CS2040S
Data Structures and Algorithms

Welcome!

ARCHIPELAGO

is open

How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

Admin

Zoom Chat

60% of respondants said chat was distracting

< 20% of respondants said chat was useful

Conclusion:

Please do not chat during class on Zoom.

If possible, I will leave it enabled for urgent questions, if you want to ask me a question, and for chatting before and after class.

But if there is random chatter, I will disable.

Puzzle of the Week (Contest)

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

Puzzle of the Week (Contest)

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 (Contest)


Advanced version:

Each student you send on a mission costs 1 SGD.



Puzzle of the Week

Find:

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

Puzzle of the Week

Prove:

Assume:

N = 1,024 and K = 17.

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

Announcements

Competition:

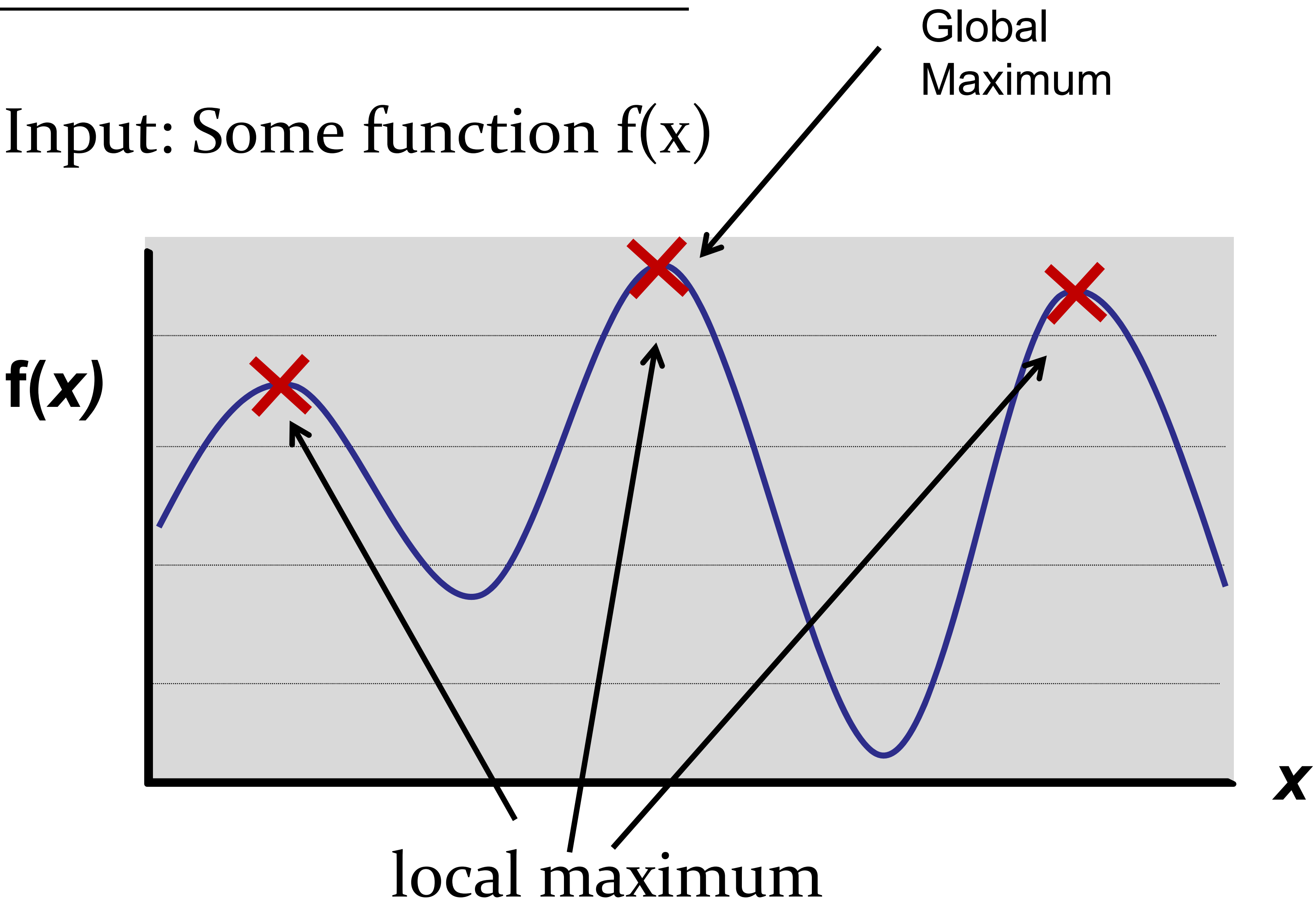
Find the spies!

Open on Coursemology this week:

- Optional.
- Write a program to implement your best spy catching strategy.
- We will test it on various spy rings.
- Fewest missions / fewest SGD wins!
- (And a small bonus for participating.)

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: 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

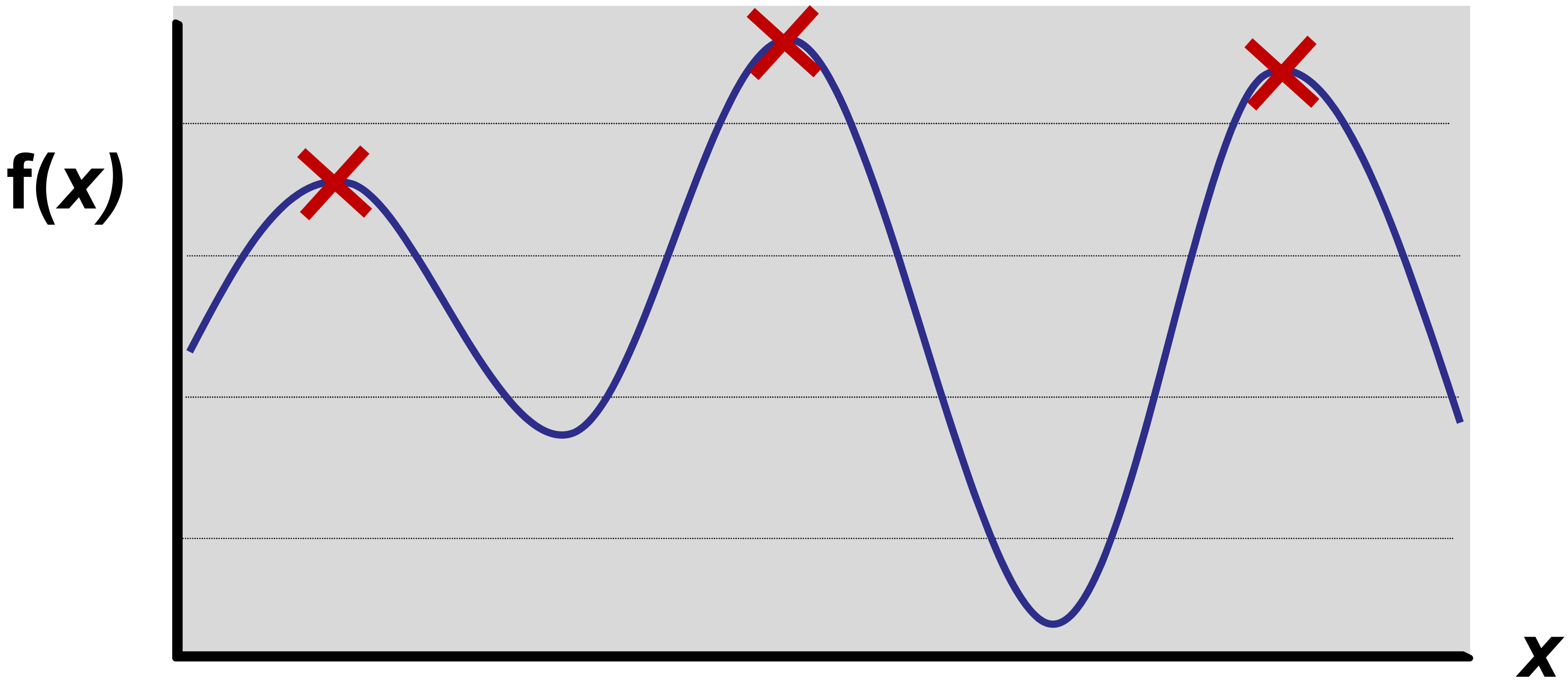
```
FindMax(A,n)
    max = A[1]
    for i = 1 to n do:
        if (A[i]>max) then max=A[i]
```

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[0..n-1]

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

Output: a local maximum in A

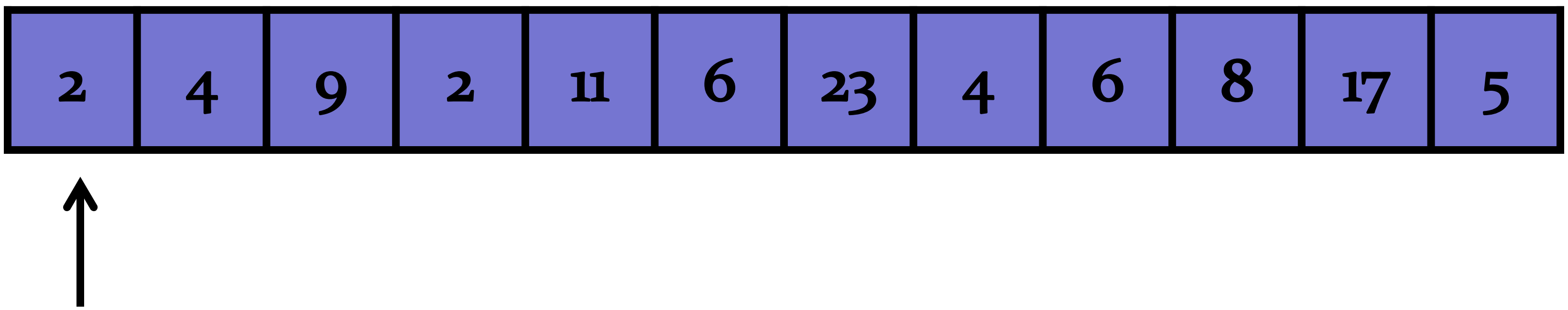
`A[i-1]` ≤ `A[i]` **and** `A[i+1]` ≤ `A[i]`

Assume that

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

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]`

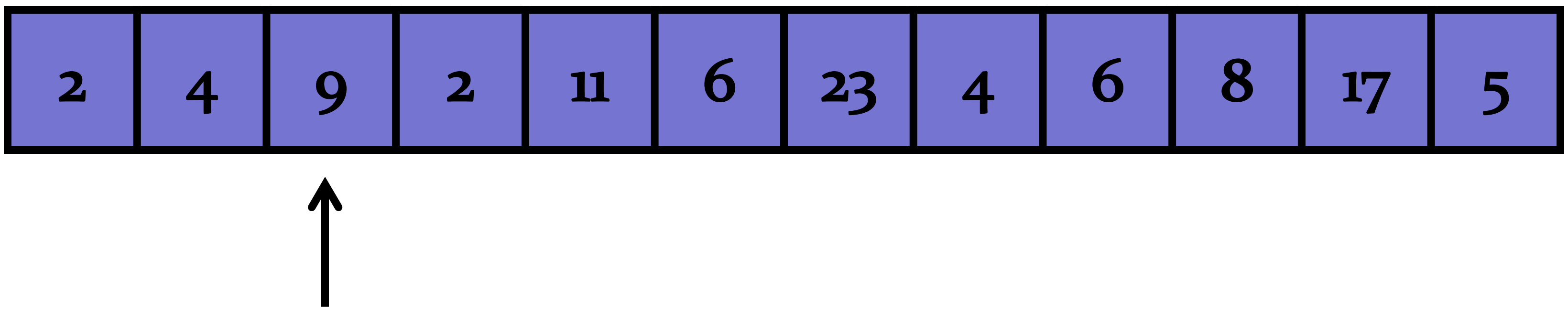
2	4	9	2	11	6	23	4	6	8	17	5

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

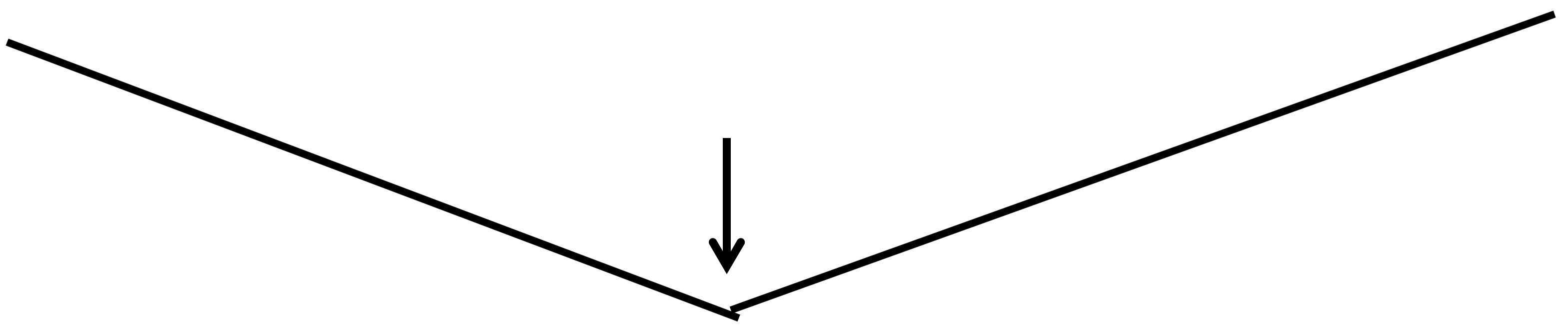
Simple improvement?

Peak Finding: Algorithm 1

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

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

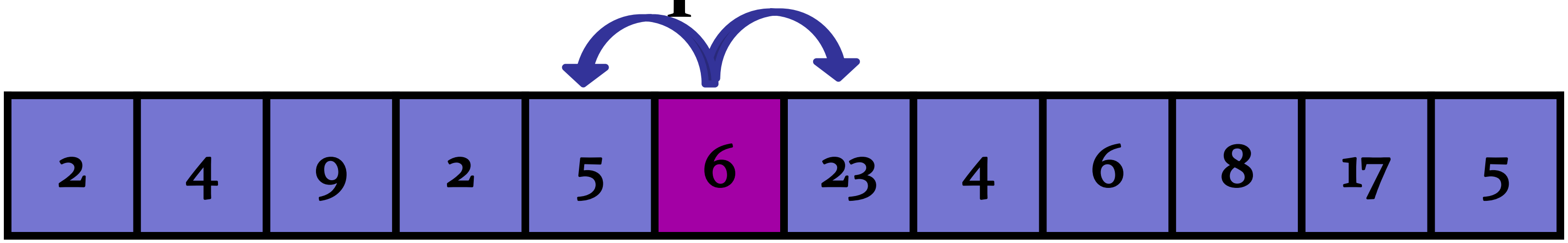
Start in the middle!



Worst-case: n/2

Peak Finding: Algorithm 2

Reduce-and-Conquer

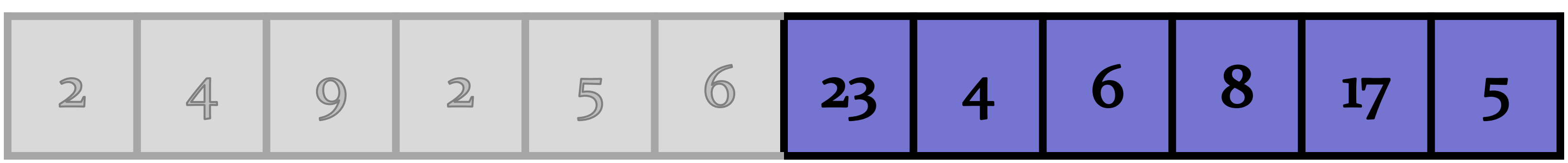


Start in the middle

5 < 6? **OK**

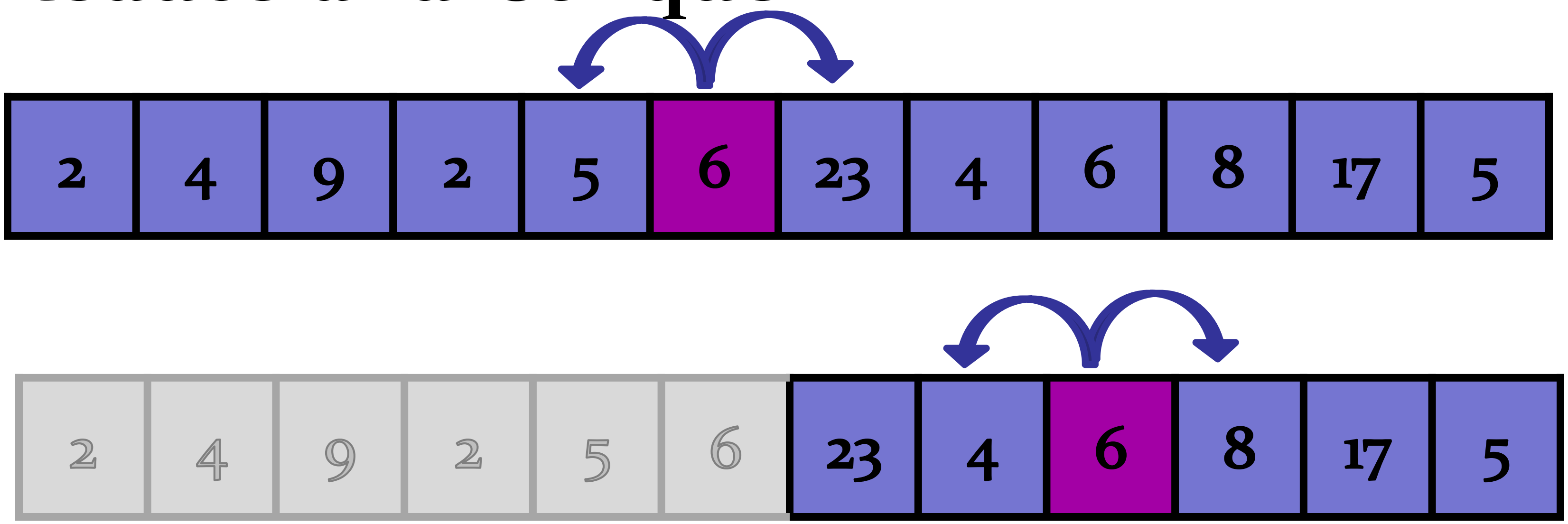
6 > 23? **NO**

Recurse!



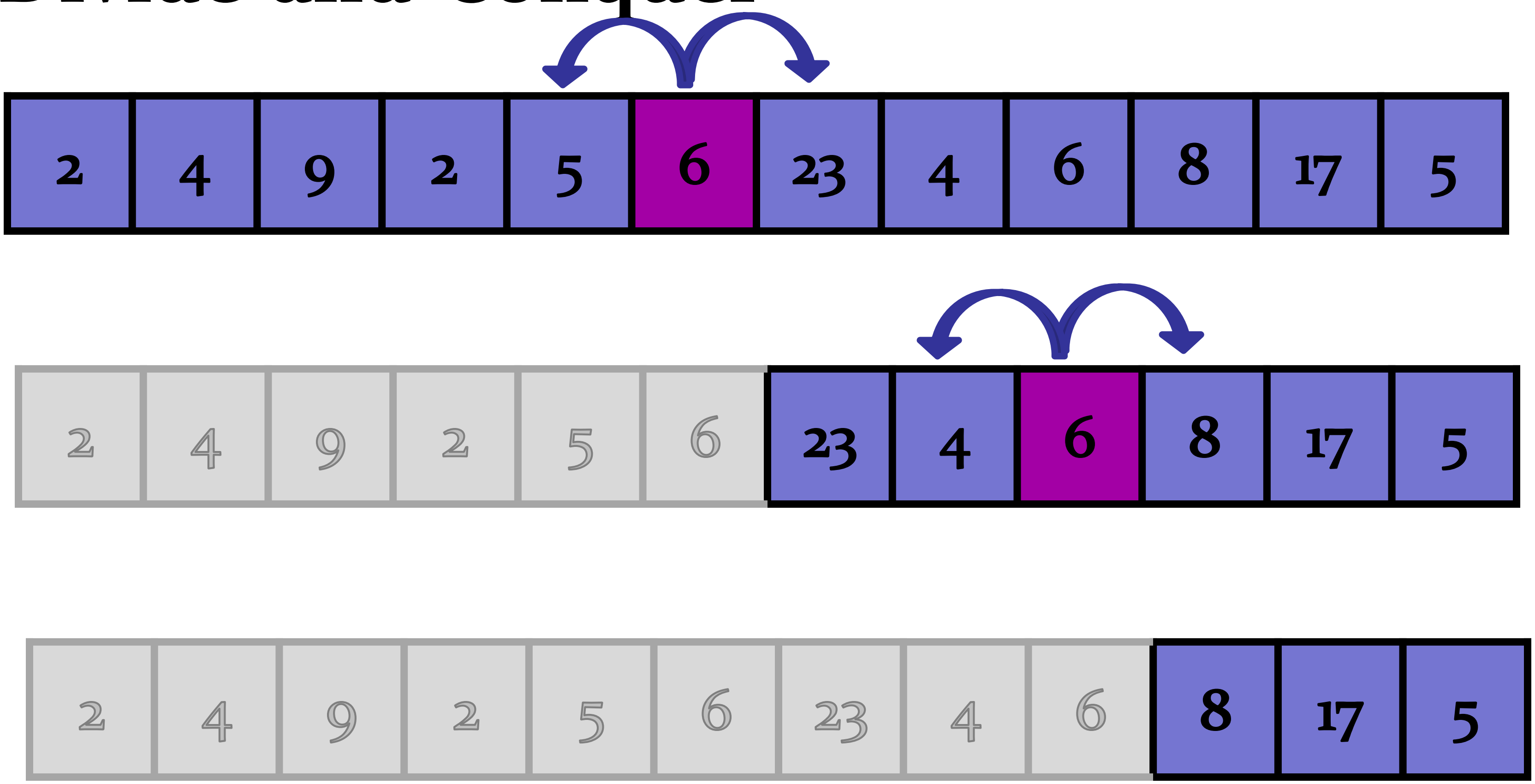
Peak Finding: Algorithm 2

Reduce-and-Conquer



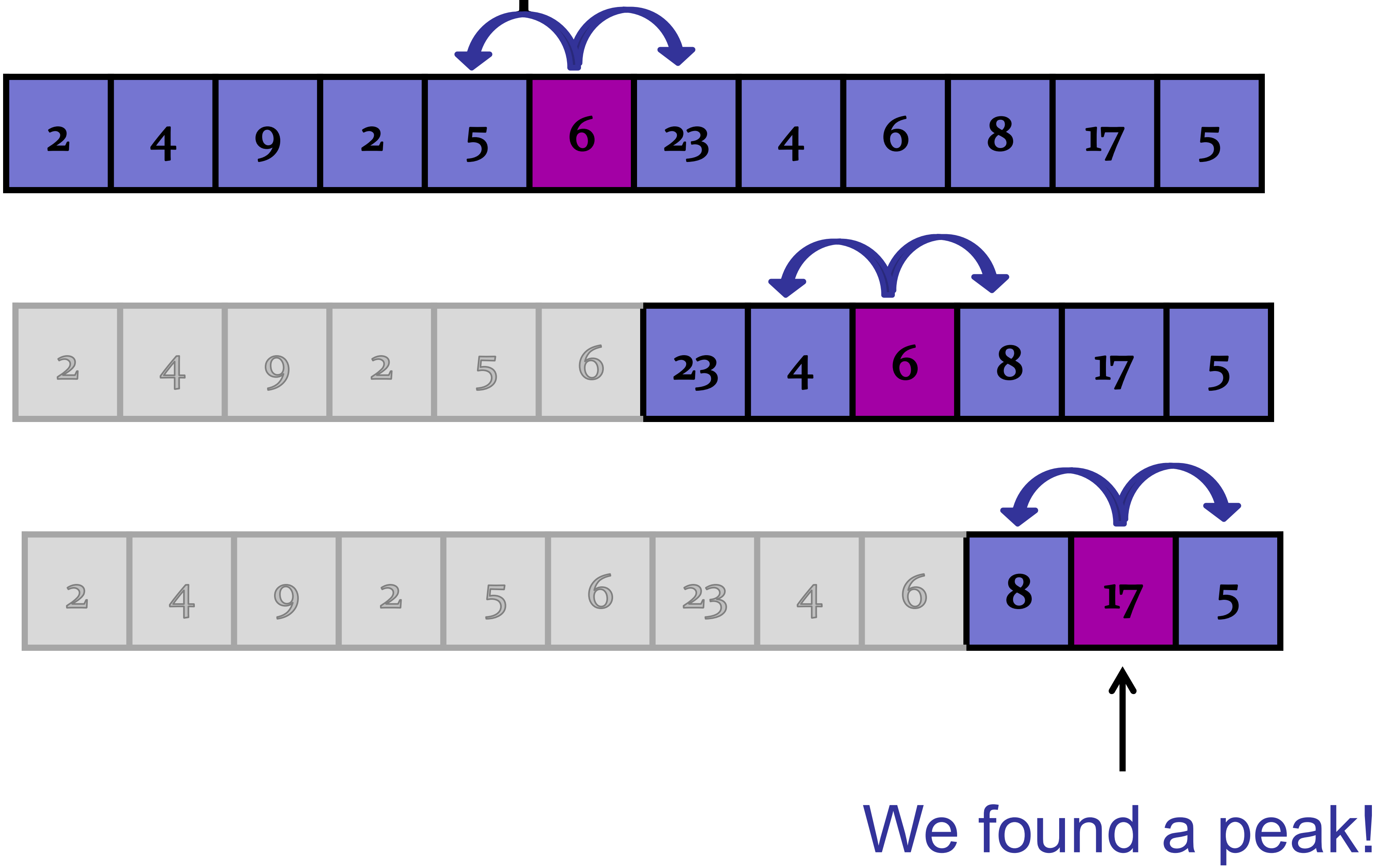
Peak Finding: Algorithm 2

Divide-and-Conquer



Peak Finding: Algorithm 2

Divide-and-Conquer



Peak Finding: Algorithm 2

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: Algorithm 2

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)

SLG1

Peak Finding: Algorithm 2

ARCHIPELAGO

is open

Is this correct?

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

Should this be >=?

```
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)
```

Missing else condition?

SLG1

Peak Finding: Algorithm 2

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)

SLG1

Peak Finding: Algorithm 2

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)

SLG1

Peak Finding: Algorithm 2

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: Algorithm 2

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: Algorithm 2

Key property:

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

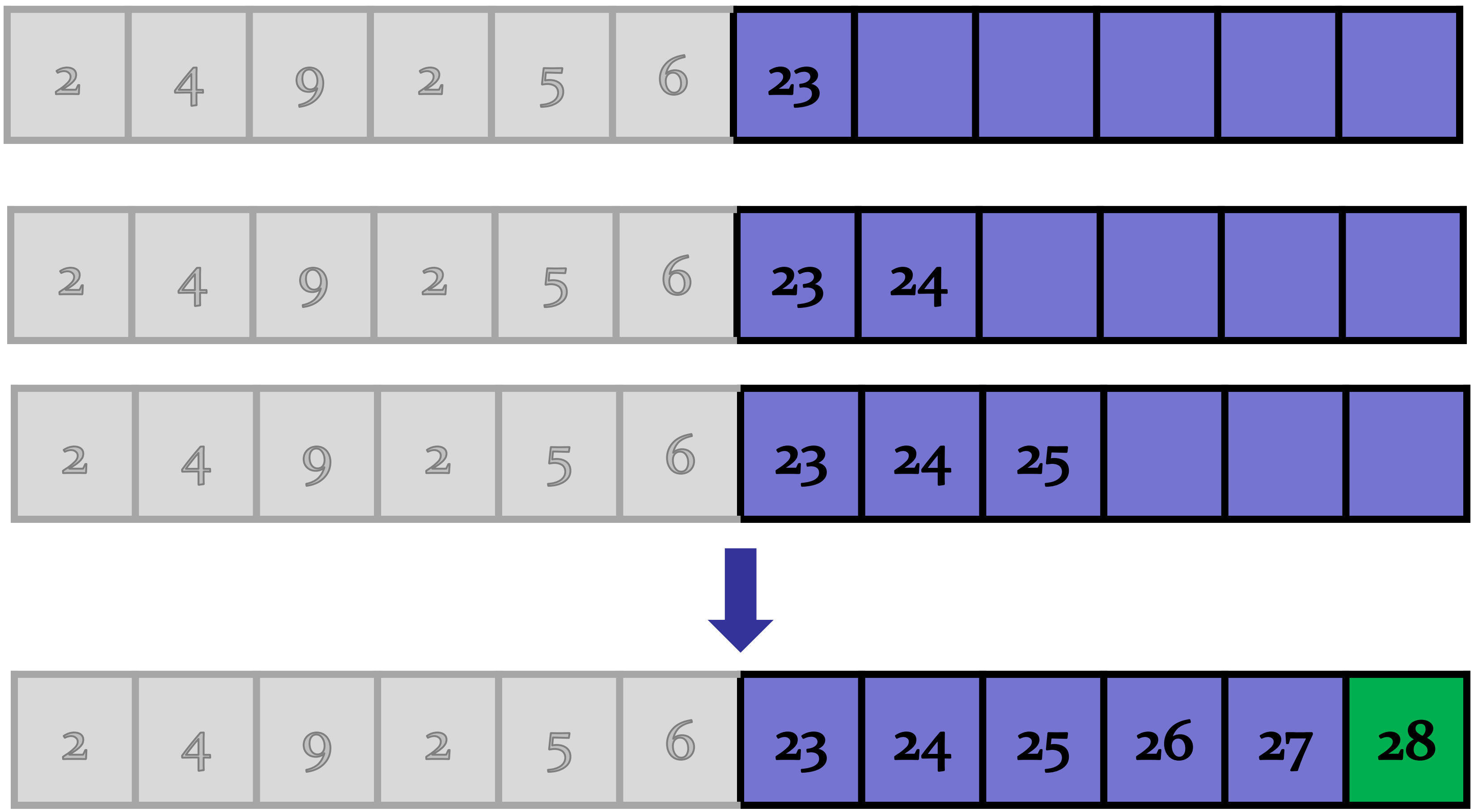
Explanation:

- 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-1] < A[n] ⟵ PEAK!!

Peak Finding: Algorithm 2

Recurse on right half, since 23 > 6.

Assume no peaks in right half.



Peak Finding: Algorithm 2

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-1] < A[n] ⟵ PEAK!!

Peak Finding: Algorithm 2

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] < j

Peak Finding: Algorithm 2

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] < j
- Base case: j = middle+1

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

Peak Finding: Algorithm 2

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] < 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 ➔ contradiction.

Peak Finding: Algorithm 2

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:

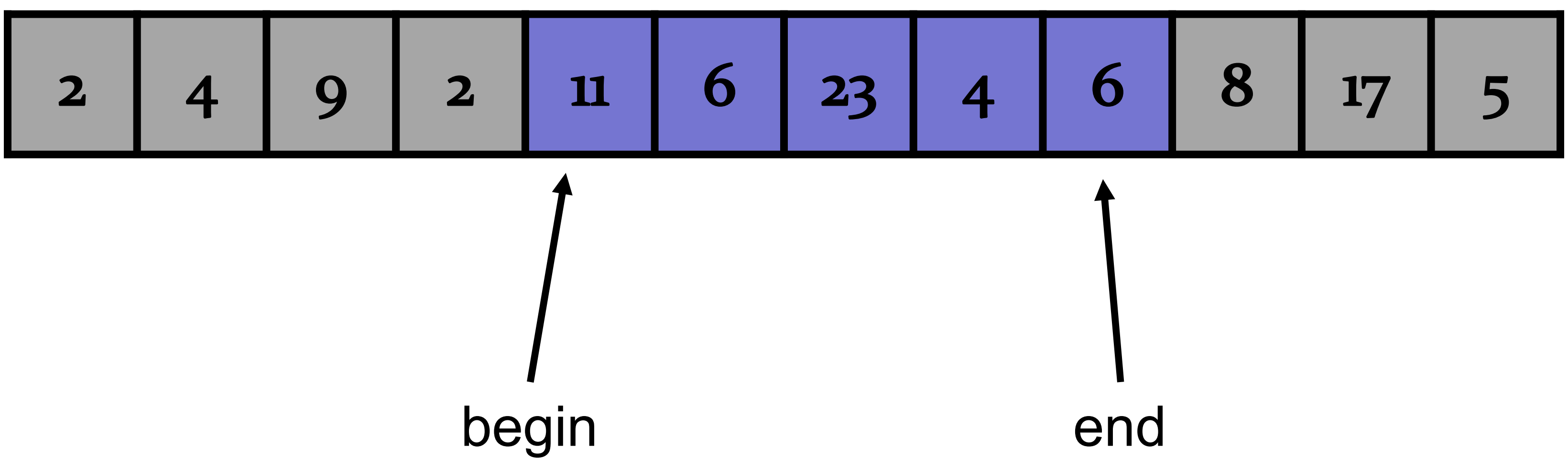
  For all (j > middle): A[j-1] < j
- Conclusion: A[n-2] < A[n-1]

  ➔ A[n-1] is a peak ➔ contradiction.

Key Invariants:

Correctness:

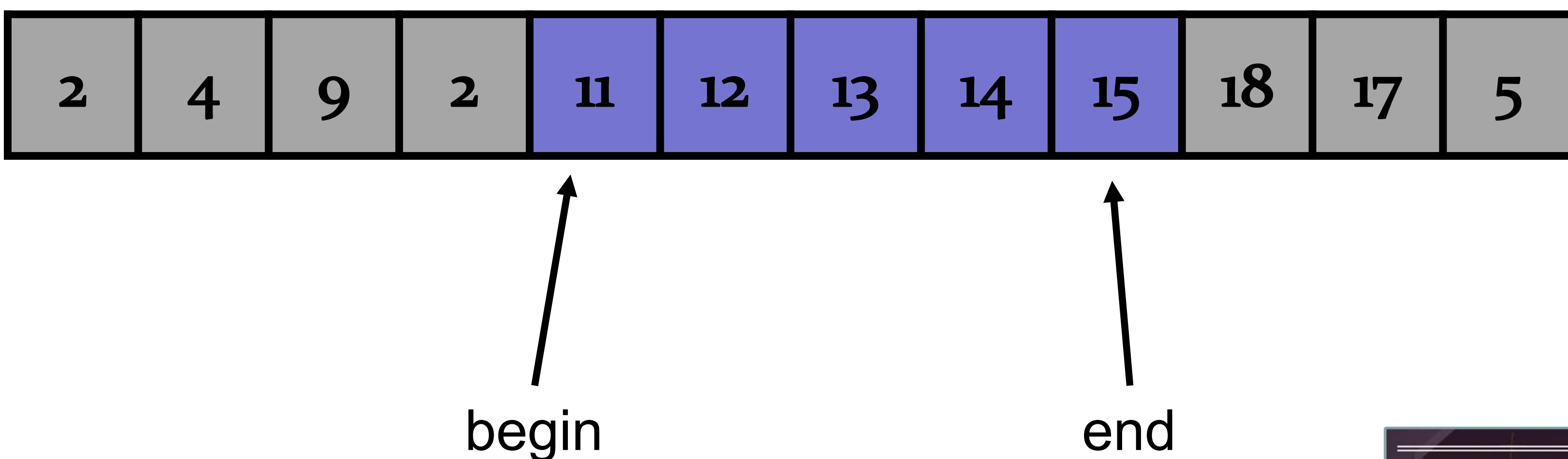
There exists a peak in the range [begin, end].



Key Invariants:

Is this good enough to prove the algorithm works?

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

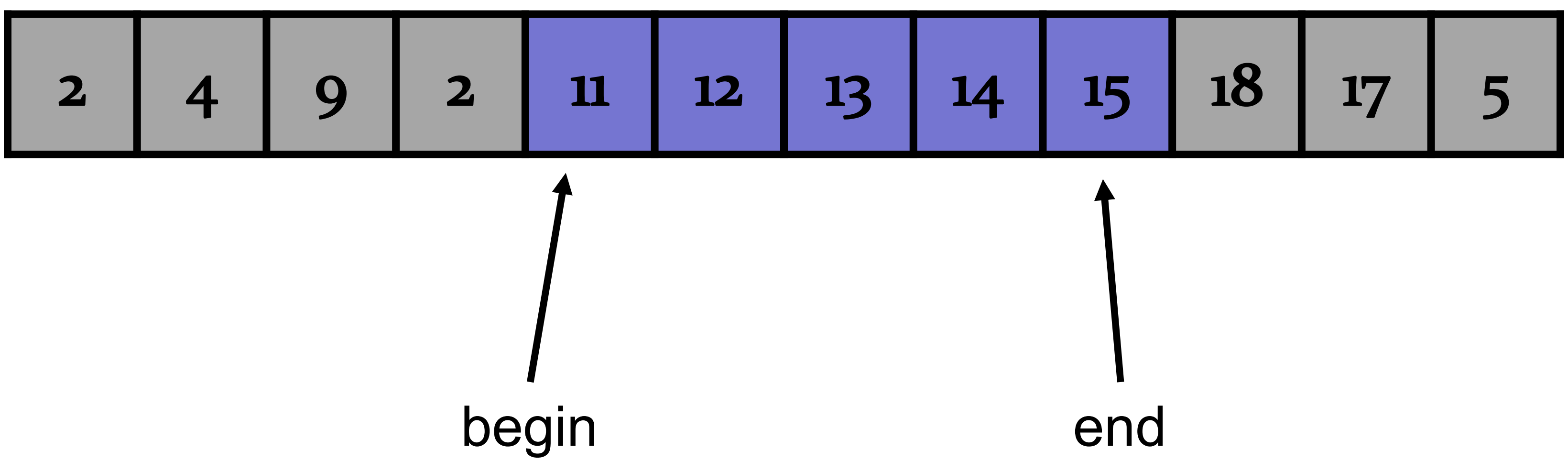


ARCHIPELAGO
is open

Key Invariants:

Not good enough to prove the algorithm works!

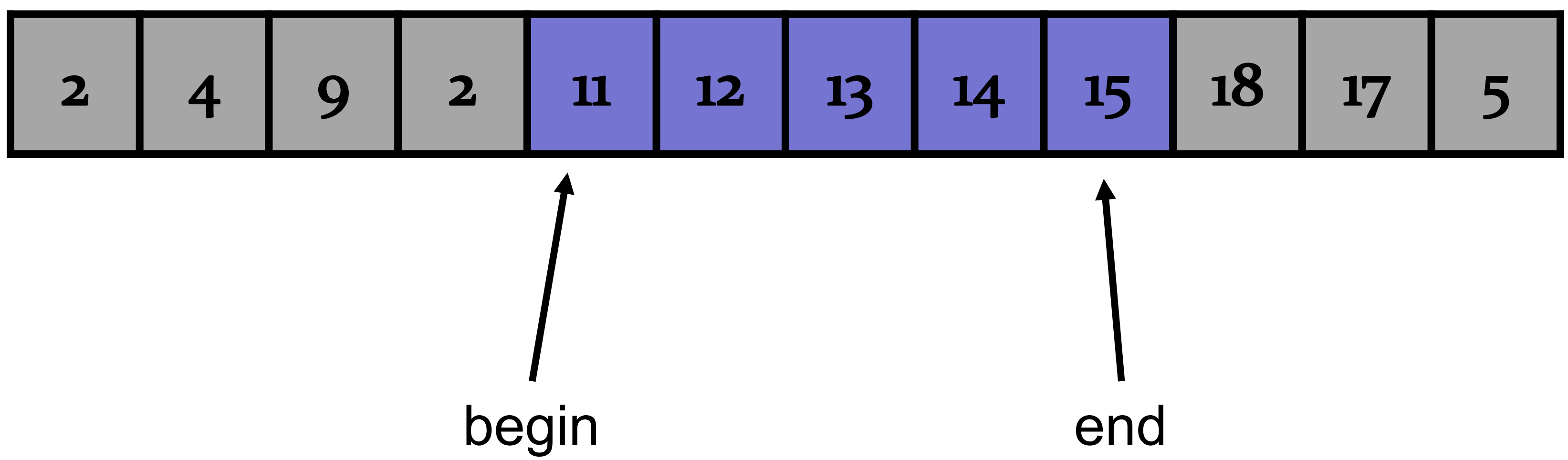
There exists a peak in the range [begin, end].



Key Invariants:

Not good enough to prove the algorithm works!

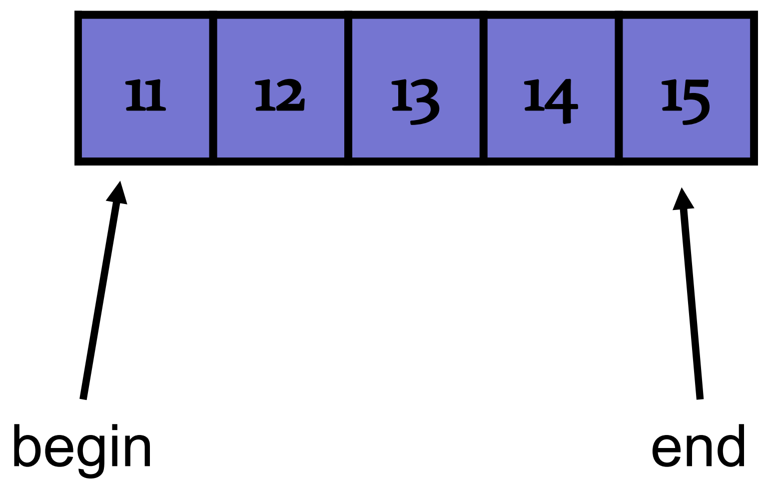
There exists a peak in the range [begin, end].



Key Invariants:

Not good enough to prove the algorithm works!

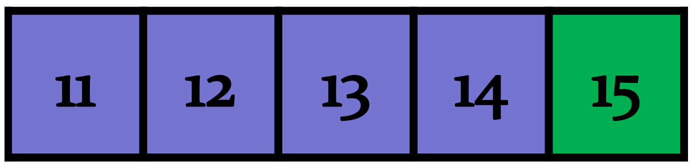
There exists a peak in the range [begin, end].



Key Invariants:

Not good enough to prove the algorithm works!

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



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 → 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 [1, n].

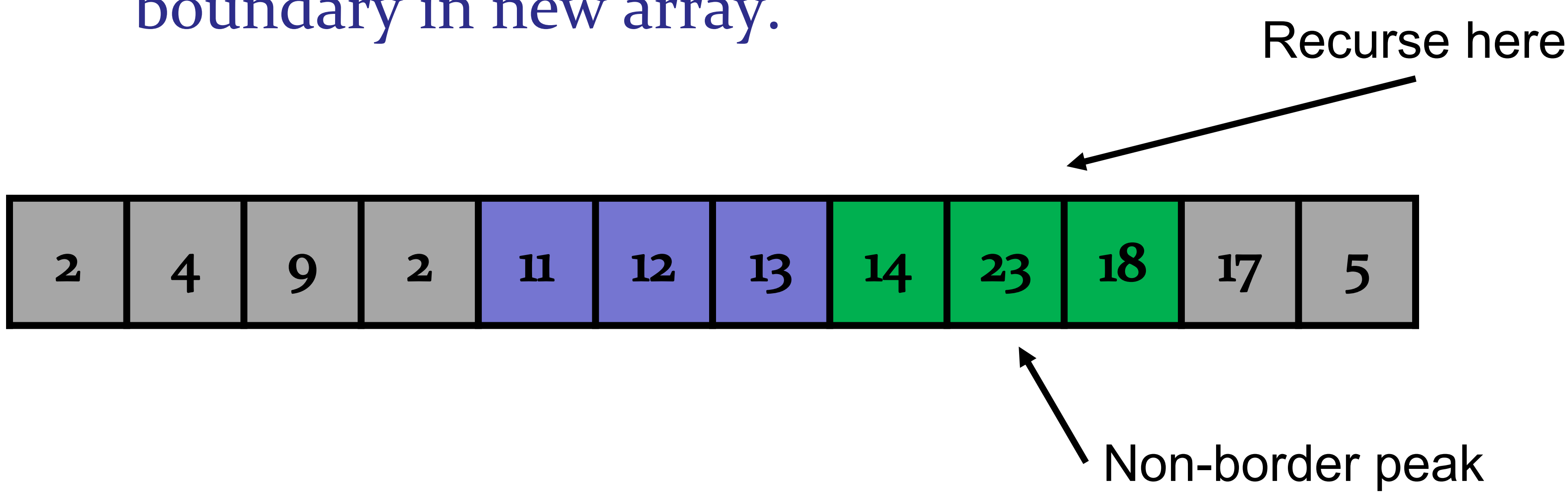
Peak Finding: Algorithm 2

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.



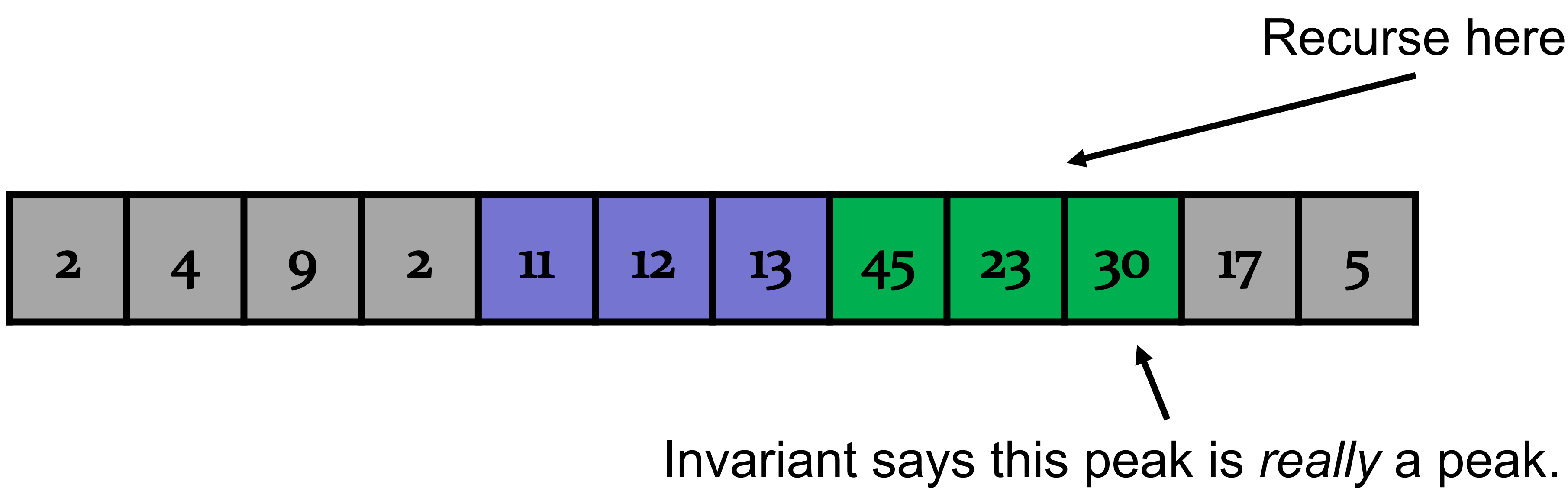
Peak Finding: Algorithm 2

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: Algorithm 2

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 [1, n].

Peak Finding: Algorithm 2

ARCHIPELAGO is open

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: Algorithm 2

Running time:

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

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

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)

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

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

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

$$\dots$$

$$\dots$$

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

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

Peak Finding: Algorithm 2

Unrolling the recurrence:

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

$$T(n) = T(n/2) + \theta(1)$$
$$= T(n/4) + \theta(1) + \theta(1)$$
$$= T(n/8) + \theta(1) + \theta(1) + \theta(1)$$
$$\cdots$$
$$\cdots$$
$$= T(1) + \theta(1) + \ldots + \theta(1) =$$
$$= \theta(1) + \theta(1) + \ldots + \theta(1) =$$

Number of times you can divide n by 2 until you reach 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 to find a peak in an array of size n

Recursion

Time for comparing A[n/2] with neighbors

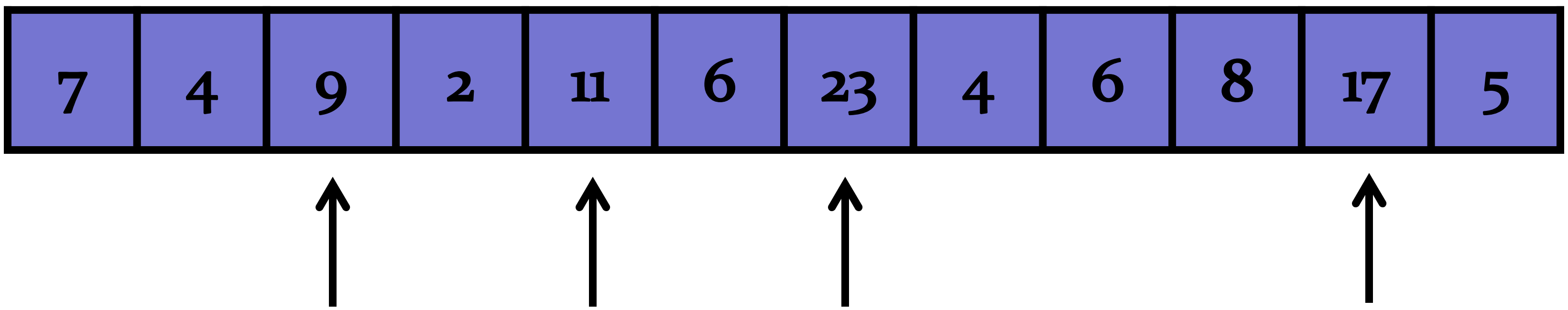
$$T(n) = T(n/2) + \theta(1)$$

Unrolling the recurrence:

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

Steep Peaks

Input: Some 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`

Steep Peaks

ARCHIPELAGO

is open

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

7	4	9	2	11	6	23	4	6	8	17	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

ARCHIPELAGO is open

Problematic example:

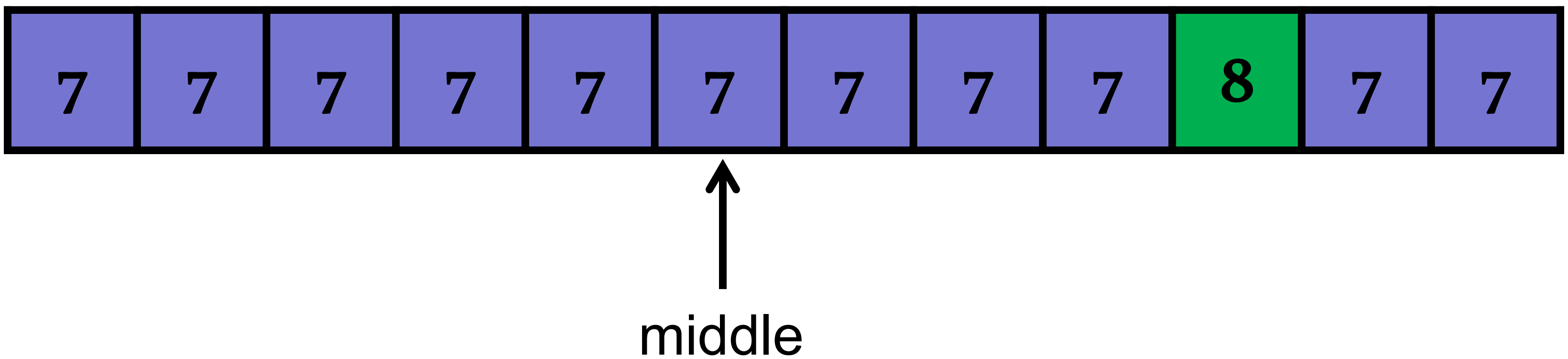
7	7	7	7	7	7	7	7	7	8	7	7

Inuitively:

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?

SLG1

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], n/2)

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

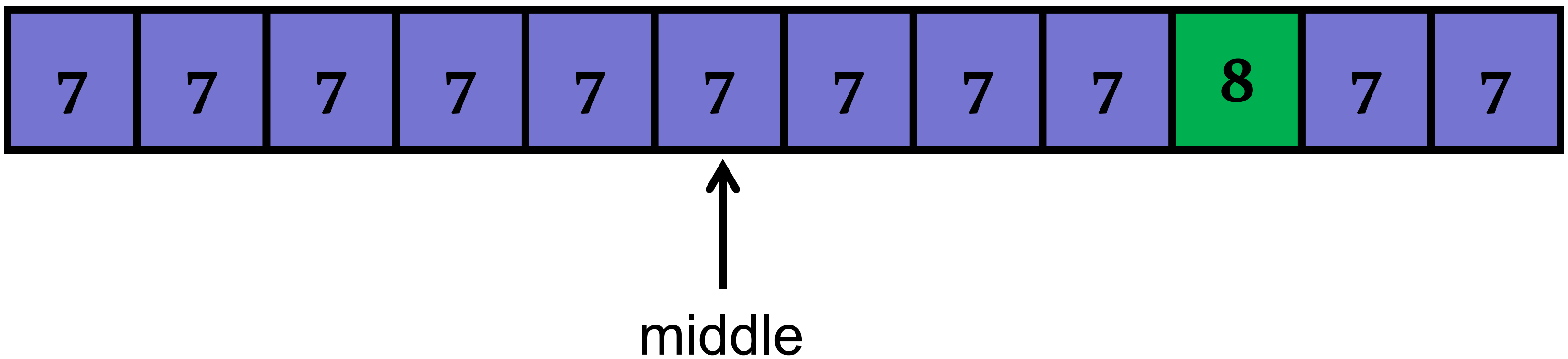
        FindPeak (A[1..n/2-1], n/2)

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

Steep Peaks

ARCHIPELAGO is open

Problematic example:



What happens if you recurse on both sides?

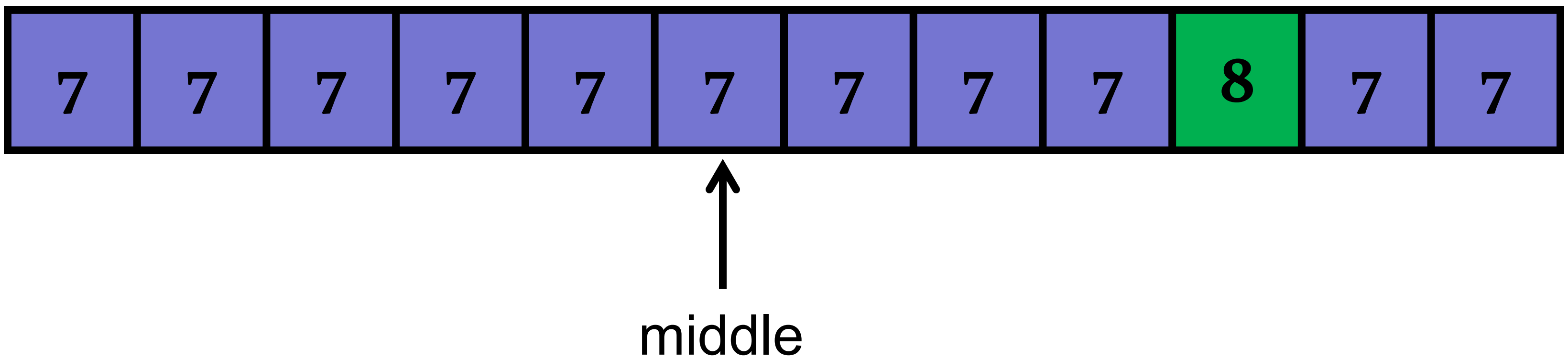
…

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

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

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

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

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

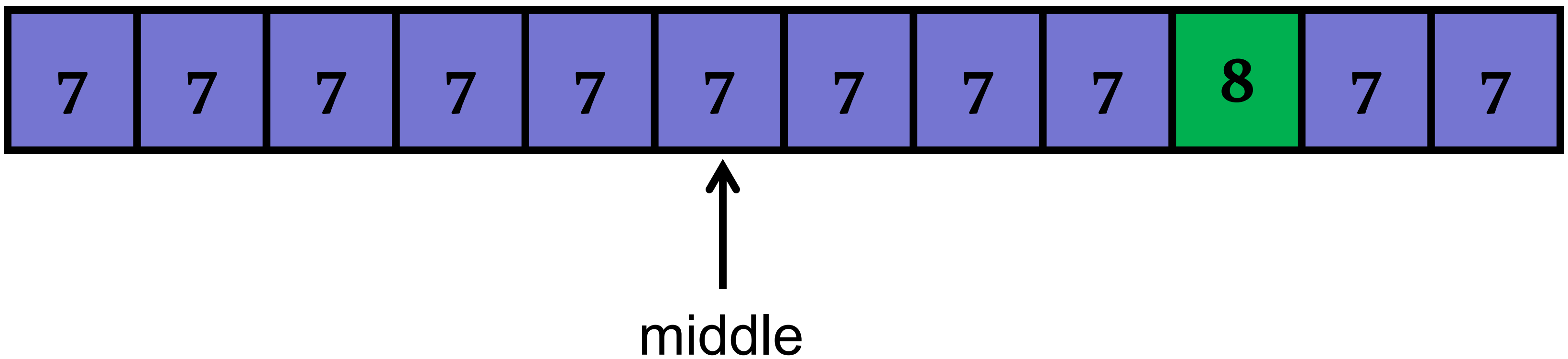
$$\cdots$$

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

$$= \theta(n)$$

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!