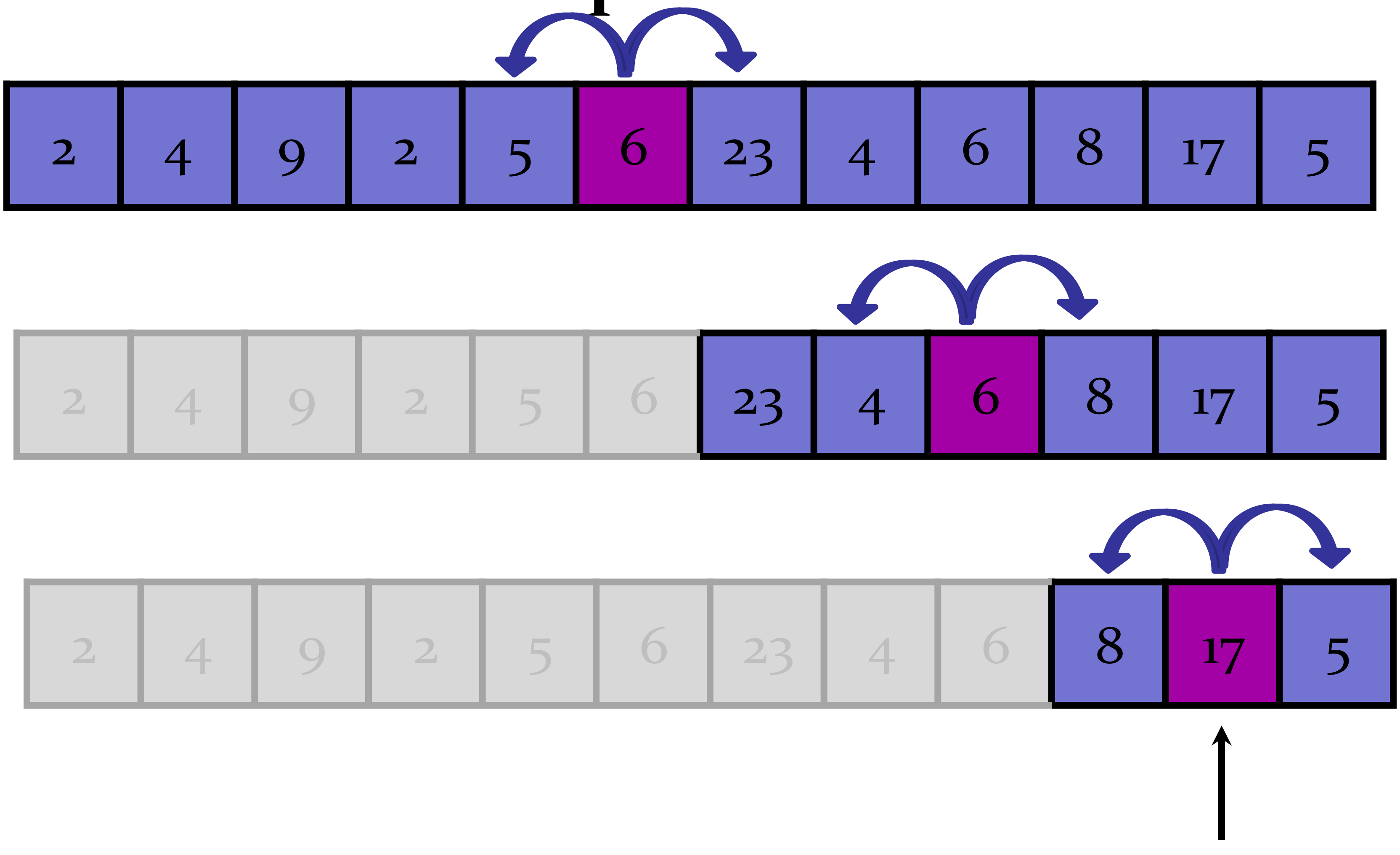
# Peak Finding: Algorithm 2

## Reduce-and-Conquer



# Peak Finding: Algorithm 2

**Reduce-and-Conquer**




We found a peak!

# Peak Finding

Input: Some array A[0..n-1]

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**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

Input: Some array $A[0..n-1]$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**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

## Is this correct?

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**FindPeak**(A, n)

Should this be >=?

**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)

Missing else condition?

# Peak Finding

Should this be >=? No: recurse on the larger half.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**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

Clarification:

Clarification: If we swap this from > to >=, either is okay.
So using > is not a bug. (It still leads to a correct answer)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**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

Missing else condition? No: else we have found a peak!

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**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

Missing else condition? No: else we have found a peak!

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**FindPeak**(A, n)

    **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)

    **else** A[n/2] is a peak; **return** n/2

# Peak Finding

Missing else condition? No: else we have found a peak!

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

n or n - 1?

**FindPeak**(A, n)

    **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)

    **else** A[n/2] is a peak; **return** n/2

# Peak Finding

Missing else condition?  No: else we have found a peak!

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

is n / 2 correct?

**FindPeak**(A, n)

  **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)

  **else** A[n/2] is a peak; **return** n/2

# Peak Finding

Key property → invariant:

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

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

# Peak Finding

Key property:

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

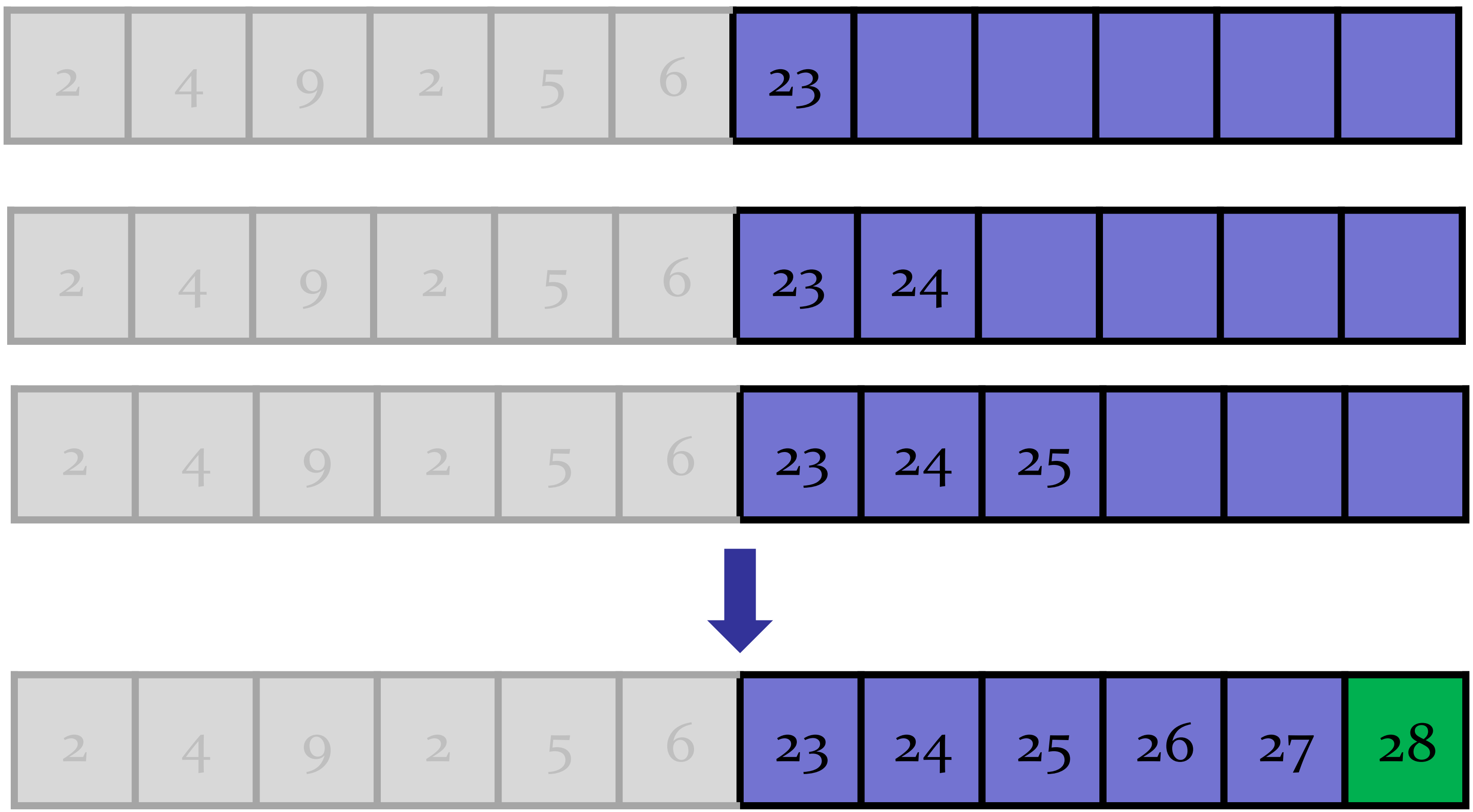
Explanation:

- Assume 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-2] < A[n-1] ← PEAK!!

# Peak Finding

Recurse on right half, since 23 > 6.

Assume no peaks in right half.



# Peak Finding

Key property:

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

Explanation:

- Assume there is "no peak" in the right half.
- Because we recursed right: 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-2] < A[n-1] ⟵ PEAK!!

# Peak Finding

Key property:

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

Induction:

- Assume there is “no peak” in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < A[j]

# Peak Finding

Key property:

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

Induction:

- Assume there is "no peak" in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < A[j]

- Base case: j = middle+1

  Because we recursed on the right half, we know that A[middle] < A[middle + 1].

# Peak Finding

Key property:

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

Induction:

- Assume there is “no peak” in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < A[j]
- Induction: j > middle+1

  By induction, A[j-2] <= A[j-1].

  If A[j-1] >= A[j], then A[j-1] is a peak $\Rightarrow$ contradiction.

# Peak Finding

Key property:

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

Induction:

- Assume there is "no peak" in the right half.
- Inductive hypothesis:

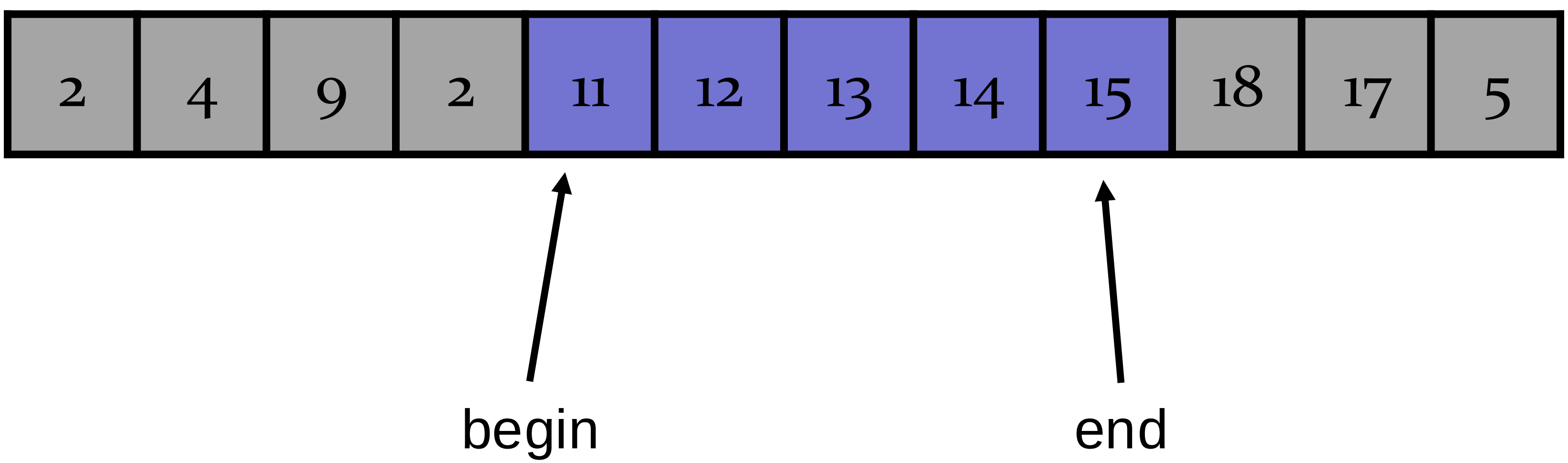
  For all (j > middle): A[j-1] < A[j]
- Conclusion: A[n-2] < A[n-1]

⇨ A[n-1] is a peak ⇨ contradiction.

# Key Invariants:

**Proposed invariant, does it work?**

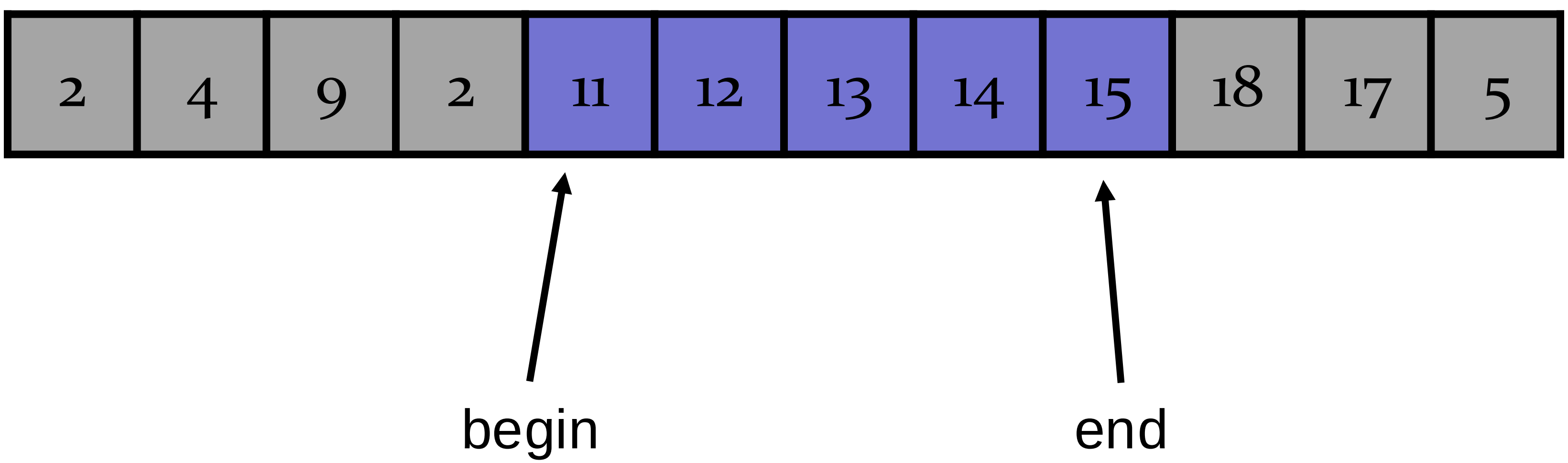
There exists a peak in the range [begin, end].



# Key Invariants:

**Is this good enough to prove the algorithm works?**

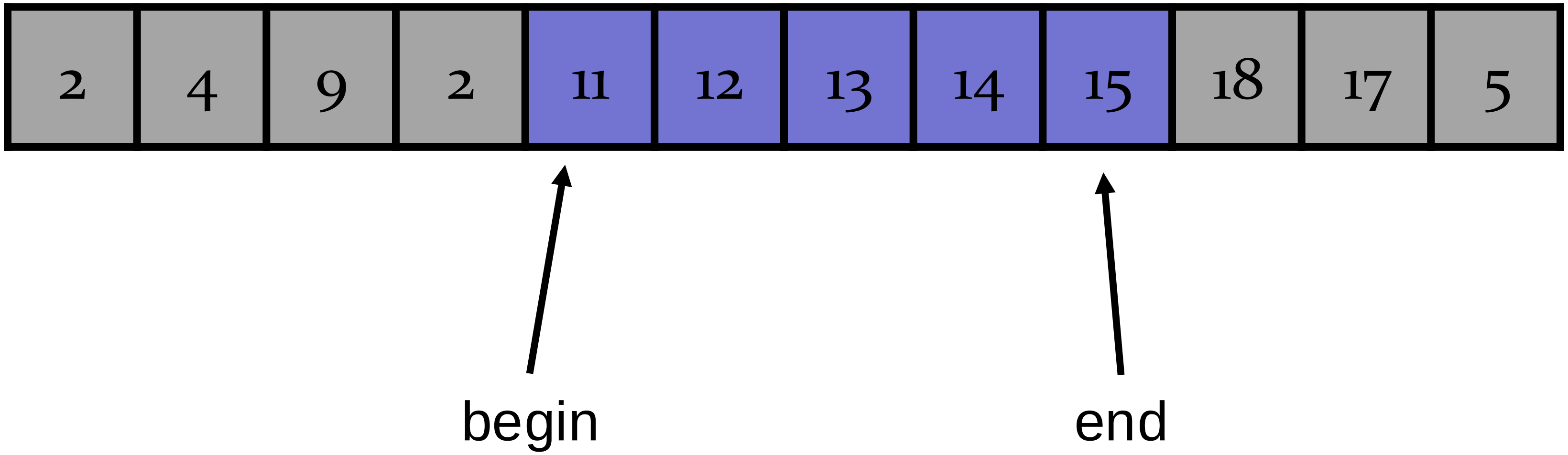
There exists a peak in the range [begin, end].



# Key Invariants:

**Not good enough to prove the algorithm works!**

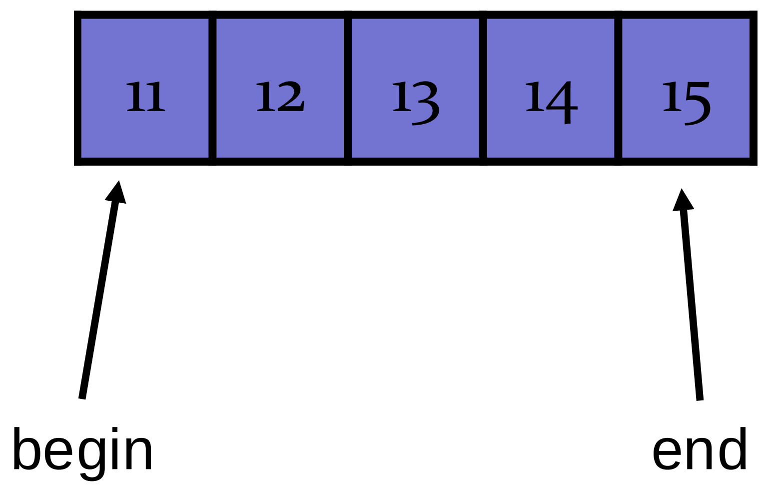
There exists a peak in the range [begin, end].



# Key Invariants:

**Not good enough to prove the algorithm works!**

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



# Key Invariants:

**Not good enough to prove the algorithm works!**

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

| | | | | |
|---|---|---|---|---|
| 11 | 12 | 13 | 14 | 15 |

Run peak finding algorithm → returns 15

# Key Invariants:

**Not good enough to prove the algorithm works!**

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 12 | 13 | 14 | 15 | 18 | 17 | 5 |

Run peak finding algorithm, then returns 15

But 15 is NOT a peak!

If the recursive call finds a peak, is it still a peak after the recursive call returns?

# Key Invariants:

**Correctness:**

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

2. Every peak in[begin, end] is a peak in [0, n-1].

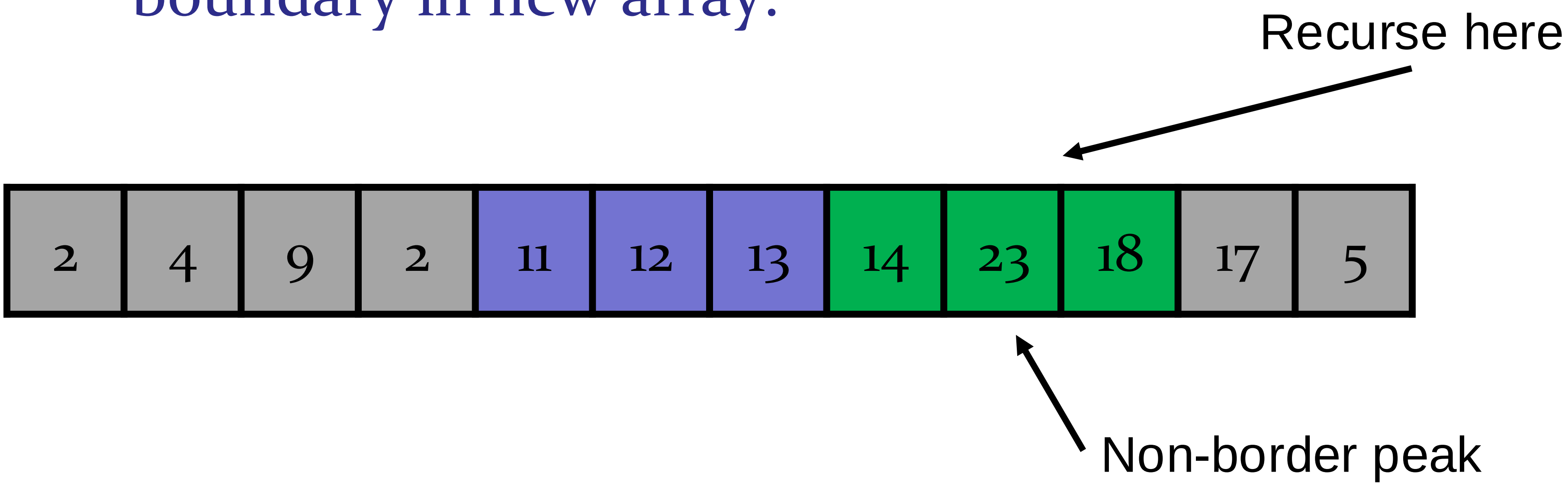
# Peak Finding

Key property:

- If we recurse in the right half, then every peak in the right half is a peak in the array.

Proof: use the invariant (inductively)

- Immediately true for every peak that is not at a boundary in new array.



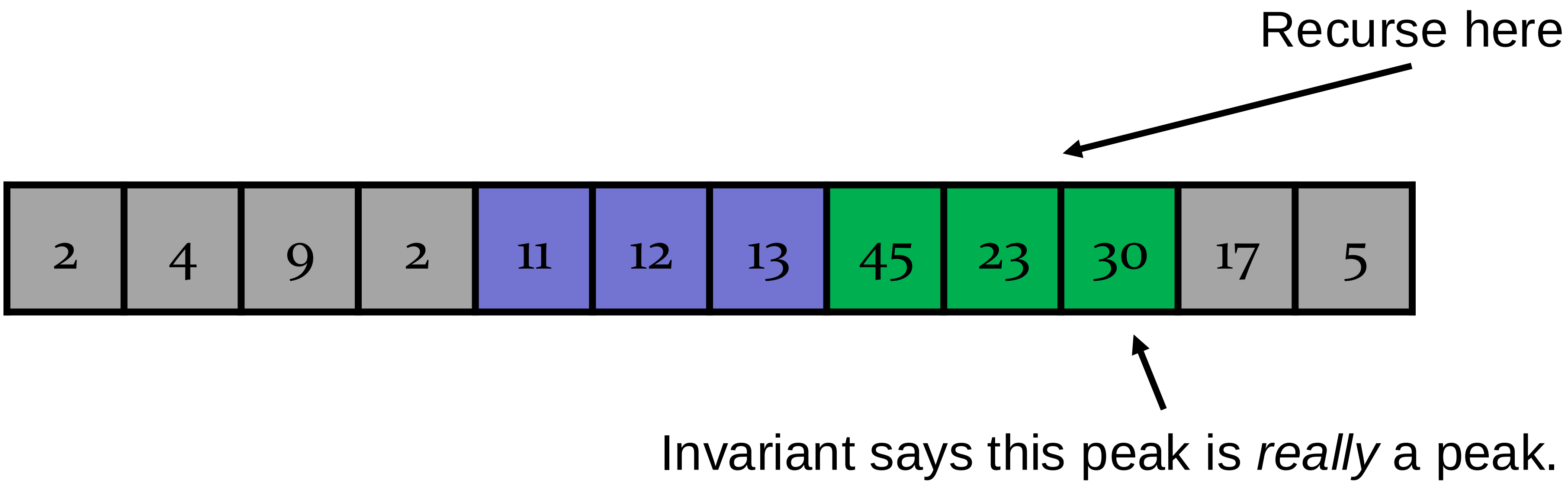
# Peak Finding

Key property:

- If we recurse in the right half, then every peak in the right half is a peak in the array.

Proof: use the invariant (inductively)

- True by invariant for current array.



# Peak Finding

Key property:

- If we recurse in the right half, then every peak in the right half is a peak in the array.

Proof: use the invariant (inductively)

- If 45 is a peak in the new array but not the old array, then we would not recurse on the right side.

⇨ If left edge is a peak in new array, then it is a peak.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 12 | 60 | 45 | 23 | 30 | 17 | 5 |

If 45 is a peak in right half and we recurse on right half, then it is a peak.

# Key Invariants:

**Correctness:**

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

2. Every peak in[begin, end] is a peak in [0, n-1].

# Peak Finding

## Running time?

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**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

**Running time:**

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

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

# Peak Finding

**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

Unrolling the recurrence:

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

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

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

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

$$\ldots$$

$$\ldots$$

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

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

# Peak Finding

Unrolling the recurrence:

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

$$
\begin{aligned}
T(n) &= T(n/2) + \Theta(1) \\
&= T(n/4) + \Theta(1) + \Theta(1) \\
&= T(n/8) + \Theta(1) + \Theta(1) + \Theta(1) \\
&\qquad \cdots \\
&\qquad \cdots \\
&= T(1) + \Theta(1) + \ldots + \Theta(1) = \\
&= \Theta(1) + \Theta(1) + \ldots + \Theta(1) =
\end{aligned}
$$

Number of times you can divide n by 2 until you reach 1.

# Peak Finding

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

**Running time:**

Time to find a peak in an array of size n

Recursion

Time for comparing A[n/2] with neighbors

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

Unrolling the recurrence:

$$T(n) = \underbrace{\Theta(1) + \Theta(1) + \ldots + \Theta(1)}_{\text{at most } \log(n) \text{ many terms}} = O(\log n)$$

# Peak Finding

Input: Some array A[0..n-1]

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 8 | 21 |

Output: a local maximum in A

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

Assume that

`A[-1] = A[n] = -MAX_INT`

# What about Steep Peaks?

# Steep Peaks

Input: Some array A[0..n-1]

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 8 | 5 |

Output: a local maximum in A

`A[i-1] < A[i]` **and** `A[i+1] < A[i]`

Assume that

`A[-1] = A[n] = -MAX_INT`

# Steep Peaks

Input: Some array A[0..n-1]

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 8 | 5 |

Output: a local maximum in A

`A[i-1]` **<** `A[i]` **and** `A[i+1]` **<** `A[i]`

Can we find *steep* peaks efficiently (in O(log n) time) using the same approach?

# Steep Peaks

Problematic example:

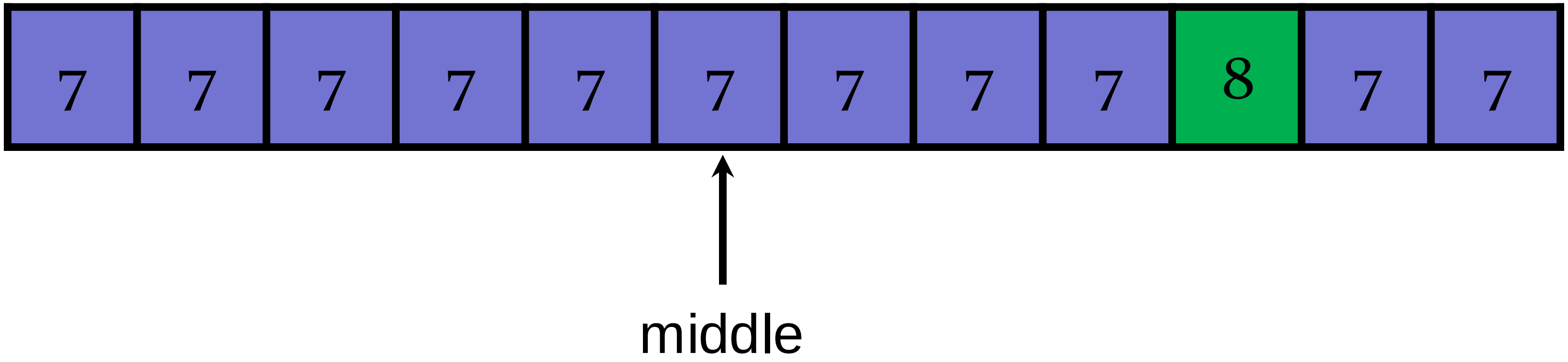
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 7 | 7 |

<u>Inuitively:</u>

There are n different positions to search for the steep peak, and no hints as to where it might be found!

# Steep Peaks

Problematic example:




Which side does the algorithm recurse on?

# Regular Peaks vs Steep Peaks

Missing else condition? We have found a peak, but not a steep peak!

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 9 | 2 | 11 | 6 | 23 | 4 | 6 | 8 | 17 | 5 |

**FindPeak**(A, n)

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

    **FindPeak** (A[n/2+1..n-1], n/2)

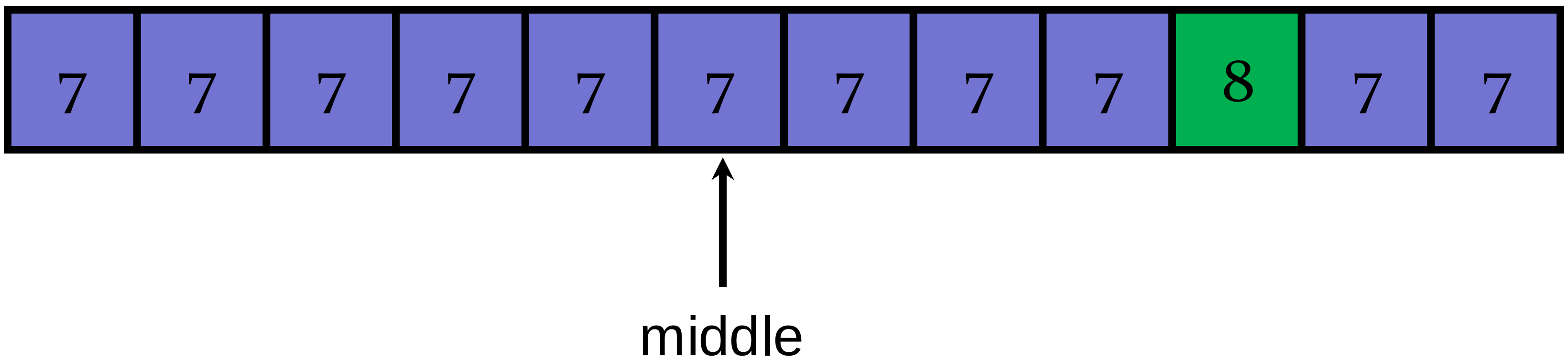
  **else if** A[n/2–1] > A[n/2] **then**

    **FindPeak** (A[1..n/2-1], n/2)

  **else** A[n/2] is a peak; **return** n/2

# Steep Peaks

Problematic example:




What happens if you recurse on both sides?

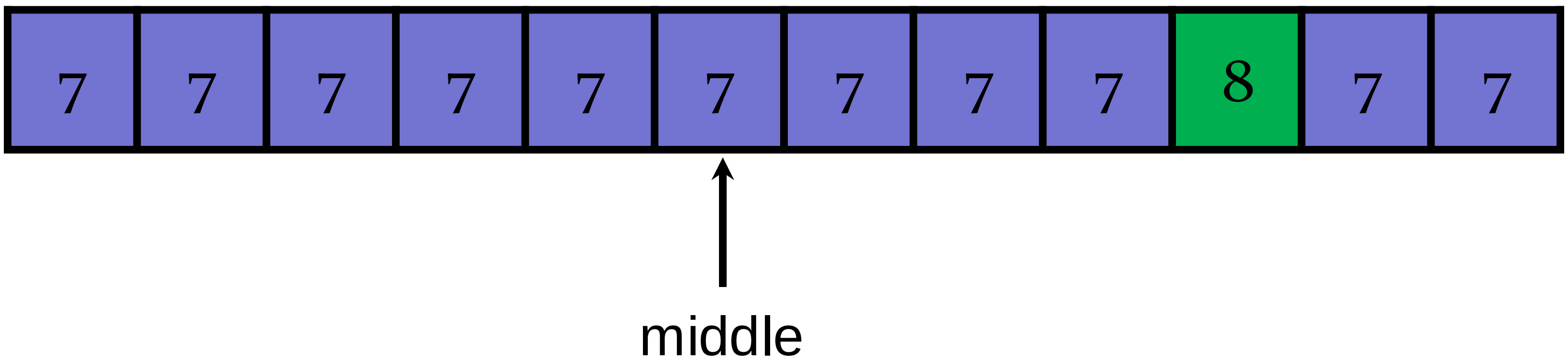
…

**if** A[n/2-1] == A[n/2] == A[n/2+1] **then**

Recurse on left & right sides

# Steep Peaks

Problematic example:




What happens if you recurse on both sides?

Recurrence: $T(n) = 2T(n/2) + O(1)$

# Steep Peak Finding

Unrolling the recurrence:

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

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

$$= 2(2T(n/4) + 1) + 1 = 4T(n/4) + 2 + 1$$

$$= 8T(n/8) + 4 + 2 + 1$$

$$= 16T(n/16) + 8 + 4 + 2 + 1$$

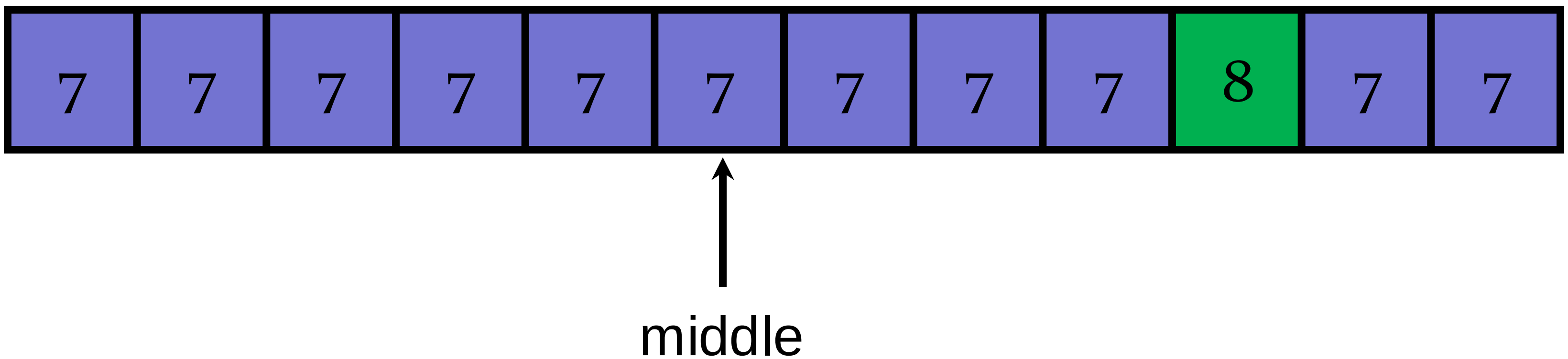
$$\ldots$$

$$= nT(1) + n/2 + n/4 + n/8 + \ldots + 1 =$$

$$= n + n/2 + n/4 + n/8 + \ldots + 1 = \Theta(n)$$

# Steep Peaks

Problematic example:




What happens if you recurse on both sides?

Recurrence: $T(n) = 2T(n/2) + O(1) = O(n)$

# 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]

$$
n\left\{
\overbrace{
\begin{array}{|c|c|c|c|c|}
\hline
10 & 8 & 5 & 2 & 1 \\
\hline
3 & 2 & 1 & 5 & 7 \\
\hline
17 & 5 & 1 & 4 & 1 \\
\hline
7 & 9 & 4 & 6 & 4 \\
\hline
8 & 1 & 1 & 2 & 6 \\
\hline
\end{array}
}^{m}
\right.
$$

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.

# 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 <u>peak</u> in the array of max elements.

Is this algorithm correct?

# 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.

Is this algorithm correct? YES

# 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.

Is this algorithm efficient?

# 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.

Is this algorithm efficient? NO

# 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 (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.

Is this algorithm correct and/or efficient?

# 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 <u>peak</u> in the array of peaks.

Is this algorithm correct? 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.

Is this algorithm efficient? Yes.

# 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 3

~~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 <u>peak</u> 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: Find max of middle column.

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 | ? | ? | ? | ? | ? |

Column Max Array

Find 1D Peak:

Step 1: Find max of middle column.

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 ? 18 8 17 ?

Find 1D Peak:

Step 1: Find max of middle column.

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 ? 18 8 17 8

Find 1D Peak:

Step 1: Find max of middle column.

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 | ? | 18 | 8 | 17 | 8 |

How many columns do we need to examine?

1. O(m)
2. O(√m)
3. O(log m)
4. O(1)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 17 8

How many columns do we need to examine?

1. $O(m)$
2. $O(\sqrt{m})$
3. $O(\log m)$
4. $O(1)$

# 2D: 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 max
- Total: $\mathbf{O(n \log m)}$

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

# Remarks

- We can design a "more direct" Divide-and-Conquer algorithm with time $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{m})$

- Further improvement (using Divide-and-Conquer) to $\mathbf{O}(\boldsymbol{n} + \boldsymbol{m})$

- If interested, see OPTIONAL SLIDES

# 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)

# Announcements

PS2: Due next week.

Week 2 Lecture Trainings: Due tomorrow

Next week: Sorting!

Recorded Tutorials and Recitations for Week 3

# For Interested Readers

OPTIONAL SLIDES

# 2D Algorithm 4

**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.

| | | | | |
|---|---|---|---|---|
| 10 | 8 | 4 | 2 | 1 |
| 3 | 2 | 2 | 12 | 13 |
| 17 | 5 | 1 | 11 | 1 |
| 7 | 4 | 6 | 9 | 4 |
| 8 | 1 | 1 | 2 | 6 |

recurse right

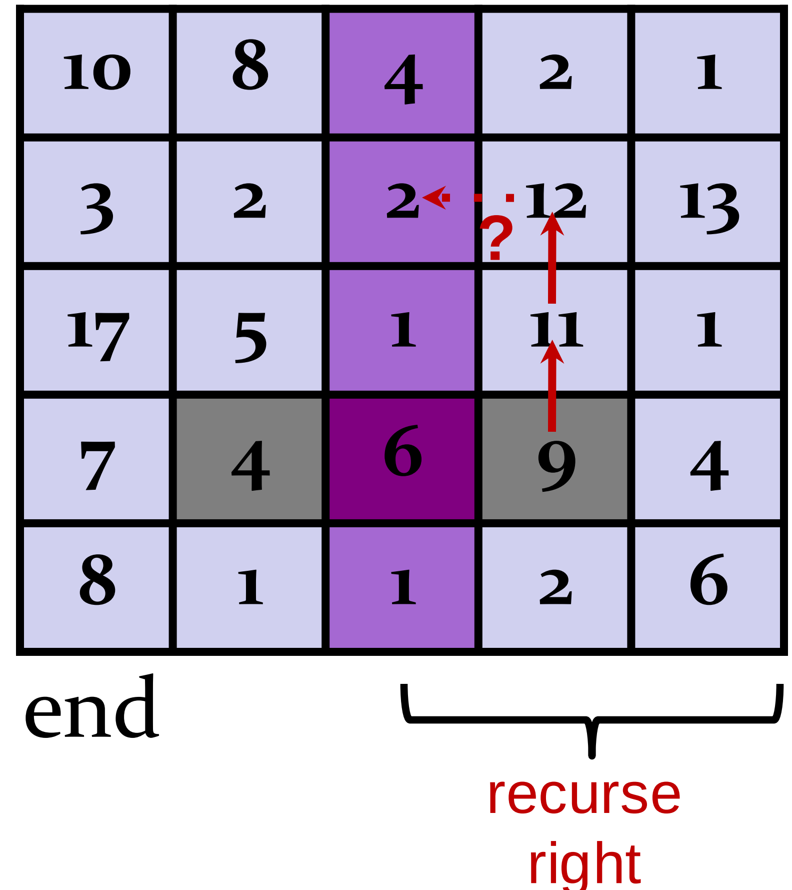
# 2D Algorithm 4

**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 maximum element.
4. If there is no max in the right half, then it must cross to the left half… Impossible!



# 2D Algorithm 4

## Reduce-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.

| | | | | |
|---|---|---|---|---|
| 10 | 8 | 4 | 2 | 1 |
| 3 | 2 | 2 | 12 | 13 |
| 17 | 5 | 1 | 11 | 1 |
| 7 | 4 | 6 | 9 | 4 |
| 8 | 1 | 1 | 2 | 6 |

recurse right

# Recurrence Analysis

ARCHIPELAGO

is open

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

$T(n,m) = ??$

# Recurrence Analysis

$$\left.\begin{aligned}
T(n, m) &= T(n, m/2) + n \\
&= T(n, m/4) + n + n \\
&= T(n, m/8) + n + n + n \\
&= T(n, m/16) + n + n + n + n \\
&= \dots \\
&= \dots \\
&= \dots \\
&= \underbrace{T(n, 1)}_{n} + \underbrace{n + n + n + \dots + n}_{\log(m)}
\end{aligned}\right\} \log(m)$$

# Recurrence Analysis

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

$T(n,m) = ??$

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

# 2D Algorithm 4

**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)**

| | | | | |
|---|---|---|---|---|
| 10 | 8 | 4 | 2 | 1 |
| 3 | 2 | 2 | 12 | 13 |
| 17 | 5 | 1 | 11 | 1 |
| 7 | 4 | 6 | 9 | 4 |
| 8 | 1 | 1 | 2 | 6 |

recurse right

# 2D Algorithm 4

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

# 2D Algorithm 5

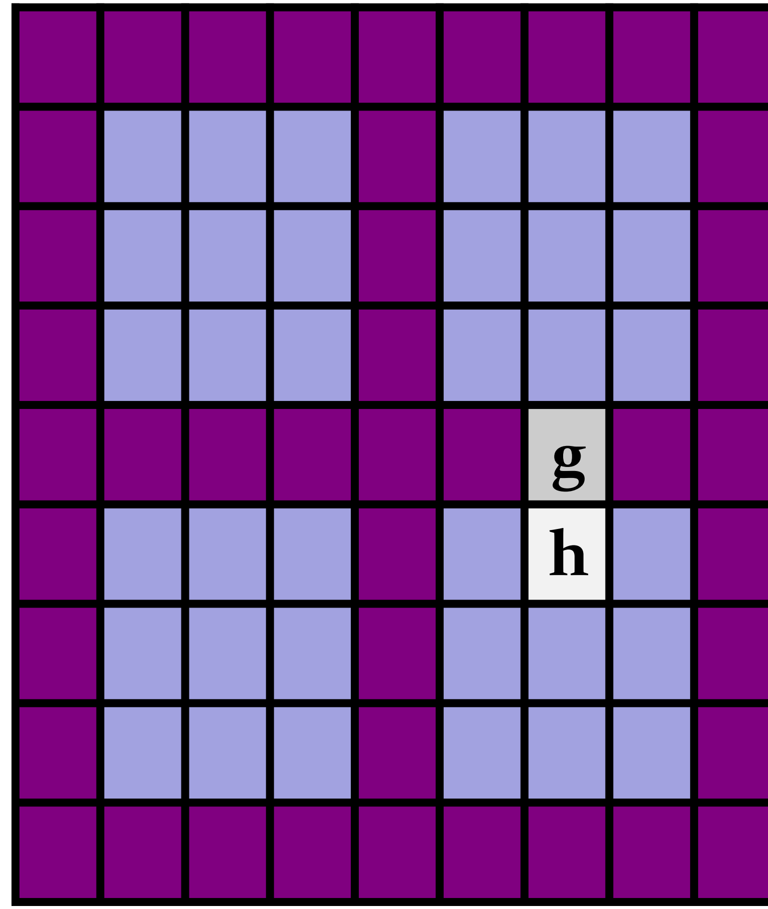
**Reduce-and-Conquer**

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

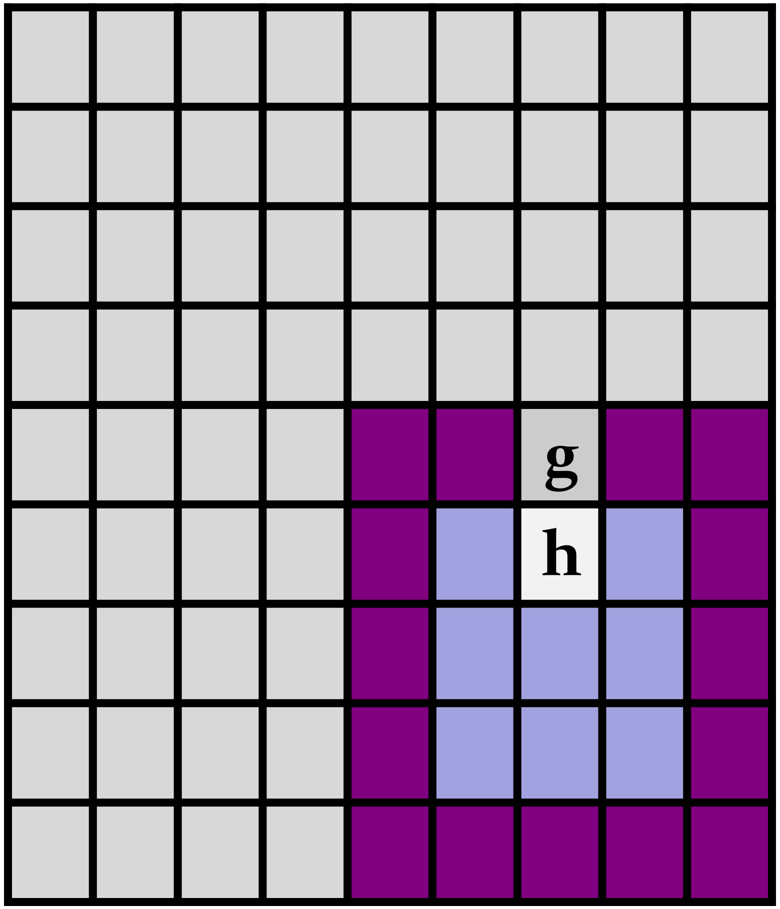


# 2D Algorithm 5

**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

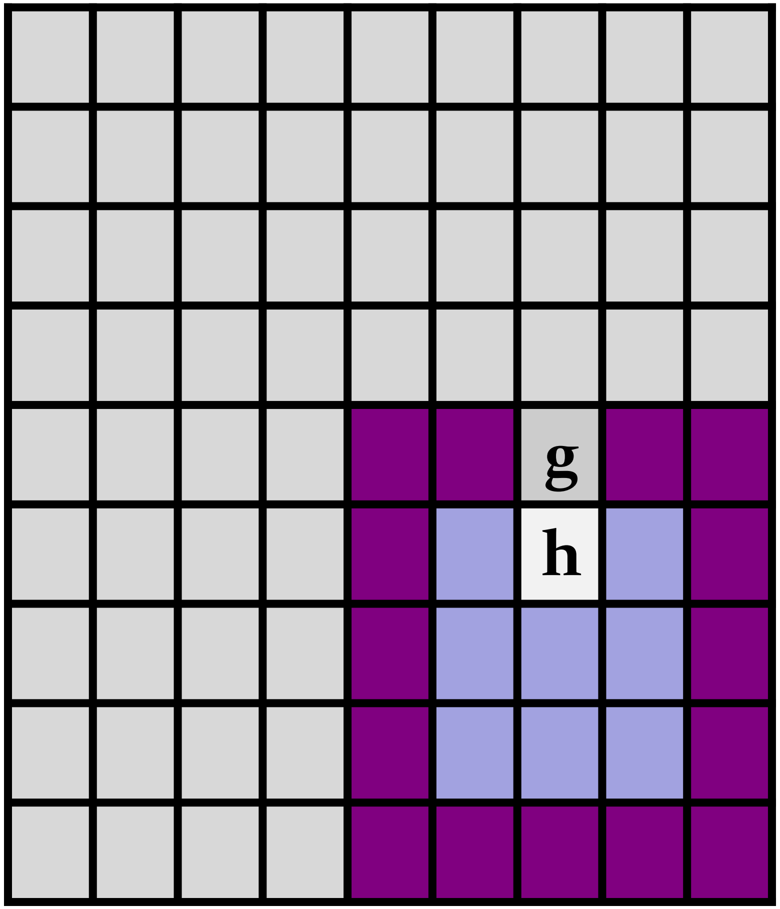
# 2D Algorithm 5

**Correctness**

1. The quadrant contains a peak.

   Proof: as before.

   If there is no peak, then you can find an increasing path that keeps going until you either find a maximum element (a peak) or it exits the quadrant.



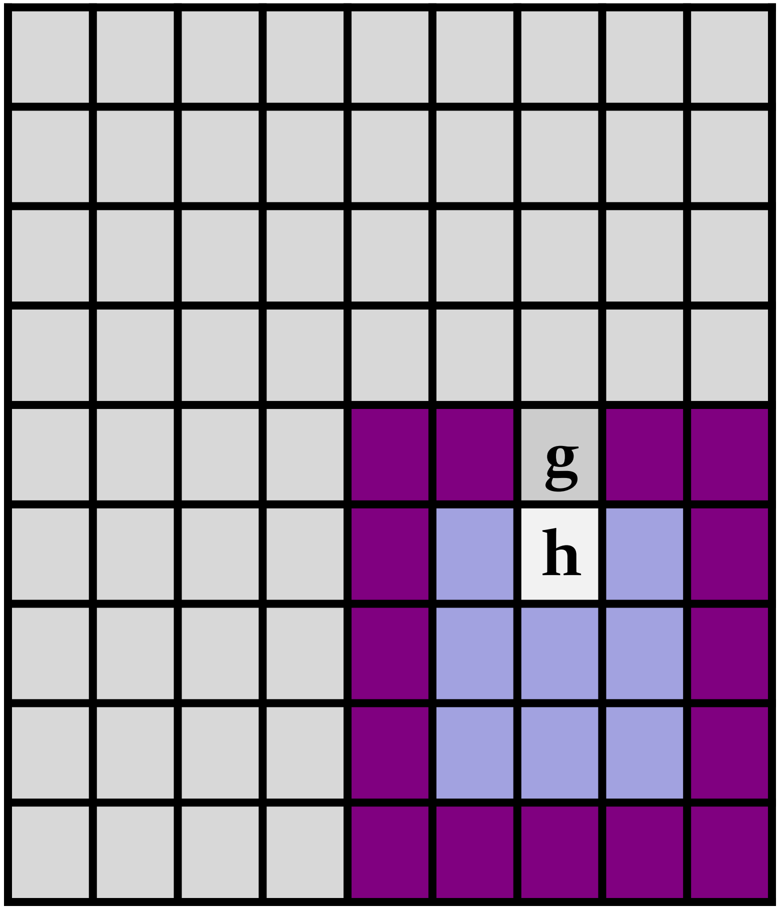
# 2D Algorithm 5

**Correctness**

1. The quadrant contains a peak.

   Proof: as before.

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



# 2D Algorithm 5

**Correctness**

1. The quadrant contains a peak.

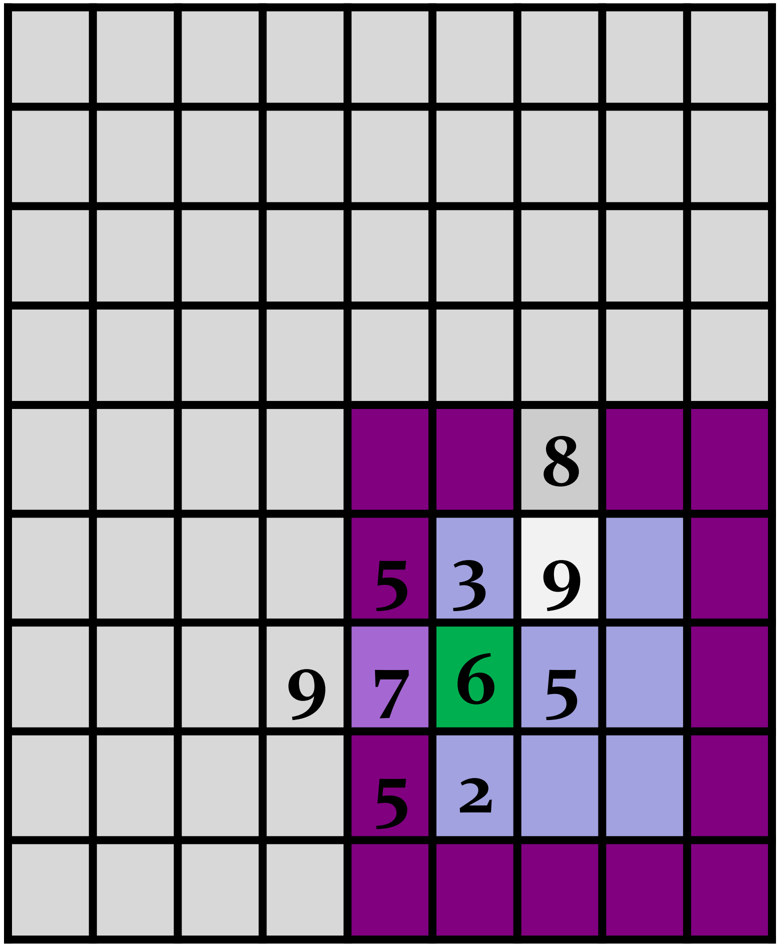
   Proof: as before.

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

   Example: 7 > 6 > 5

   6 > 3

   6 > 2




6 is a peak in the quadrant, but not in the matrix.

# 2D Algorithm 5

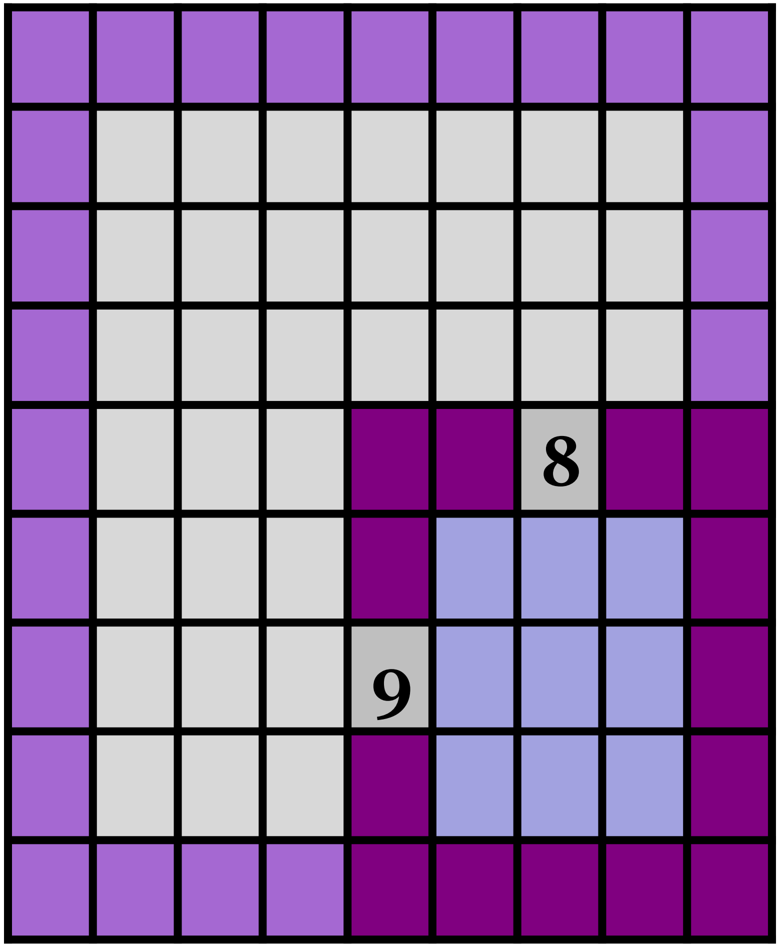
## Correctness

Key property:

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

Why is this enough?

If recursing finds an element at least as large as 9, and 9 is as big as the biggest element on the boundary, then the peak is as large as every element on the boundary.



# 2D Algorithm 5

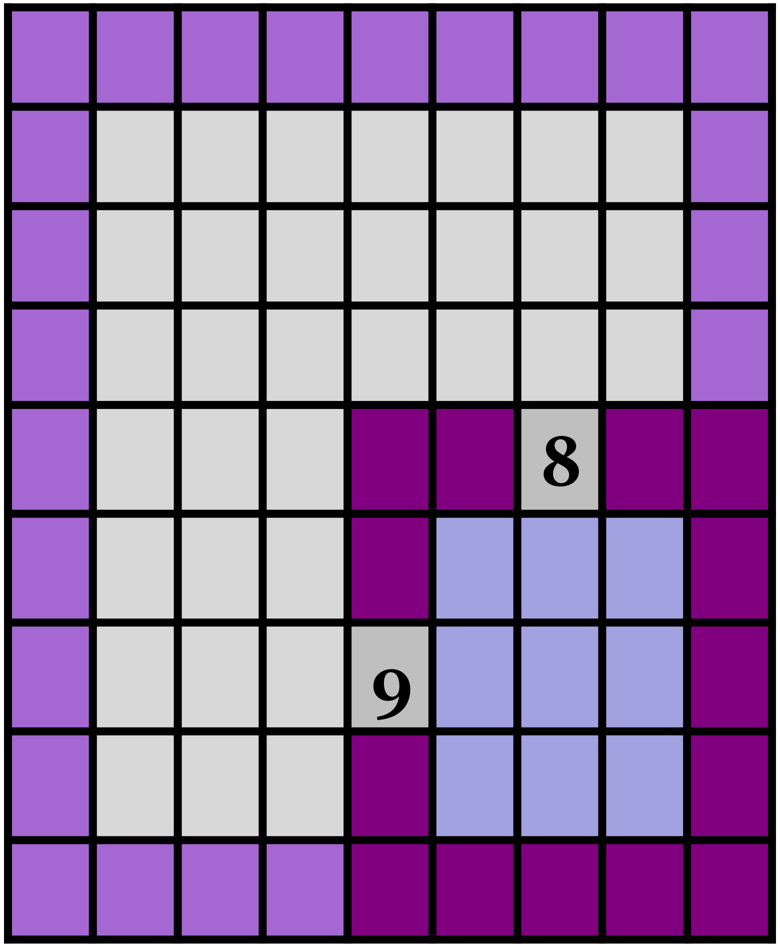
**Correctness**

Key property:

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

Proof:

(Slightly) tricky exercise...



# 2D Algorithm 4

**Reduce-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

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

# Recurrence Analysis

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