CS2020
Data Structures and Algorithms

Welcome!

Administrativia

- Today, come to Problem Sessions. If you have one assigned, come to that one. If you don't, come to any one.

- Discussion groups: still assignment in progress. If you are free for a Monday DG either 2-4 or 4-6, let me know.

- We will post problems to be discussed in DG next week today.

Today: Divide and Conquer!

Algorithm Analysis

- Big-O Notation
- Model of computation

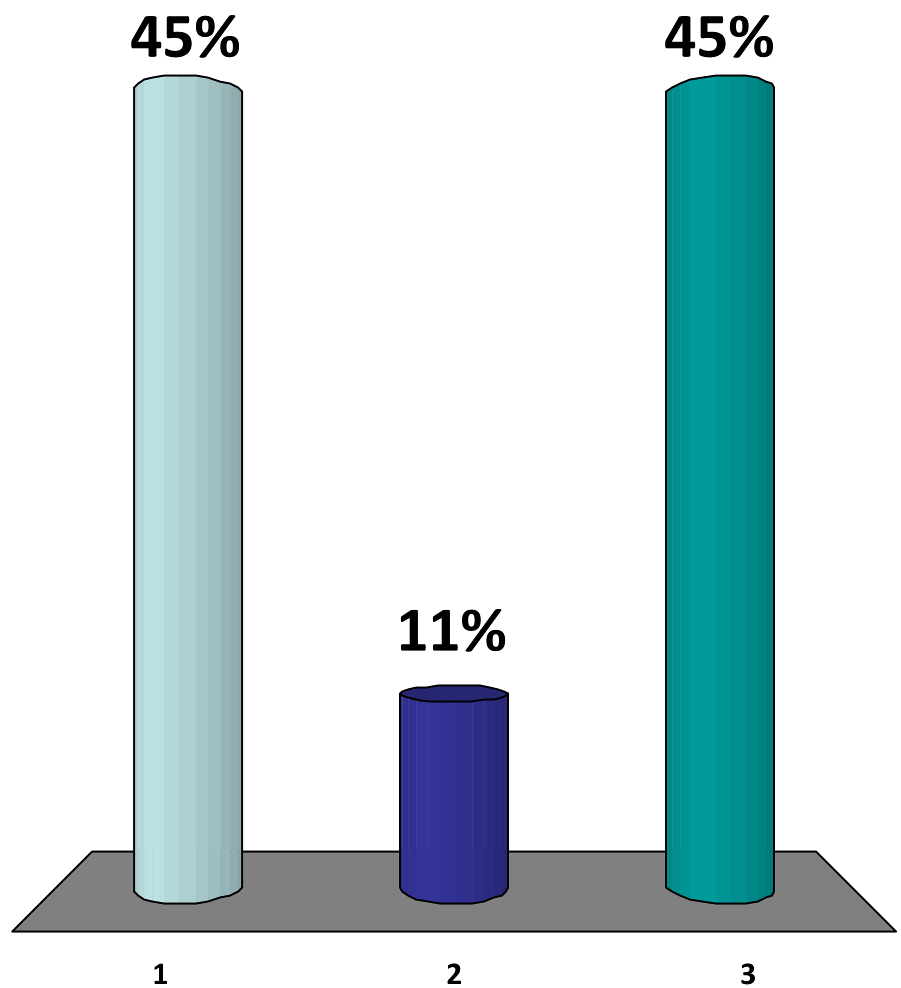
Searching

Peak Finding

- 1-dimension
- 2-dimensions

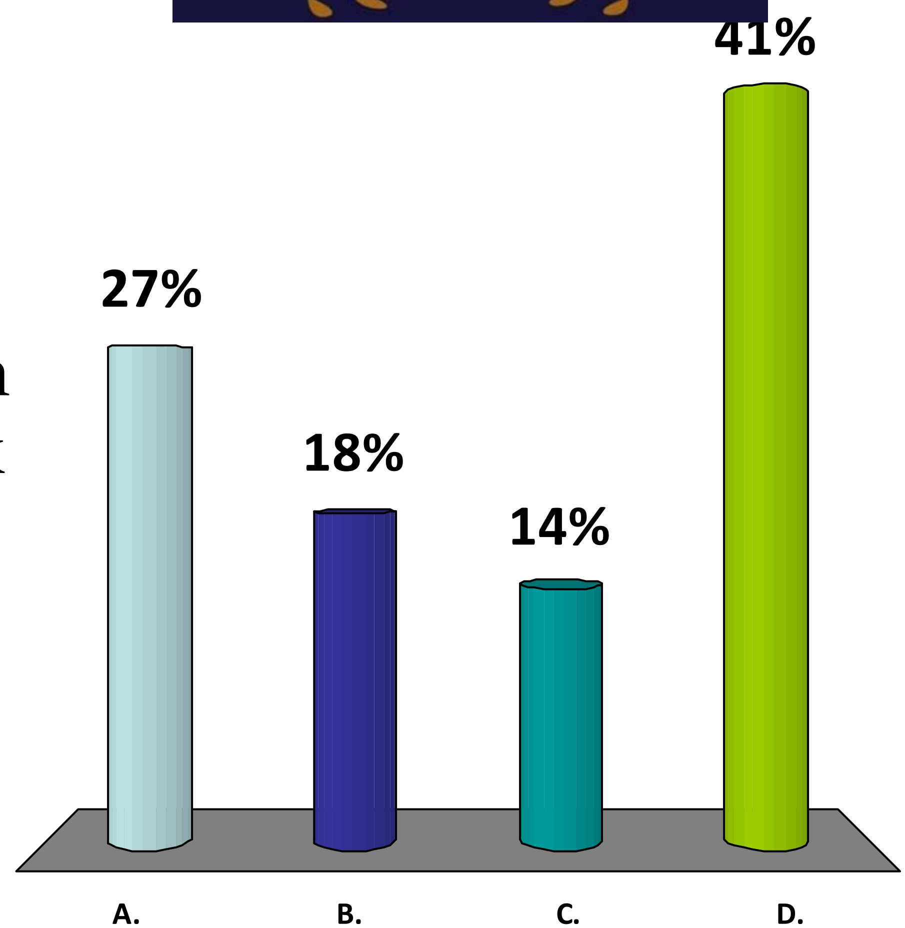
Did you remember your clicker?

1. Yes, I'm super-cool.
2. No, I'm lame.
3. Snakes?



Will you.....

A. Walk 10 min to a shop to buy one for $2
B. Walk 20 min to a supermarket and buy a dozen for $18
C. Drive 1 hour to a muffin company and ship back 200 muffins for $200
D. I don't like muffins. I prefer snakes.....



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

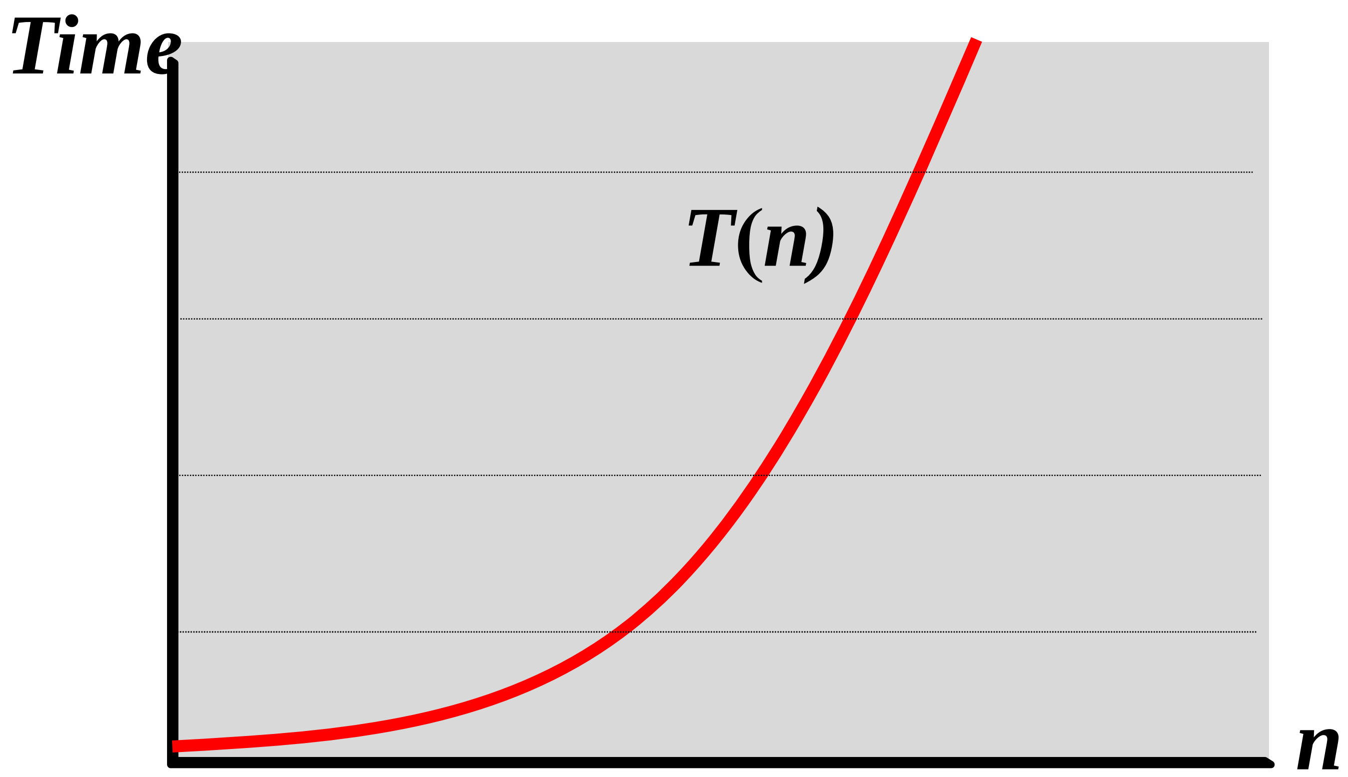
Which grows faster?

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

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: $T(\mathrm{n}) = O(f(n))$ if T grows no faster than f

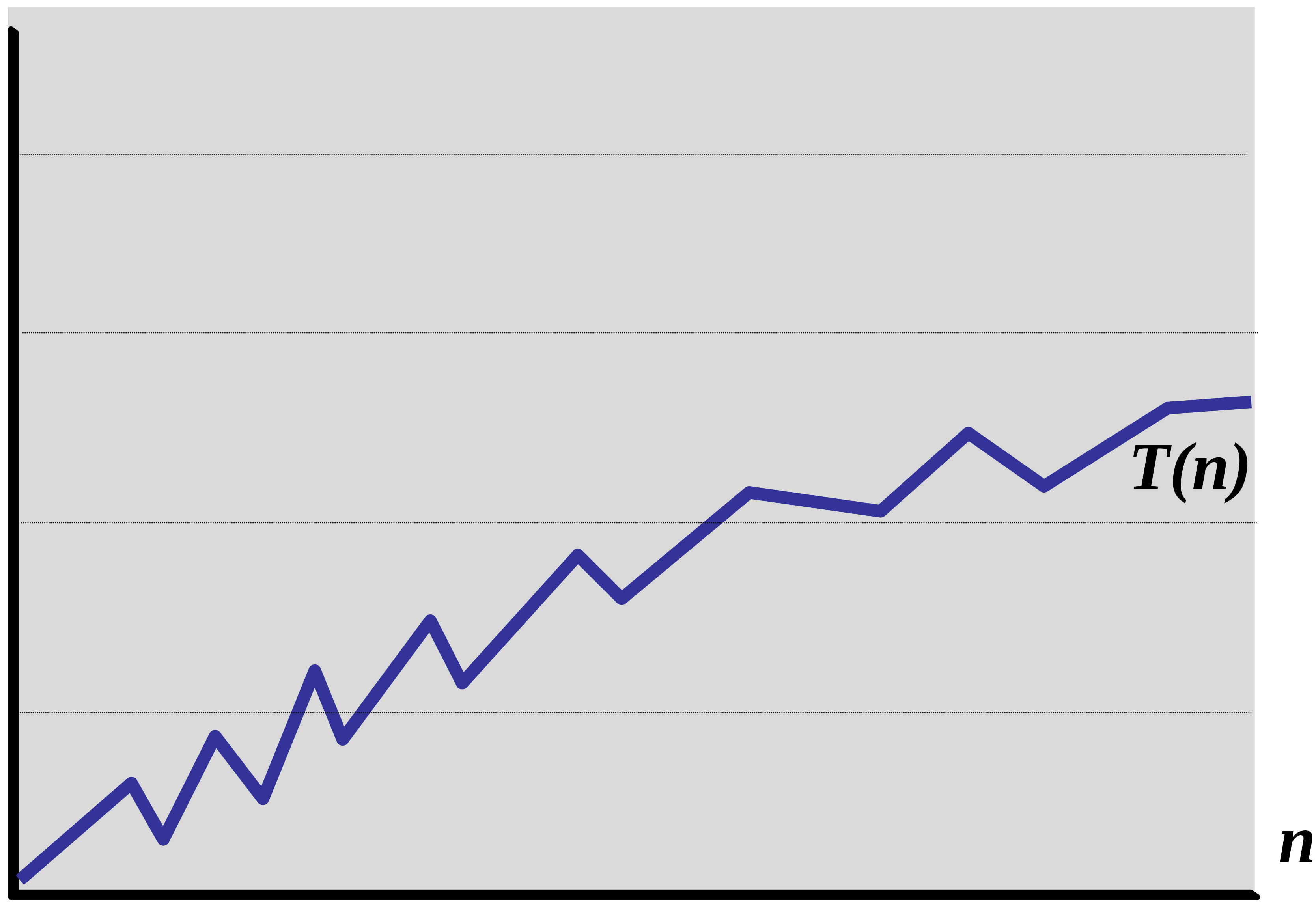
$\boldsymbol{T(n) = O(f(n))}$ if:

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

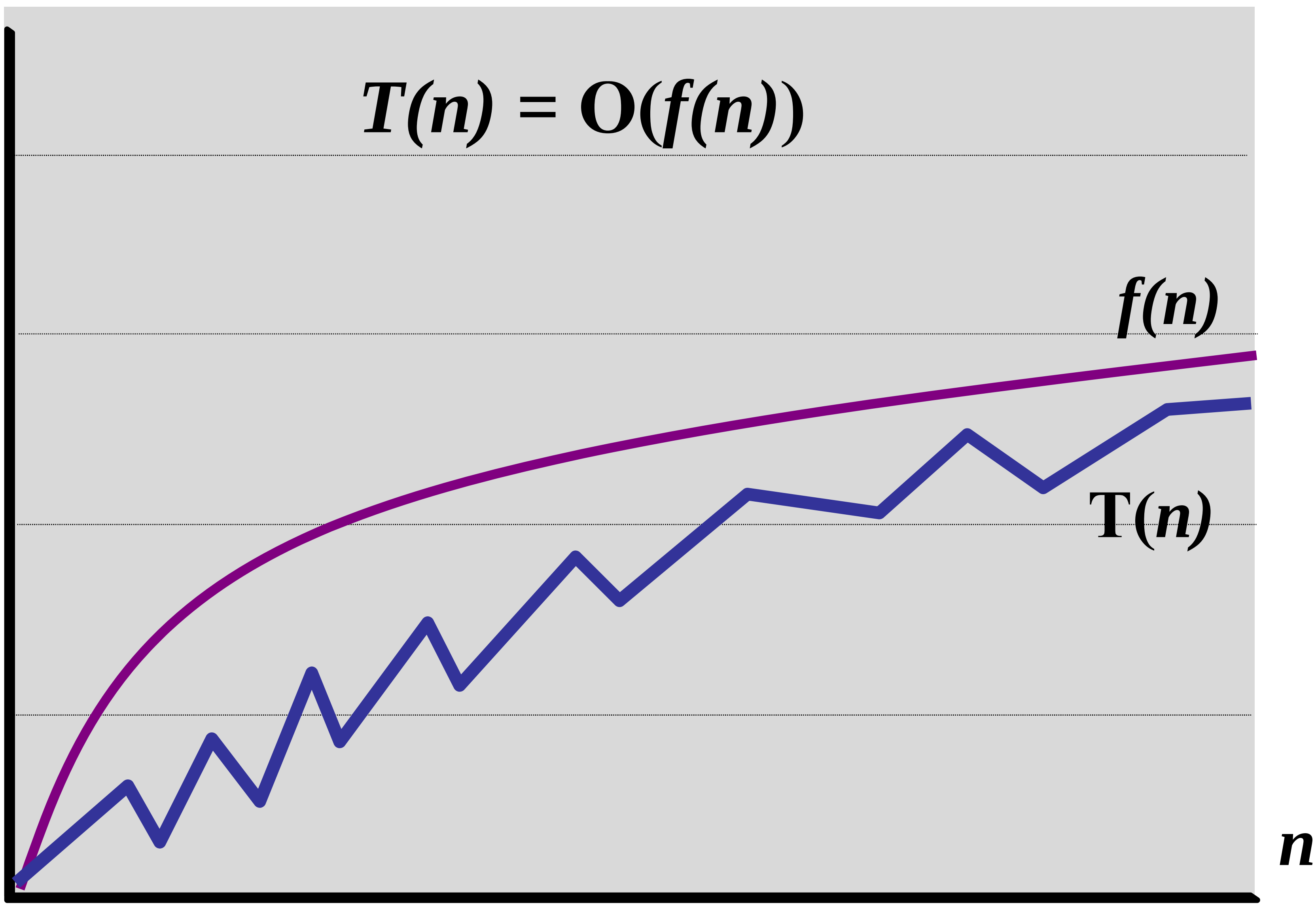
such that for all $n > n_0$:

$$\boldsymbol{T(n) \leq c\, f(n)}$$

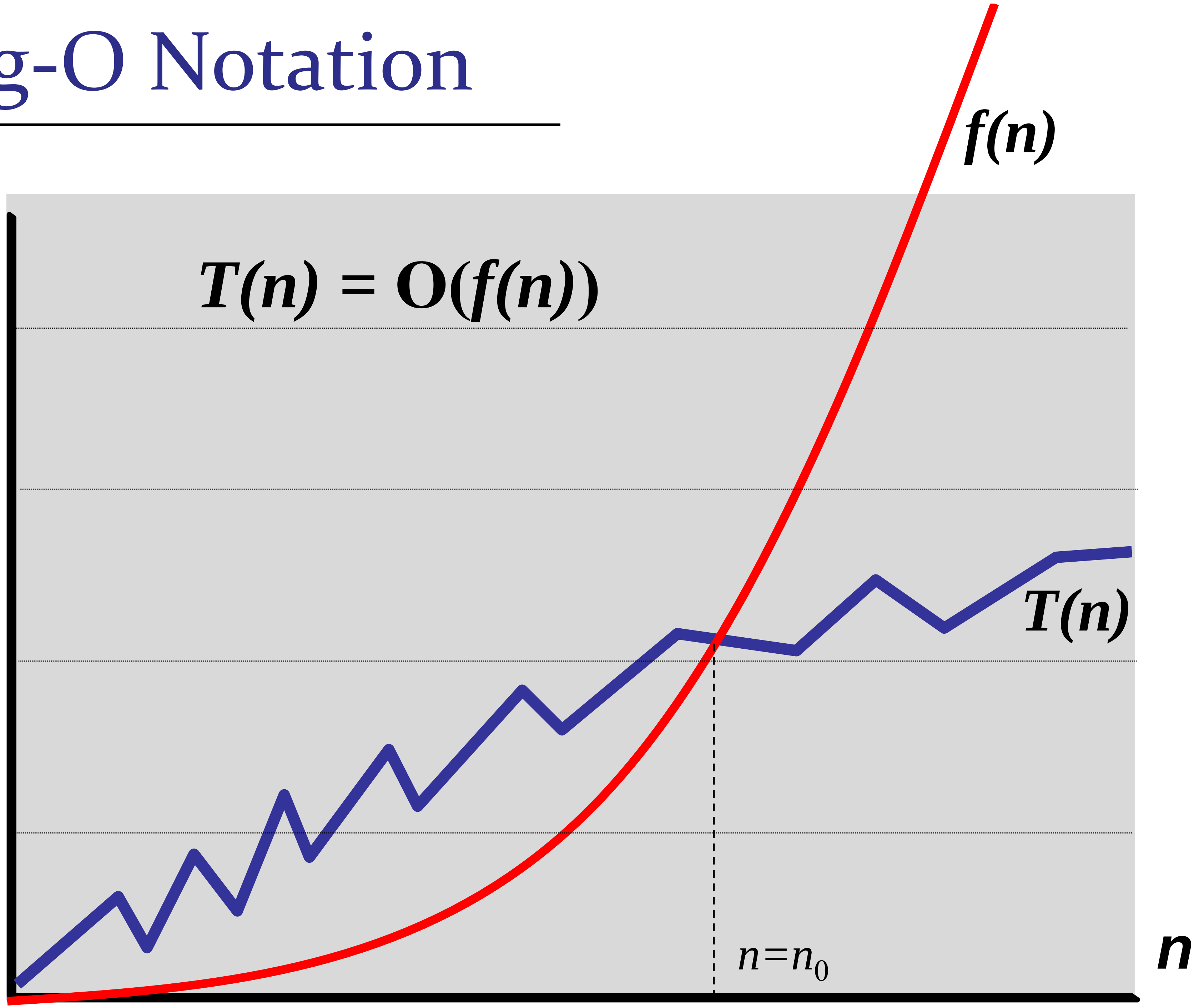
Big-O Notation



Big-O Notation



Big-O Notation



Big-O Notation

Example:

$$
\begin{aligned}
T(n) &= 4n^2 + 24n + 16 \\
&< 4n^2 + 24n^2 + n^2 \\
&= 29n^2 \quad (\text{for } n > n_0 = 4) \\
\\
&= O(n^2) \quad (\text{for } c = 29)
\end{aligned}
$$

Example

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

Not tight

Big-O Notation

Definition: $T(\mathrm{n}) = \mathrm{O}(f(n))$ if T grows no faster than f

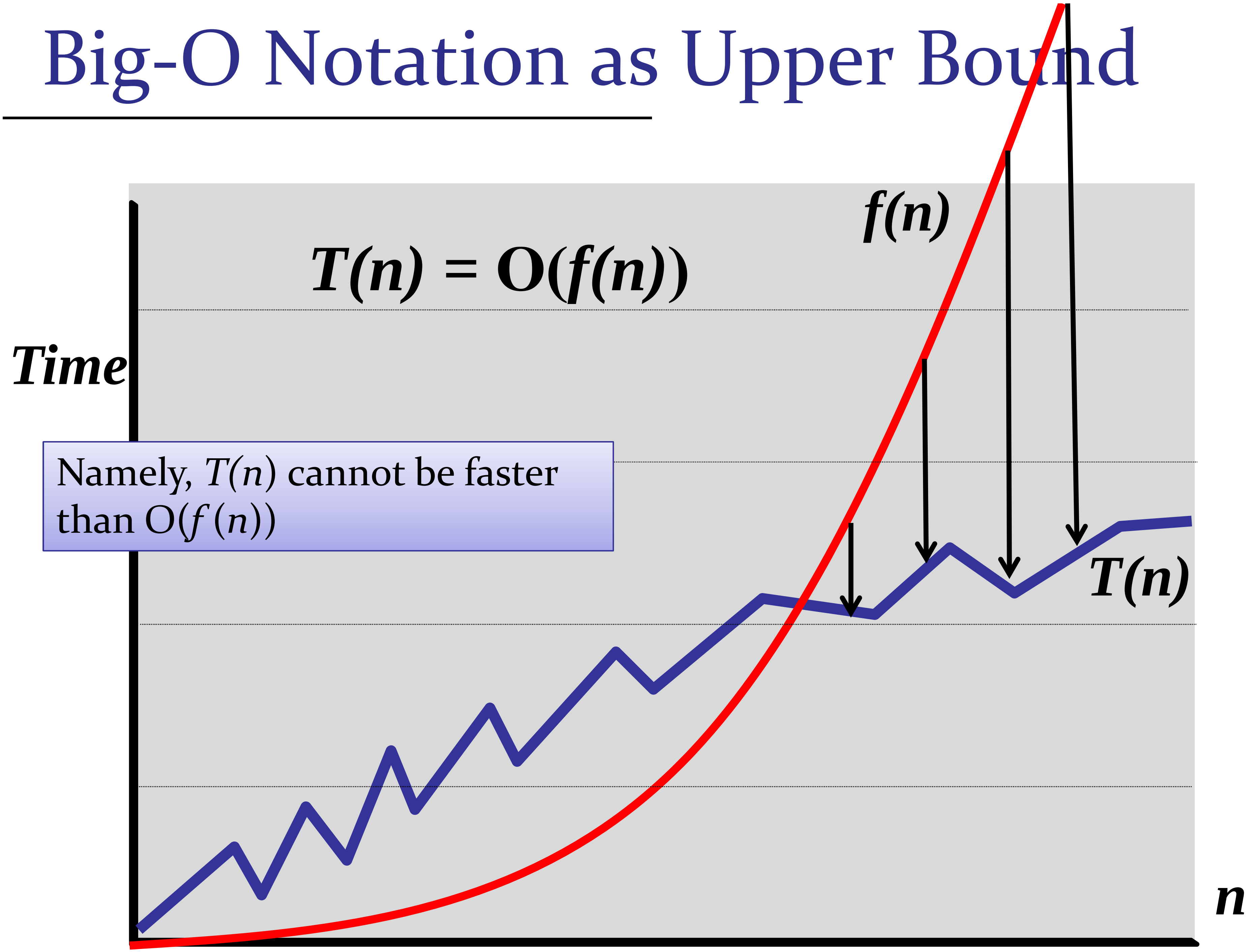
$\boldsymbol{T(n) = \mathrm{O}(f(n))}$ if:

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

such that for all $n > n_0$:

$$\boldsymbol{T(n) \leq c\, f(n)}$$

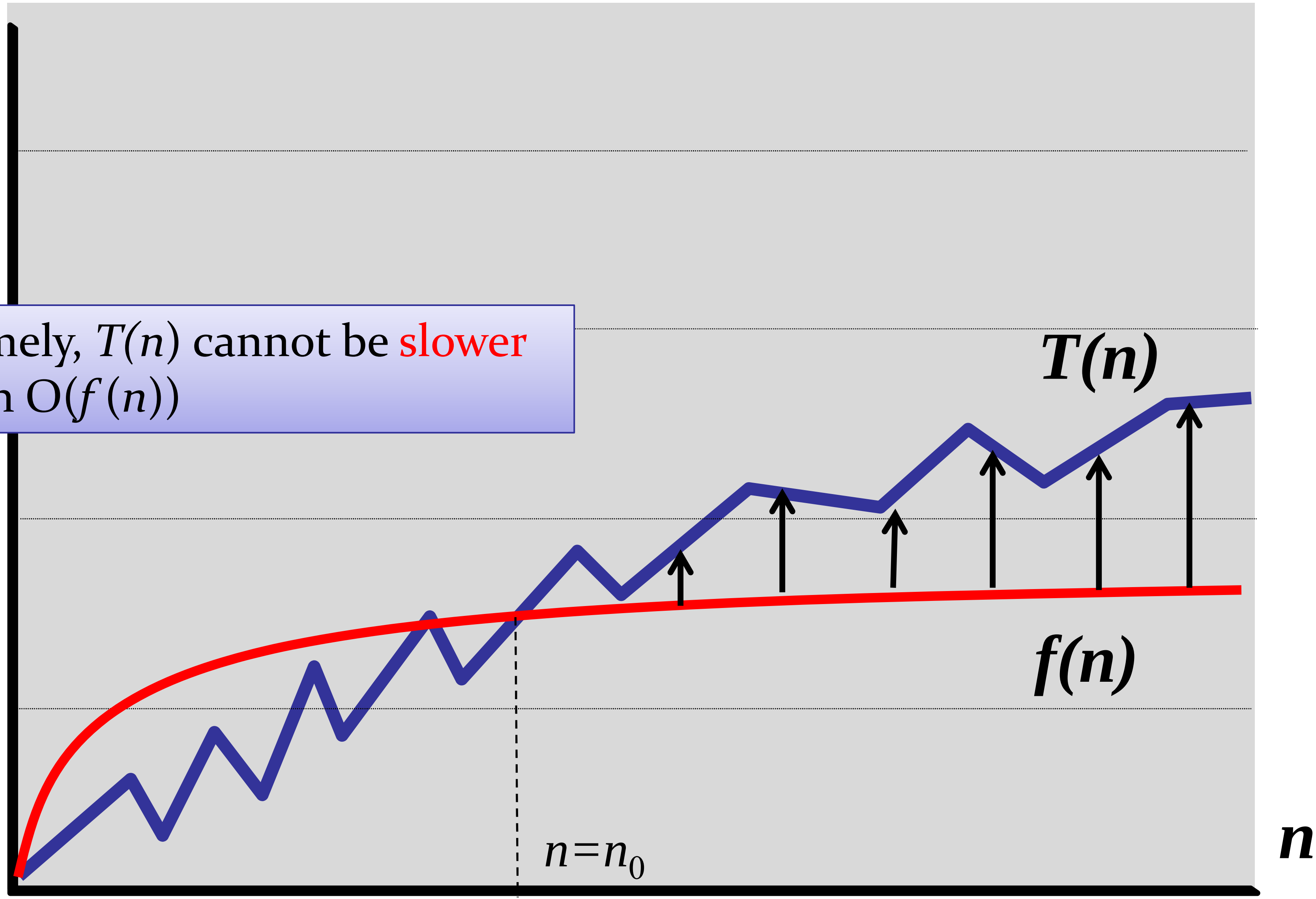
Big-O Notation as Upper Bound



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

How about Lower bound?

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



Big-O Notation

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

$\boldsymbol{T(n) = \Omega(f(n))}$ if:

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

such that for all $n > n_0$:

$\boldsymbol{T(n) \geq c\, f(n)}$

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.

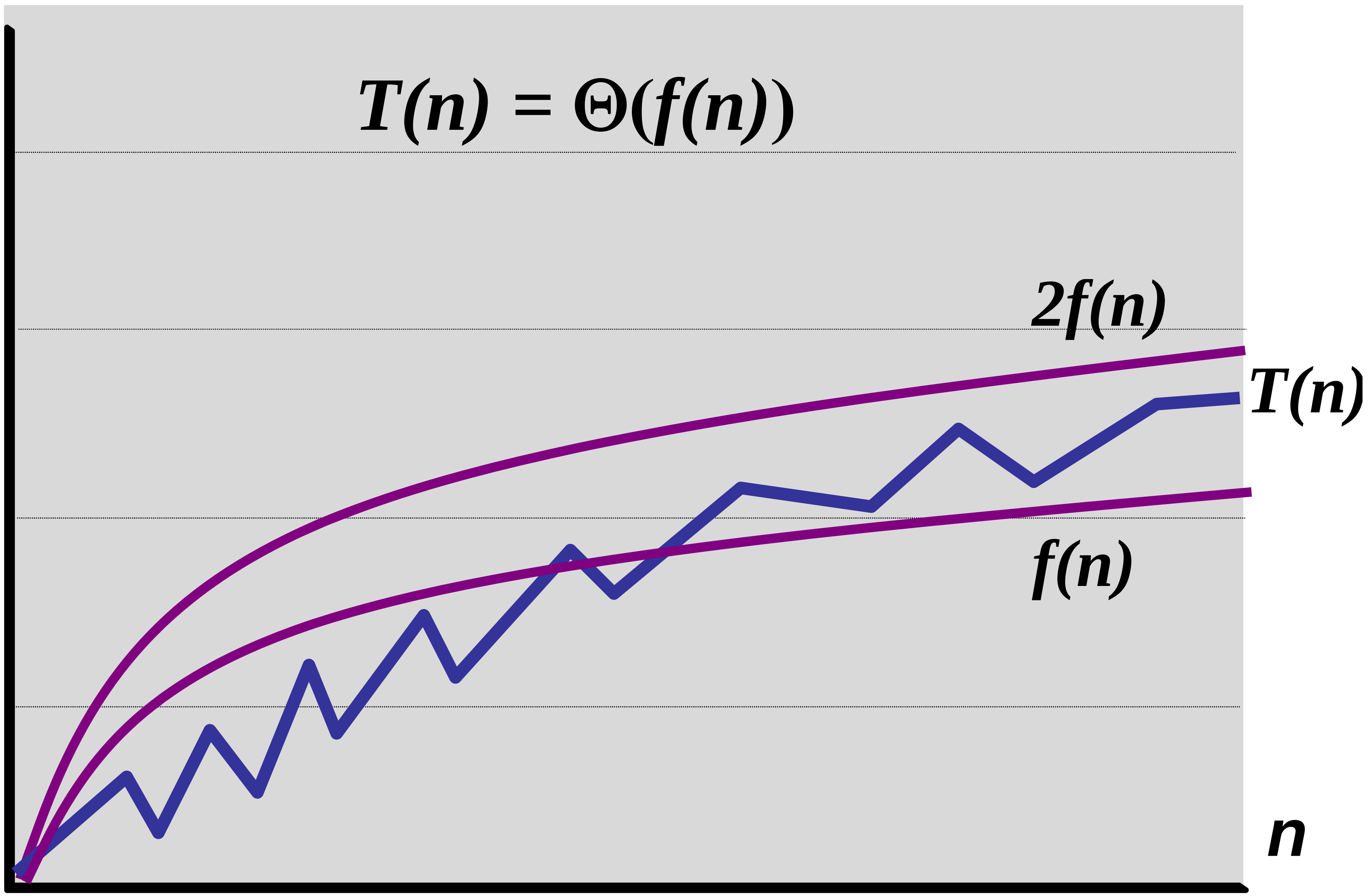
Big-O Notation

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

$\boldsymbol{T(n) = \Theta(f(n))}$ 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:

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

Example:

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

Big-O Notation

Rules:

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

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

Example:

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

$$5n = O(n)$$

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

Big-O Notation

Rules:

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

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

Example:

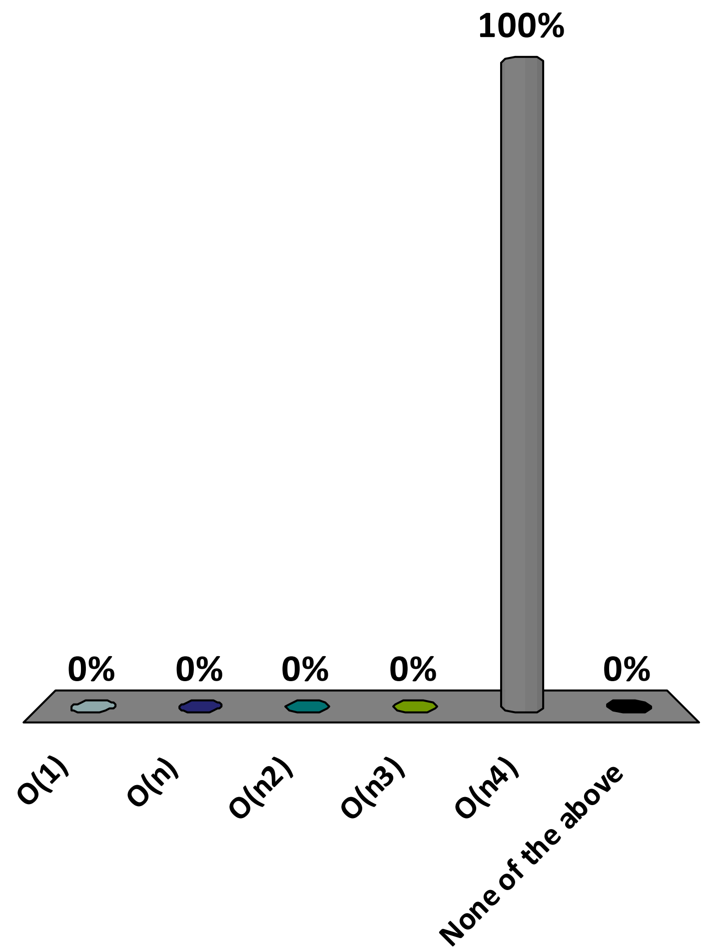
$10n^2 = O(n^2)$

$5n = O(n)$

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

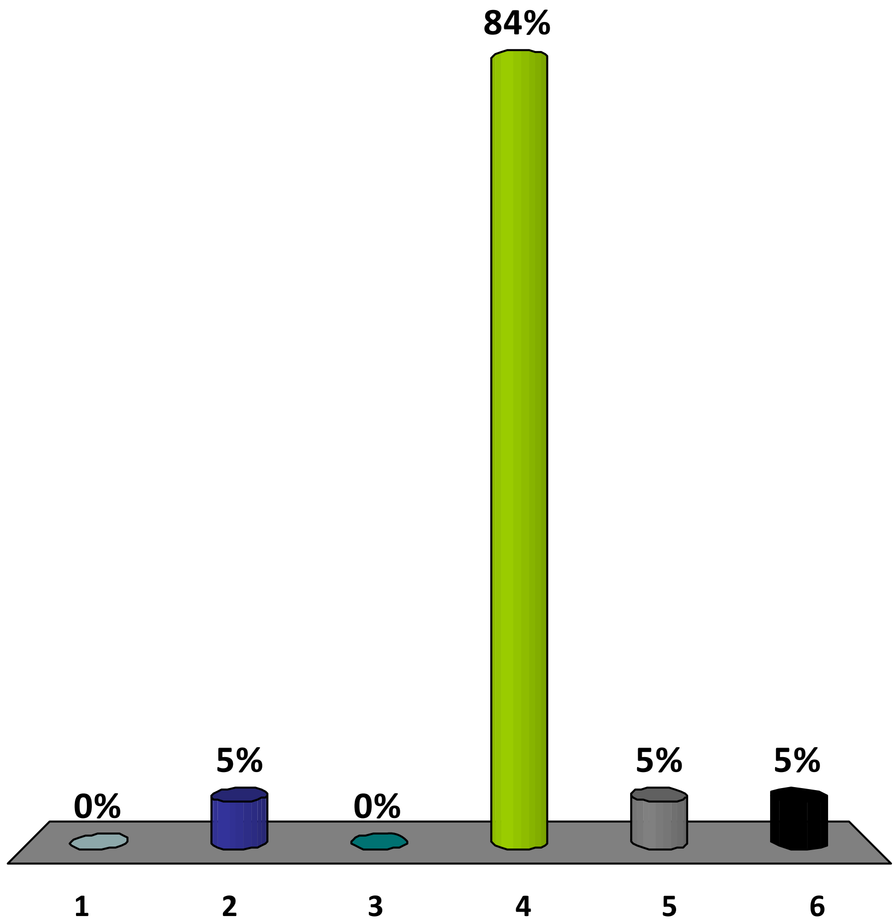
$n^4 + 3n^2 + n^2 + 17 = ?$

A. $O(1)$
B. $O(n)$
C. $O(n^2)$
D. $O(n^3)$
E. $O(n^4)$
F. None of the above



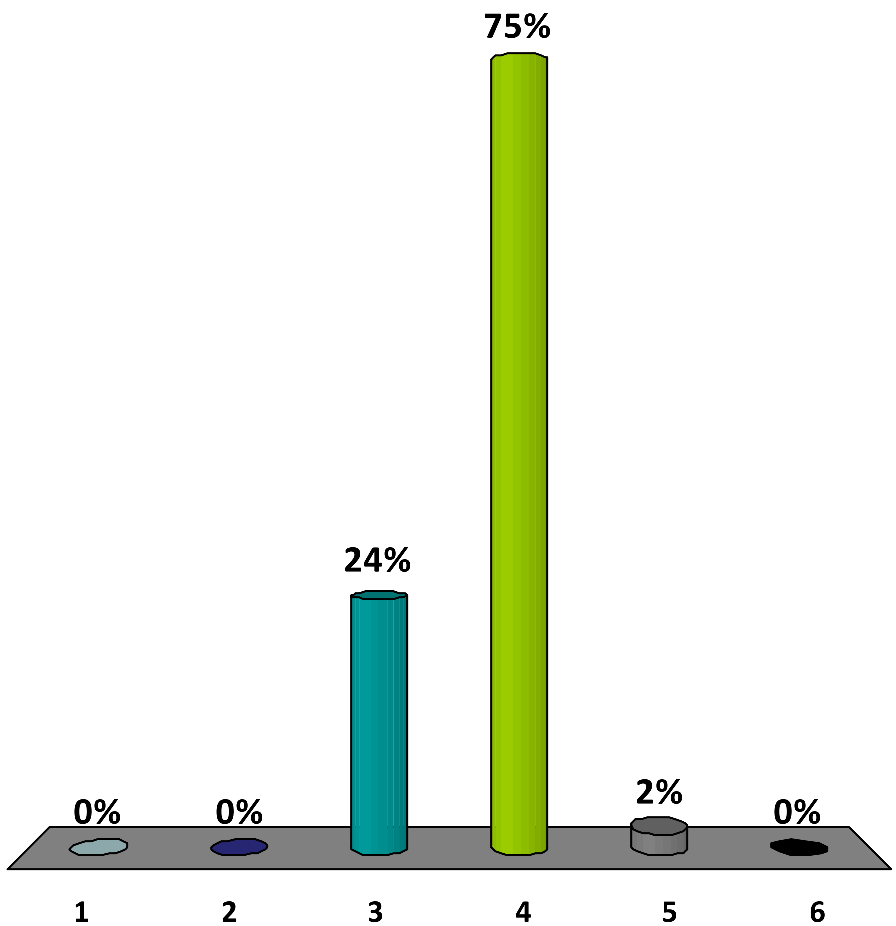
$4n^2\log(n) + 8n + 16 = ?$

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



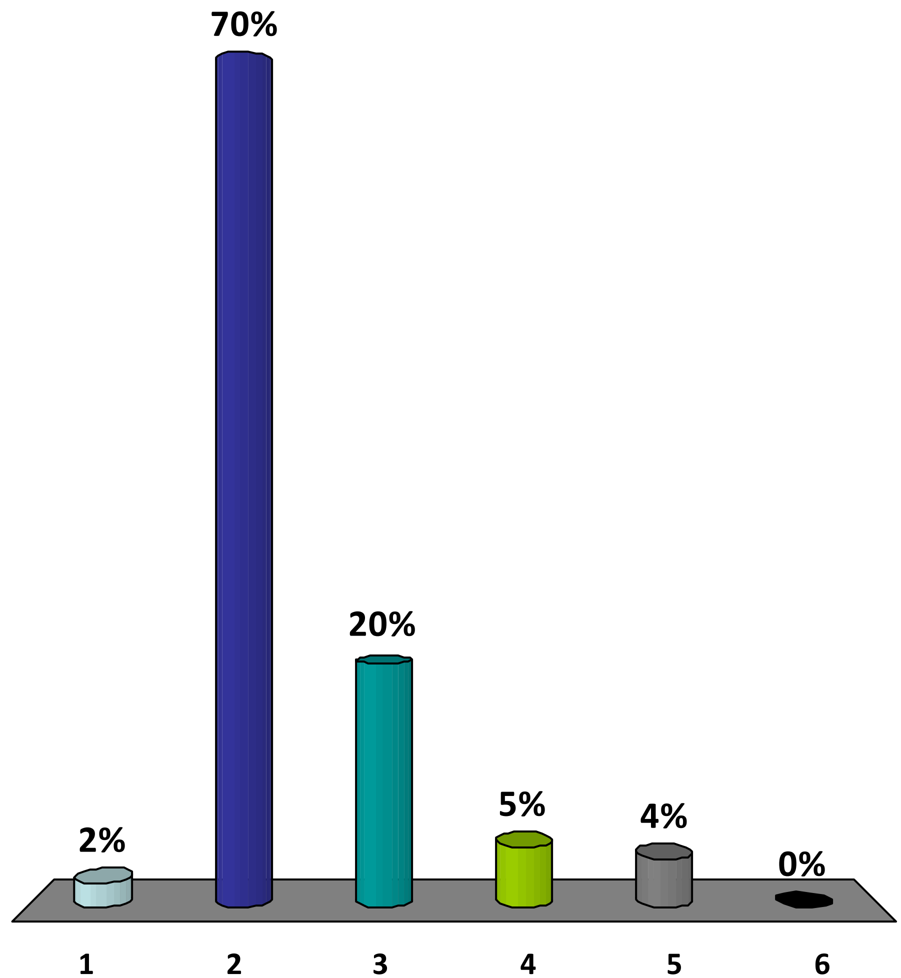
$$2^{2n} + 2^n + 2 =$$

1. O(n)
2. $O(n^6)$
3. $O(2^n)$
4. $\boxed{O(2^{2n})}$
5. $O(n^n)$
6. Still confused…



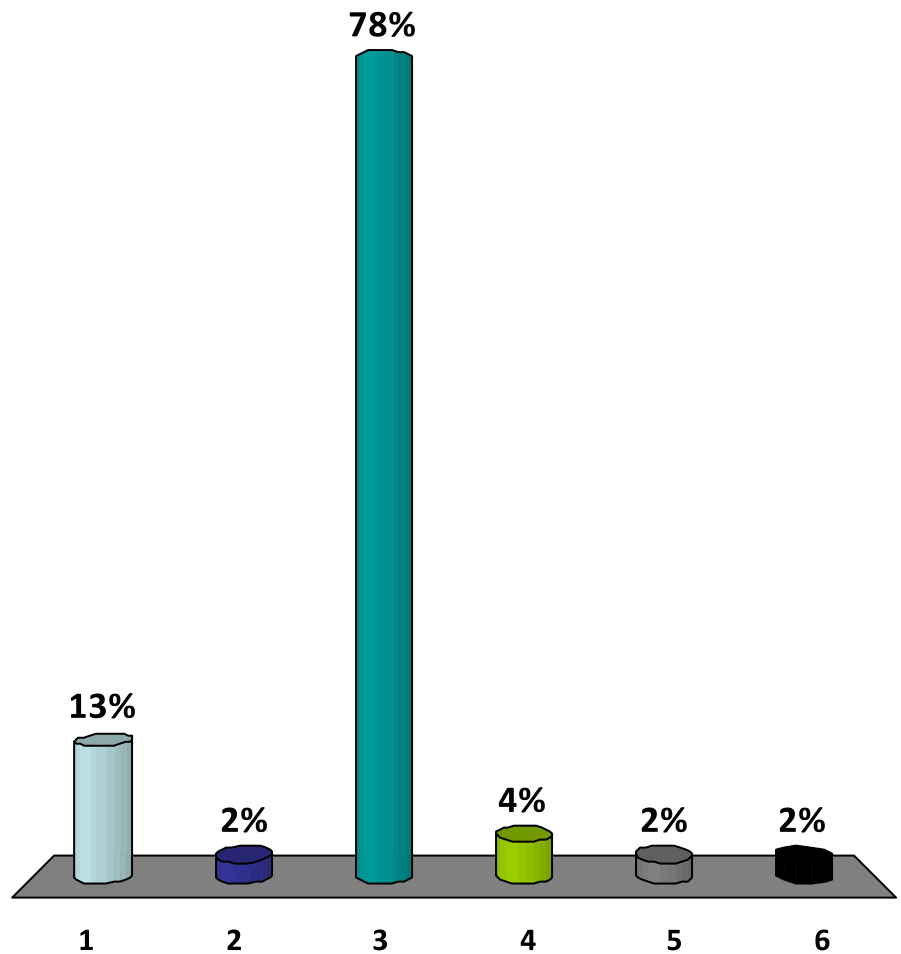
$\log(8n^2 + 4n) =$

1. $O(1)$
2. $\boxed{O(\log n)}$
3. $O(\log^2 n)$
4. $O(n)$
5. $O(n^2)$
6. Still confused…



$\log(n!) =$

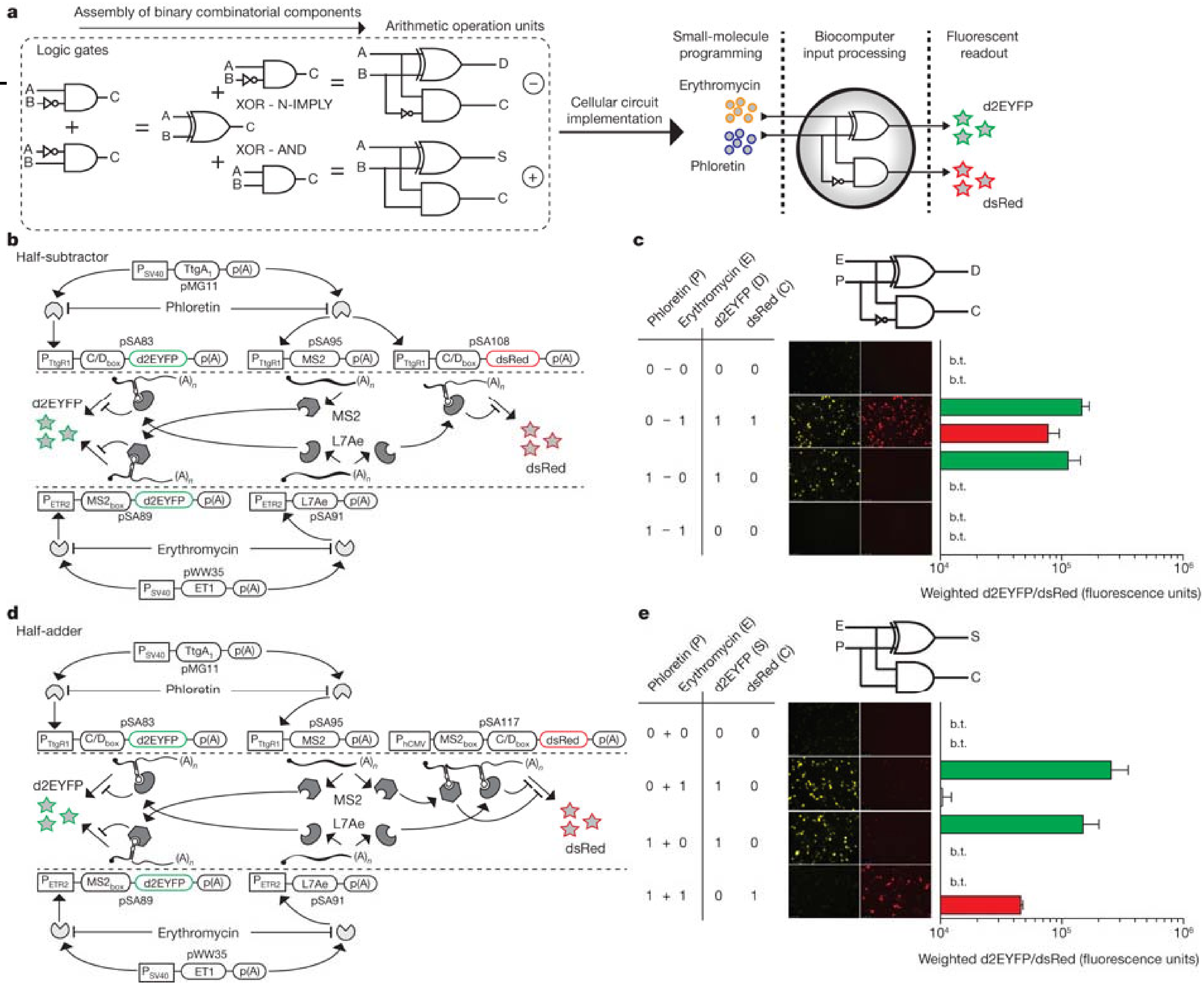
1. $O(\log n)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(2^n)$
6. Still confused…



Model of Computation?

What are the different types of “computations” or different types “of computers”

- Sequential vs Parallel
- Deterministic vs Probabilistic
- E.g. Biocomputers


a
Assembly of binary combinatorial components
Arithmetic operation units
Logic gates
XOR - N-IMPLY
XOR - AND
Cellular circuit implementation
Small-molecule programming
Biocomputer input processing
Fluorescent readout
Erythromycin
Phloretin
d2EYFP
dsRed
b
Half-subtractor
pMG11
pSA83
pSA95
pSA108
pSA89
pSA91
pWW35
MS2
L7Ae
c
Phloretin (P)
Erythromycin (E)
d2EYFP (D)
dsRed (C)
0 − 0 0 0
0 − 1 1 1
1 − 0 1 0
1 − 1 0 0
b.t.
Weighted d2EYFP/dsRed (fluorescence units)
d
Half-adder
pSA117
e
d2EYFP (S)
0 + 0 0 0
0 + 1 1 0
1 + 0 1 0
1 + 1 0 1

Model of Computation

Sequential Computer

- One thing at a time

- All operations take constant time

  Addition, subtraction, multiplication, comparison

Algorithm Analysis

Example:

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

1 assignment

1 assignment
k+1 comparisons
k increments

k array access
k addition
k assignment

1 return

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

Algorithm Analysis

Example:

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

Not 1!
Not constant!
Not k!

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;
}
```

Recurrences

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

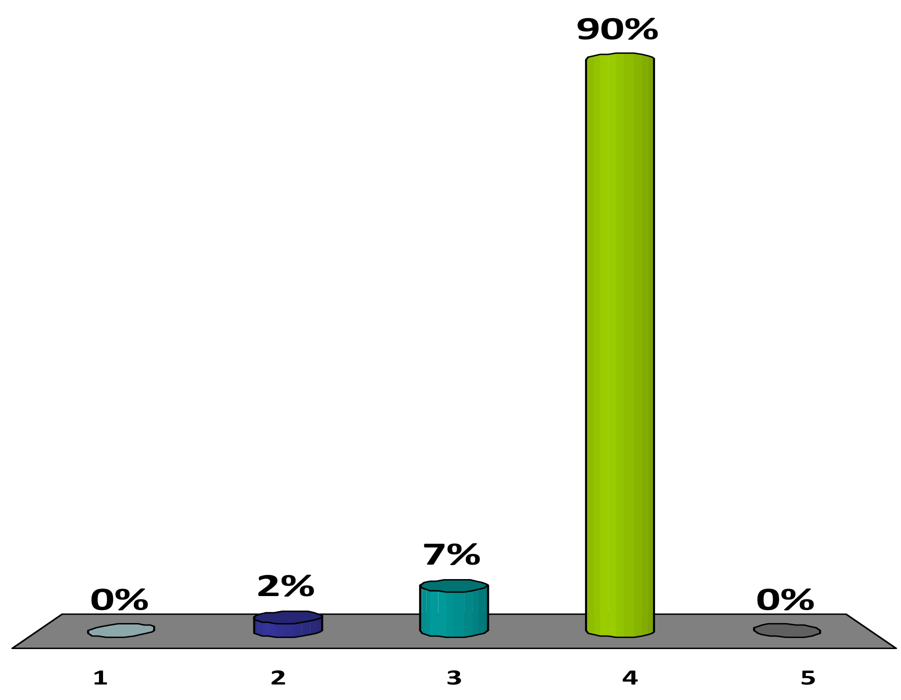
$T(n\text{-}1)$ $T(n\text{-}1)$

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

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. $\boxed{O(n^2)}$
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]`

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

Search for k in array A.

Binary Search

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

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

Search for 17 in array A.

Binary Search

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

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

Search for 17 in array A.

- Find middle element: 7

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Binary Search

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

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

Search for 17 in array A.

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

Problem Solving: Reduce the Problem

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

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

Reduce-and-Conquer:

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

Binary Search

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

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

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

Binary Search

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

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

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

Does not terminate

Round down?

A[begin] == key?

Binary Search

Specification:

- Finds element if it is in the array.
- Returns “NO” if it is not in the array

Binary Search

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

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

```
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]
```

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]`

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

```
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

Sorted array: A[1..n]

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

```
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

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

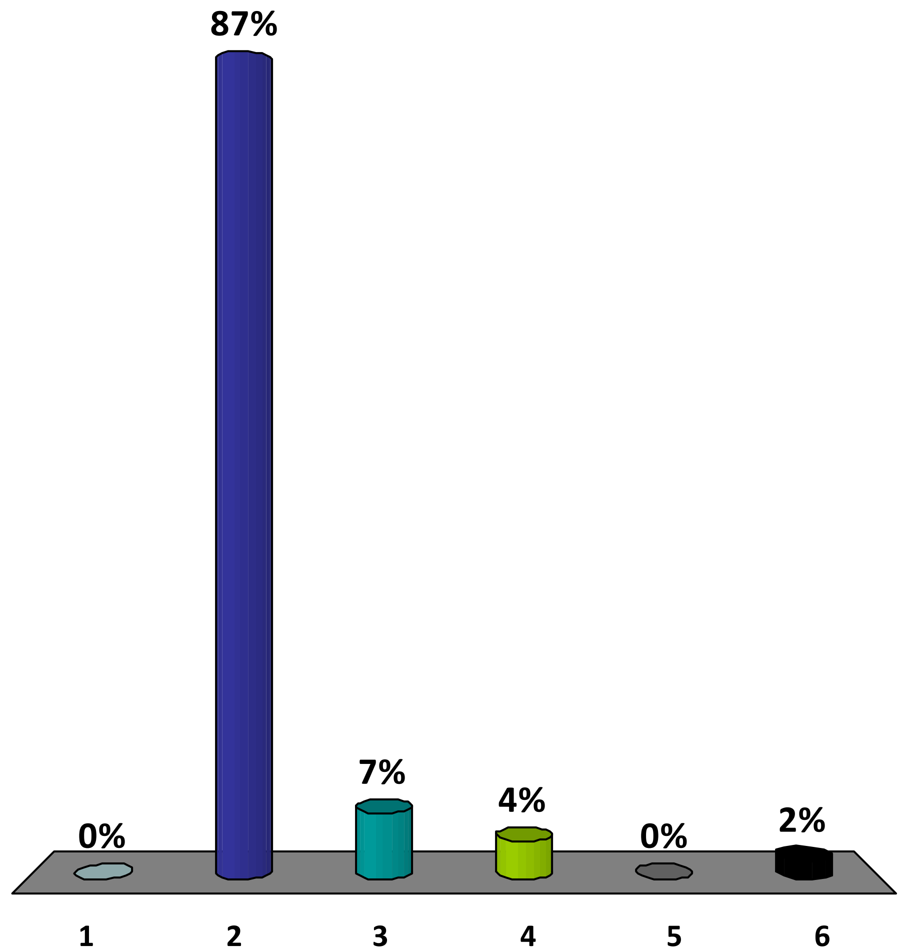
Sorted array: A[1..n]

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

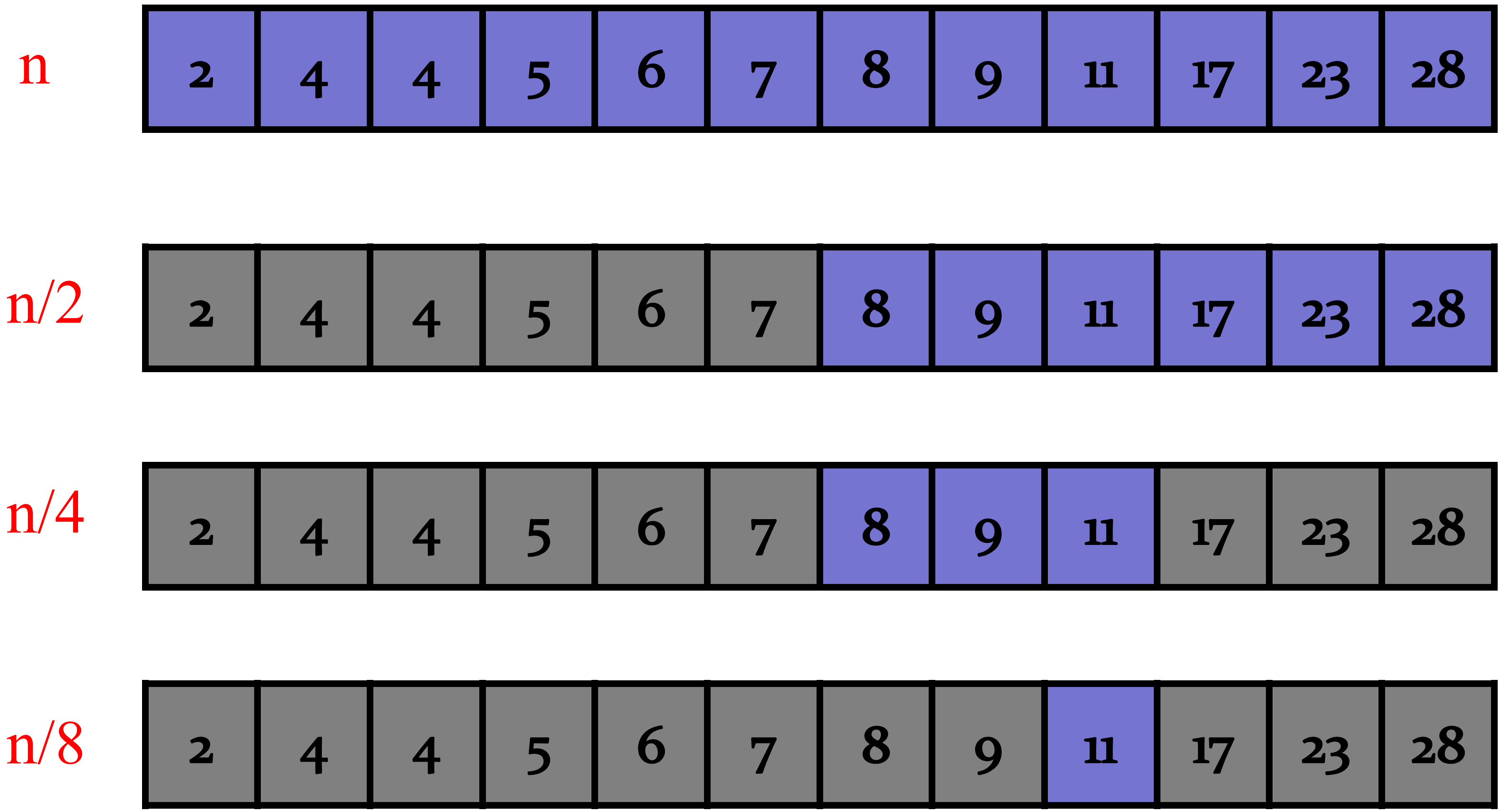
```
Search(A, key, n)
    begin = 0
    end = n
    while begin < end - 1   do:
        if  key < A[(begin+end)/2] then
            end = (begin+end)/2
        else begin = (begin+end)/2
    return A[begin]
```

What is the running time of Binary Search?

1. O(1)
2. $\boxed{\text{O(log n)}}$
3. O(n)
4. O(n log n)
5. O(n^2)
6. I'm confused...



Binary Search



Binary Search

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

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

Iteration 1: (end – begin) = n

Iteration 2: (end – begin) = $n/2$

Iteration 3: (end – begin) = $n/4$

…

Iteration k: (end – begin) = 1 = $n/2^k$

$n/2^k = 1$ ➔ $k = \log(n)$

Binary Search

Sorted array: A[1..n]

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

Not just for searching arrays:

- Assume a complicated function:

  int complicatedFunction(int s)

- Assume the function is always increasing:

  complicatedFunction(i) < complicatedFunction(i+1)

- Find the minimum value j such that:

  complicatedFunction(j) > 100

Today: Divide and Conquer!

Algorithm Analysis

- Big-O Notation
- Model of computation

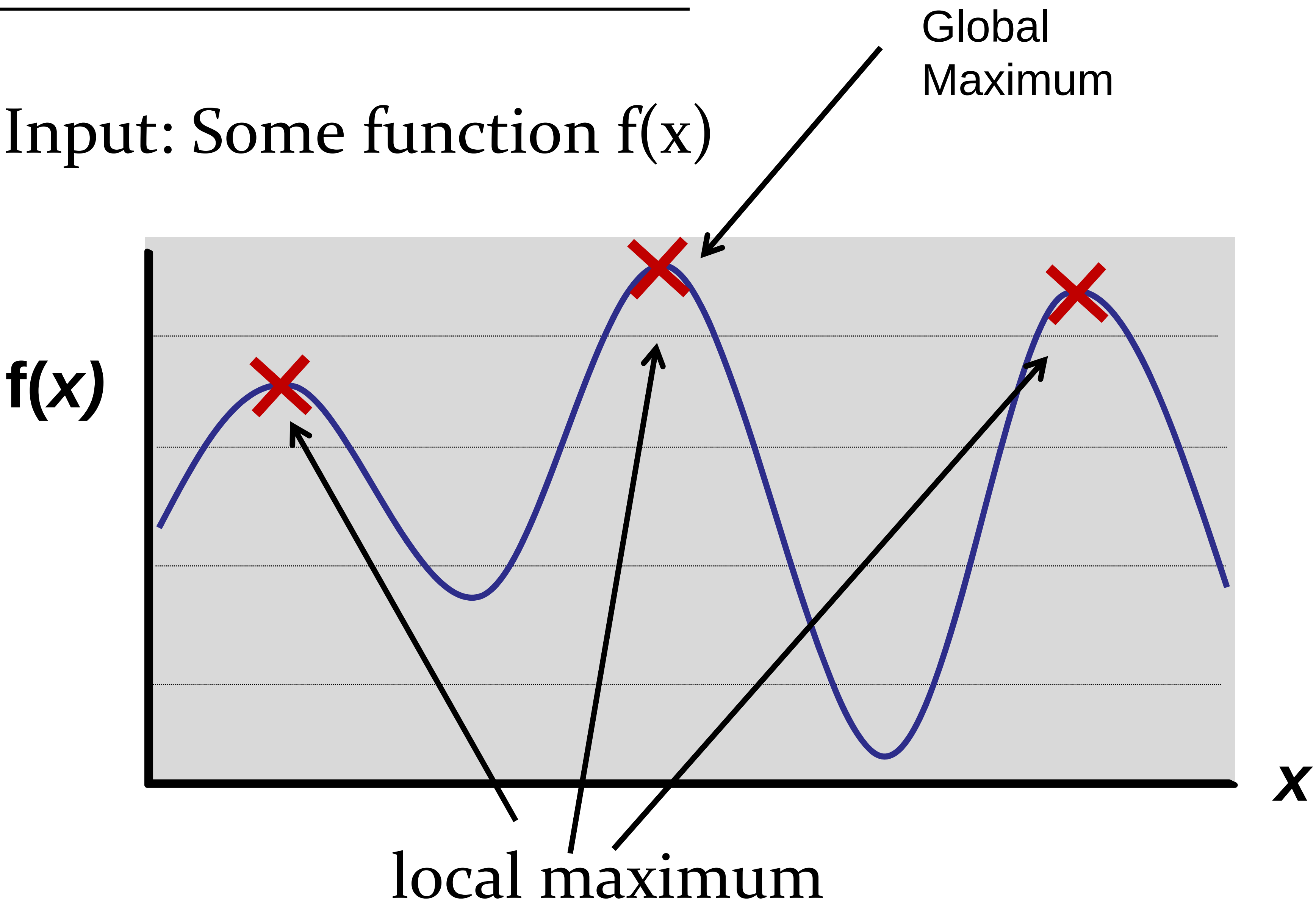
Searching

Peak Finding

- 1-dimension
- 2-dimensions

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 $A[1..n]$

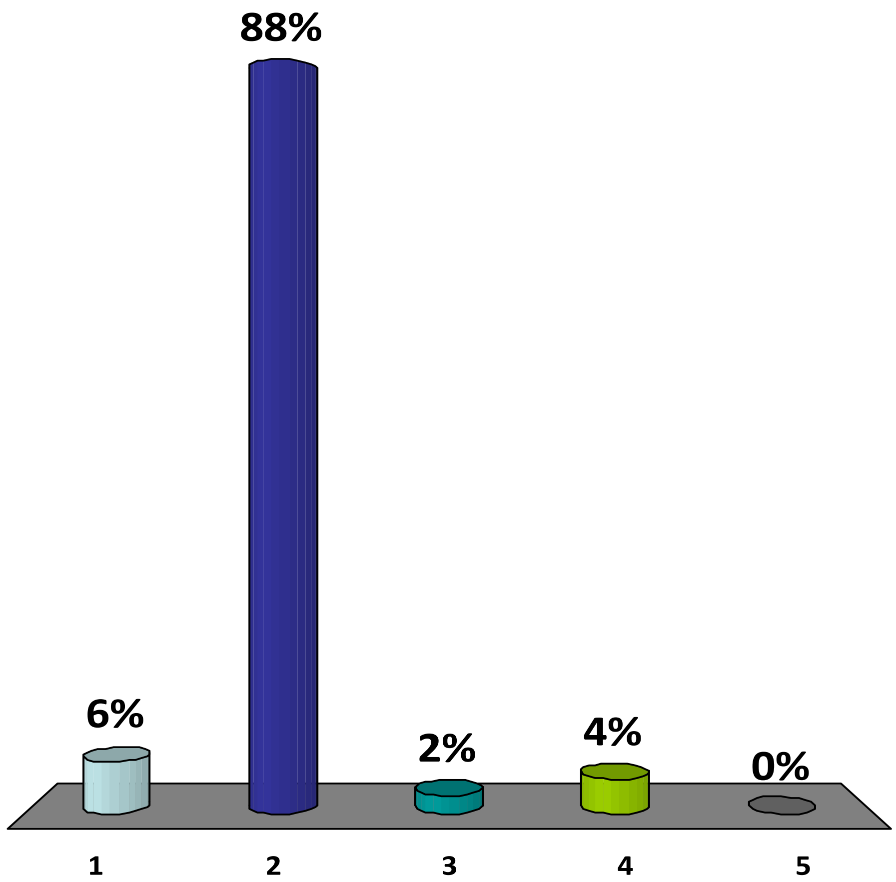
Output: global maximum element in A

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]`

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]
```

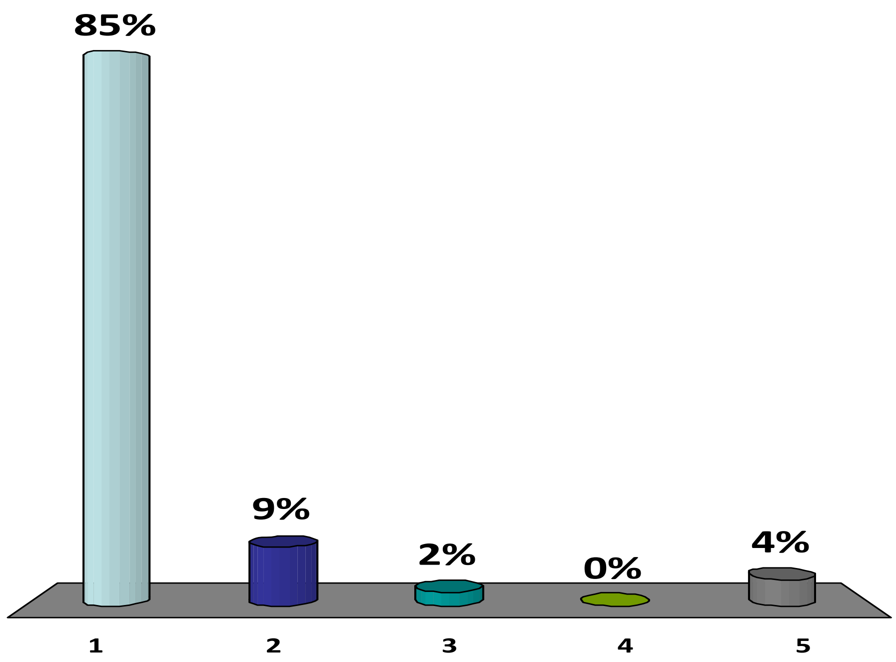
Time Complexity: O(n)

Global Maximum

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

How long to find the maximum?

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



Global Maximum

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

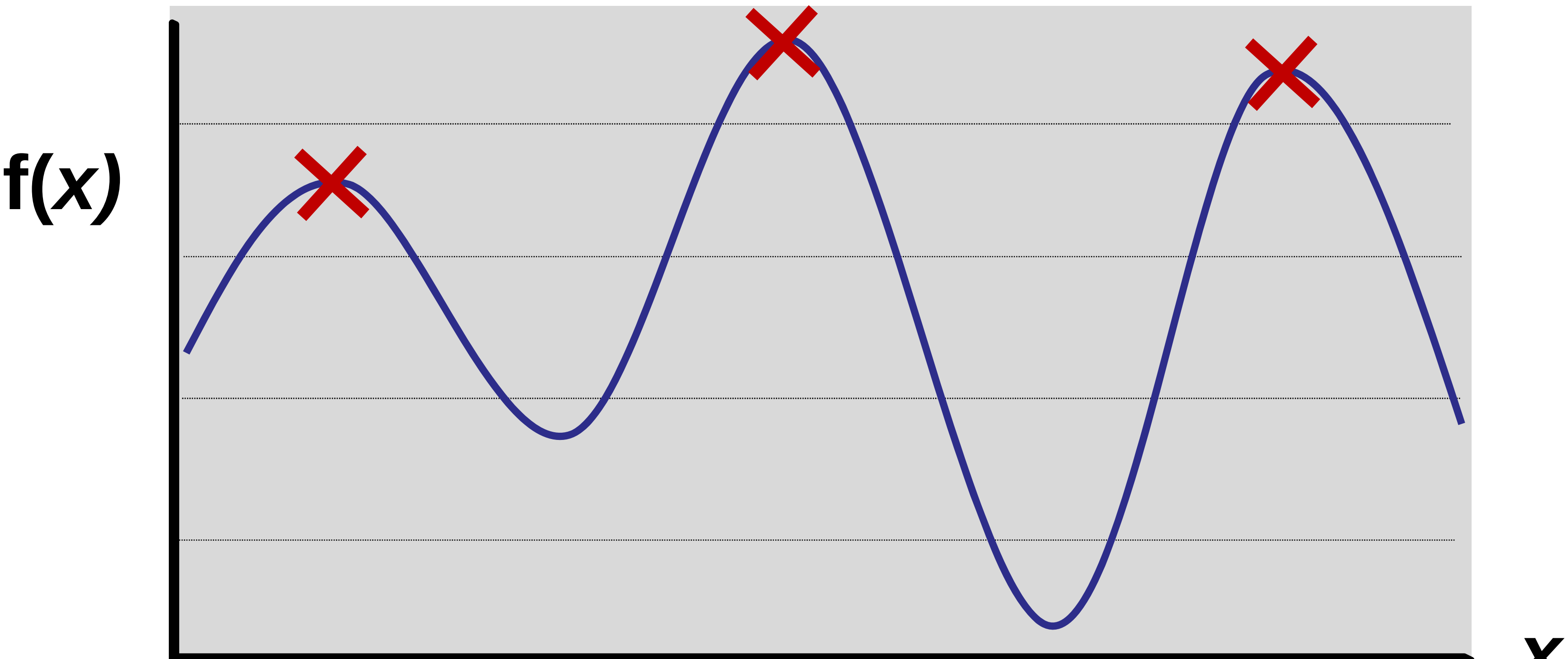
2	4	4	5	6	7	8	9	11	17	23	28

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

Time Complexity: $O(1)$

Peak (Local Maximum) Finding

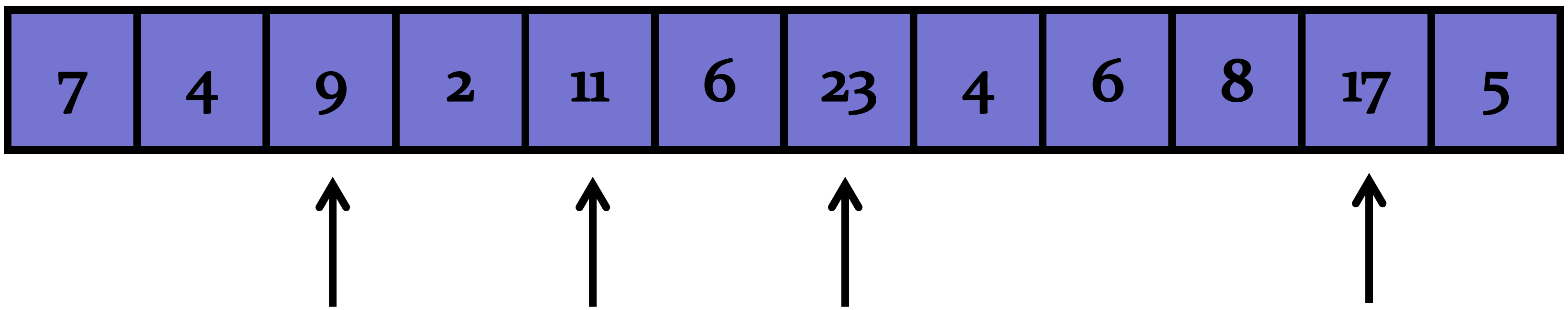
Input: Some function f(x)



Output: A local maximum

Peak Finding

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



Output: a local maximum in A

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

And we assume that

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

Peak Finding: Algorithm 1

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

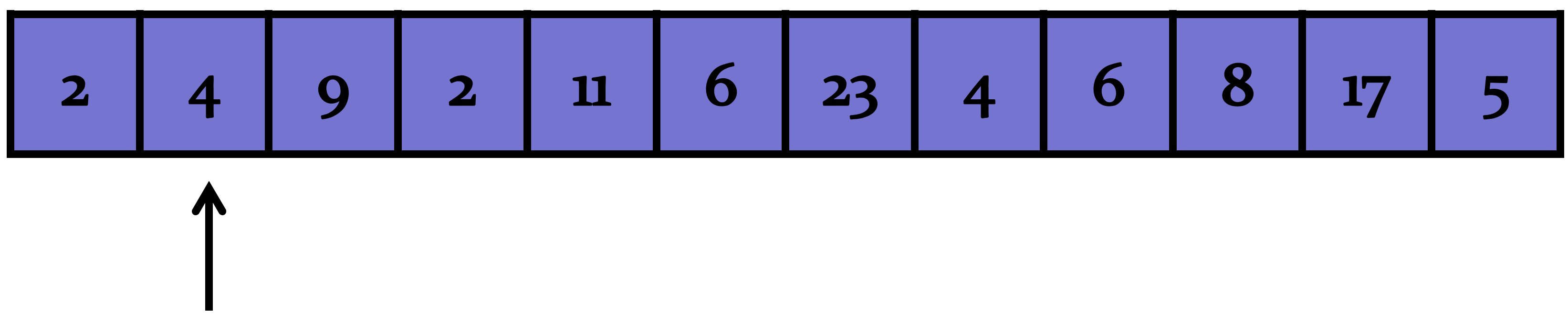
2	4	9	2	11	6	23	4	6	8	17	5

FindPeak

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

Peak Finding: Algorithm 1

Input: Some array `A[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]

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[1..n]

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

Running time: n

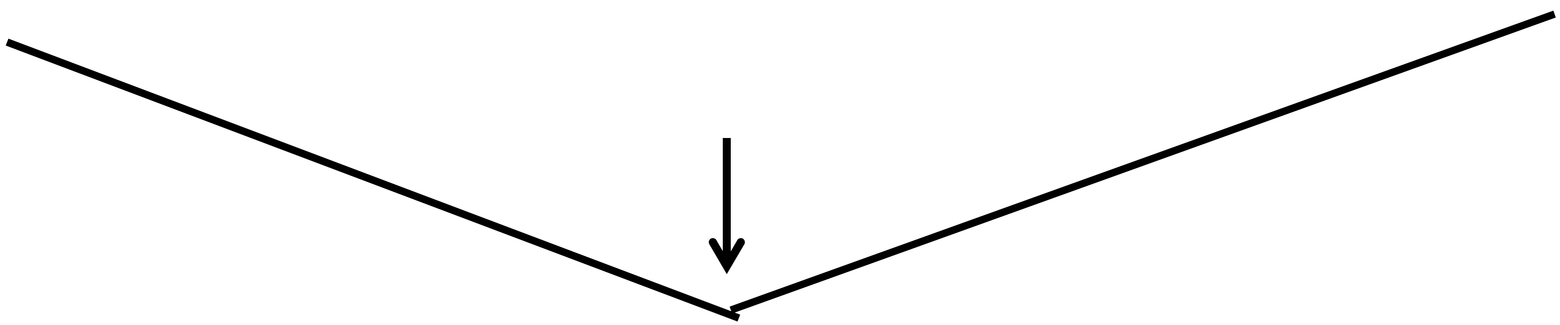
Simple improvement?

Peak Finding: Algorithm 1

Input: Some array A[1..n]

2	4	9	2	11	6	9	11	13	8	17	5

Start in the middle!



Worst-case: n/2

Peak Finding: Algorithm 2

Divide-and-Conquer

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

Start in the middle

5 < **6?** OK

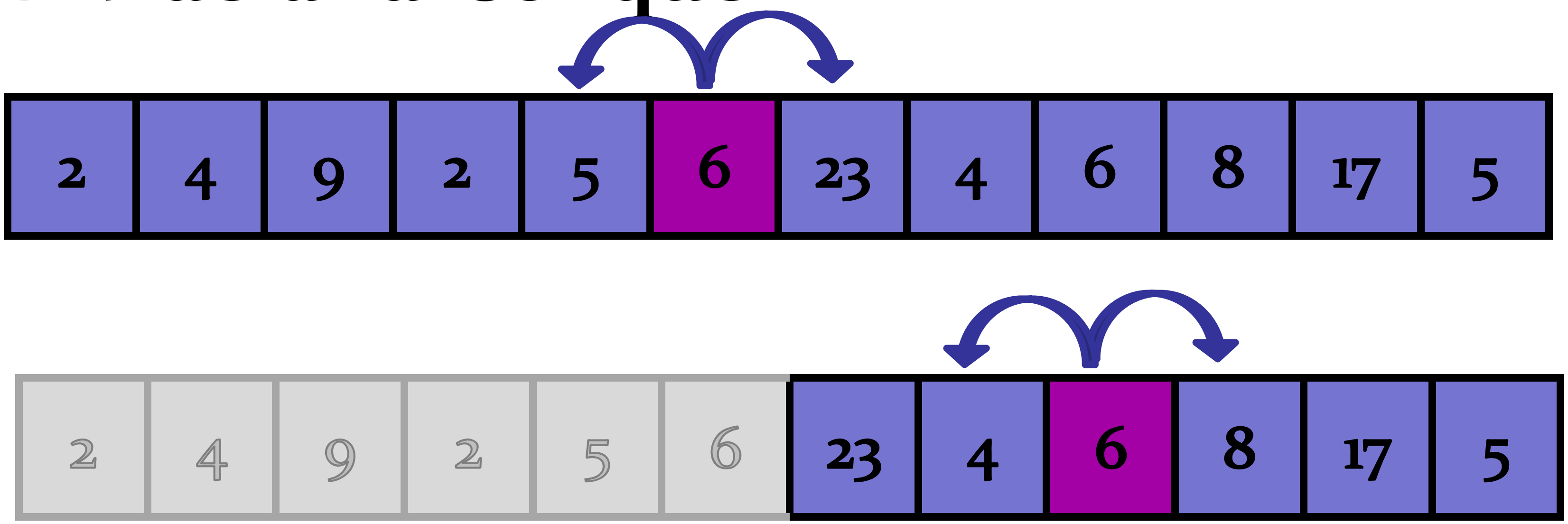
23 < **6?** NO

Recurse!

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

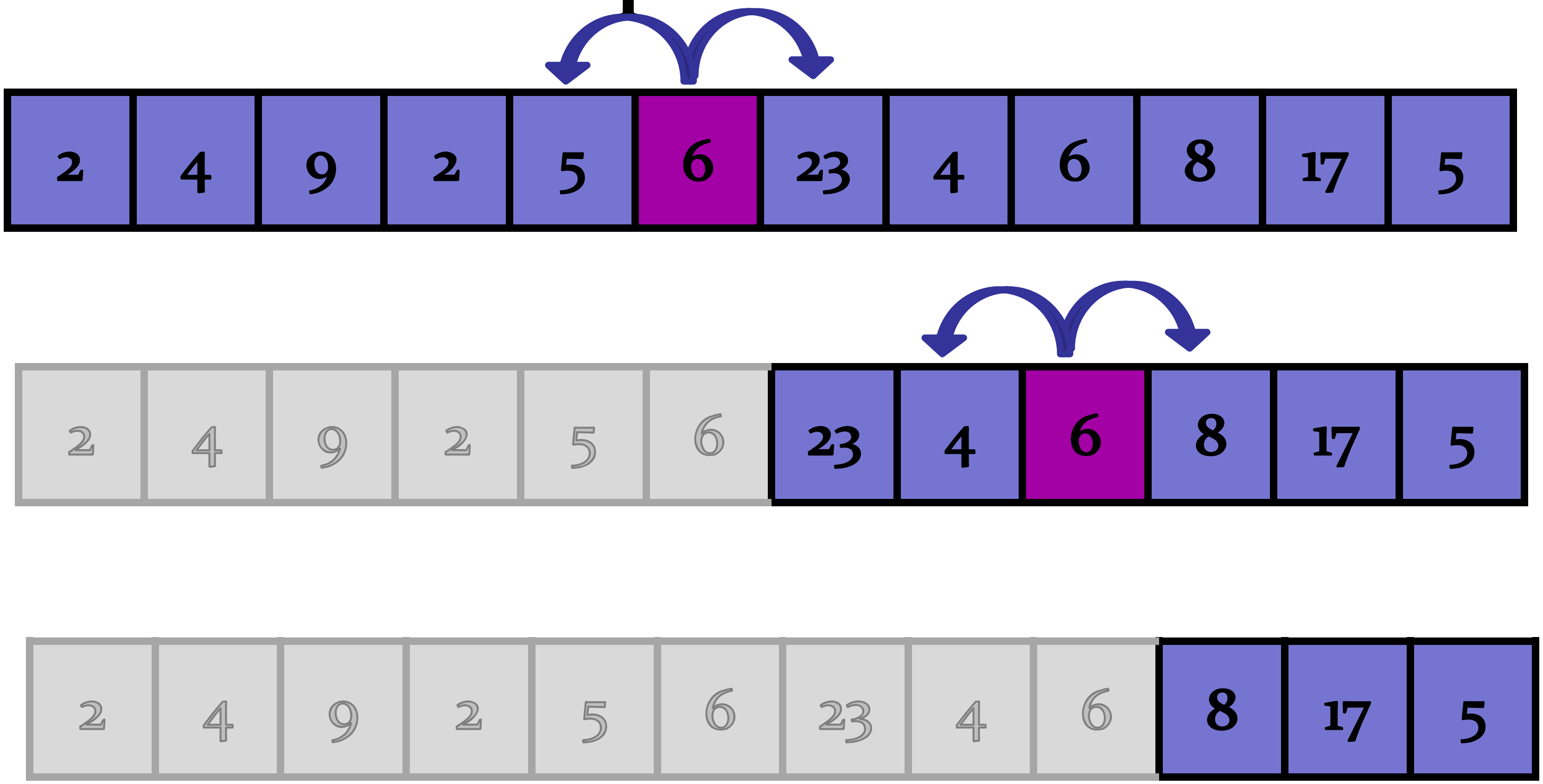
Peak Finding: Algorithm 2

Divide-and-Conquer



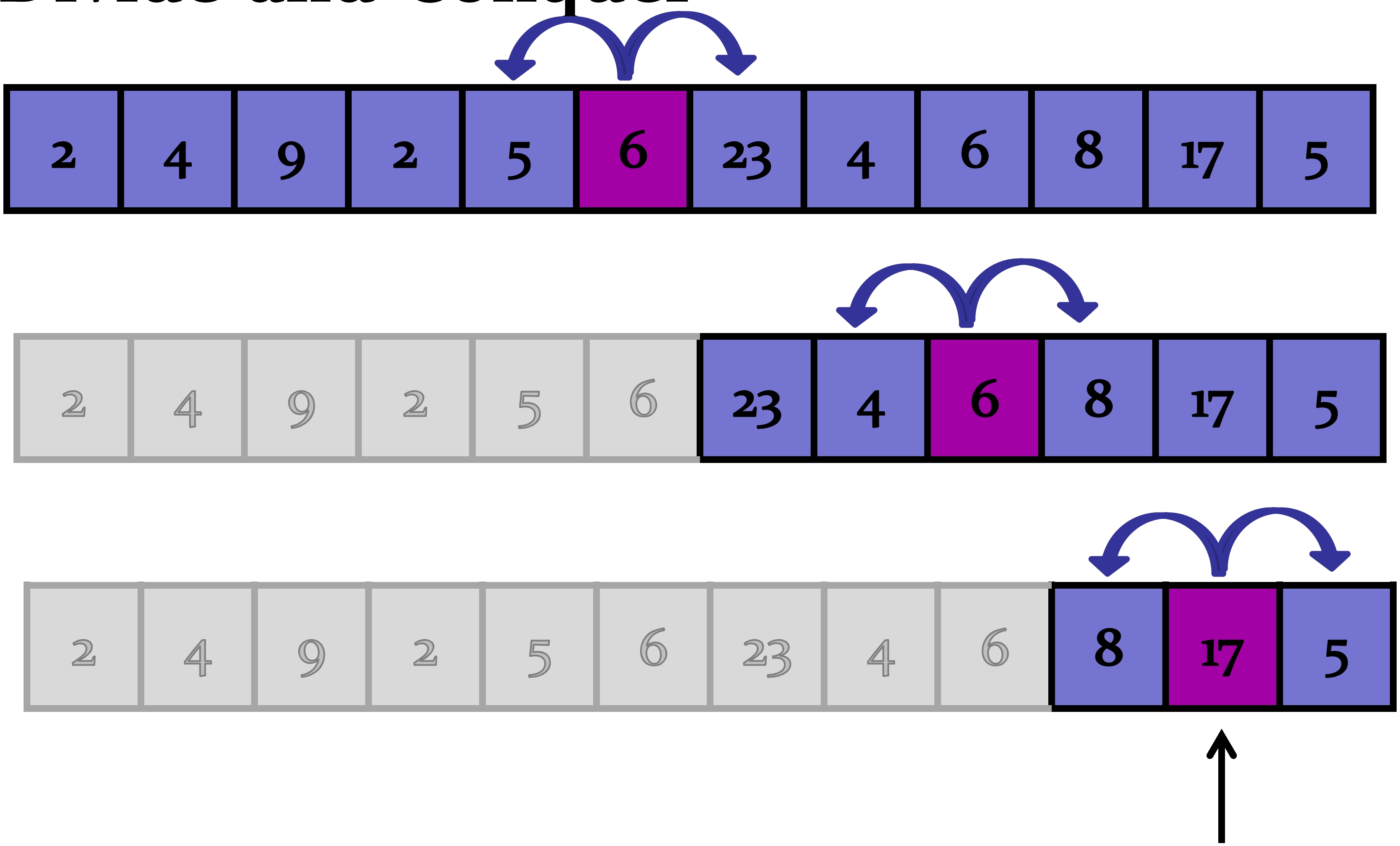
Peak Finding: Algorithm 2

Divide-and-Conquer



Peak Finding: Algorithm 2

Divide-and-Conquer



We found a peak!

Peak Finding: Algorithm 2

Input: Some array A[1..n]

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[1..n]

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: Algorithm 2

Why?

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

Key property:

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

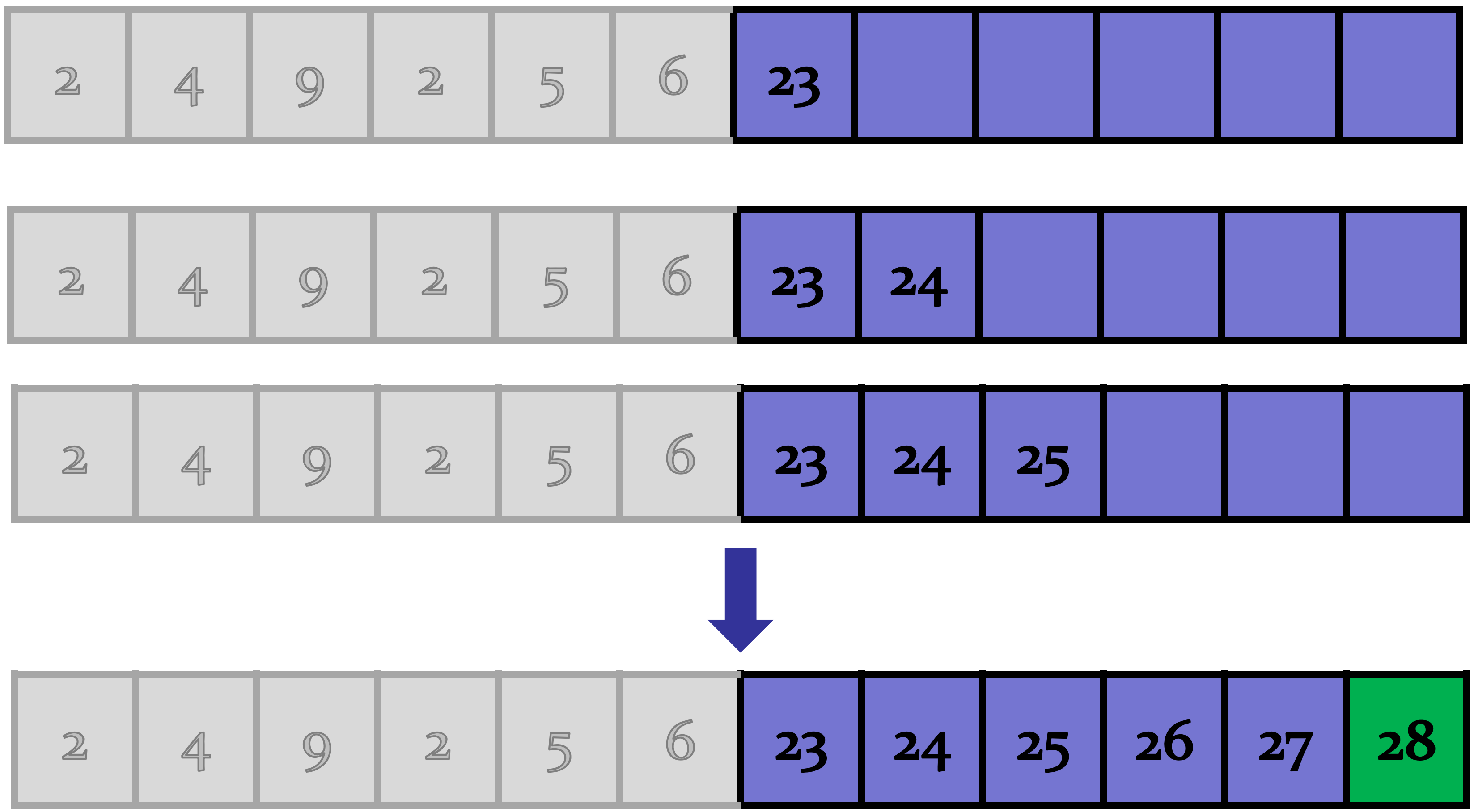
Explanation:

– 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.



Peak Finding: Algorithm 2

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

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

Unrolling the recurrence:

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

Peak Finding: Algorithm 2

Unrolling the recurrence:

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

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

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

$$\ldots$$

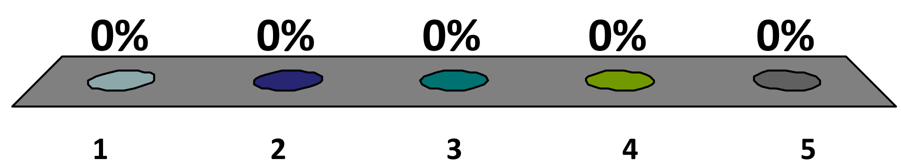
$$\ldots$$

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

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

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

Unrolling the recurrence:

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

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

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

$$\ldots$$

$$\ldots$$

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

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

Peak Finding: Algorithm 2

Running time:

Time to find a peak in an array of size n

Recursion

Time for comparing A[n/2] with neighbors

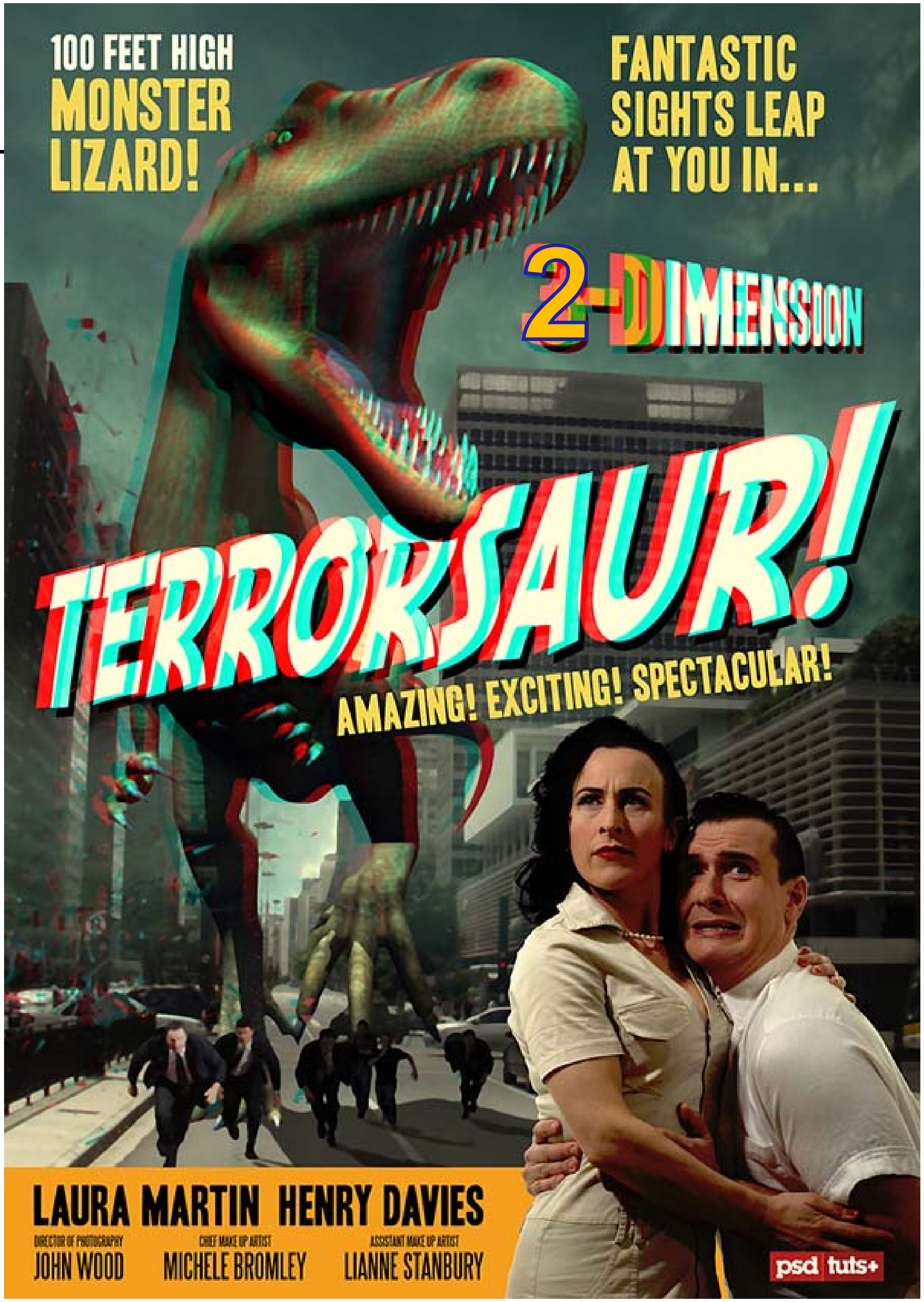
$$T(n) = T(n/2) + \theta(1)$$

Unrolling the recurrence:

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

Onwards…

The 2nd dimension!



Peak Finding 2D (the sequel)

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

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

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.

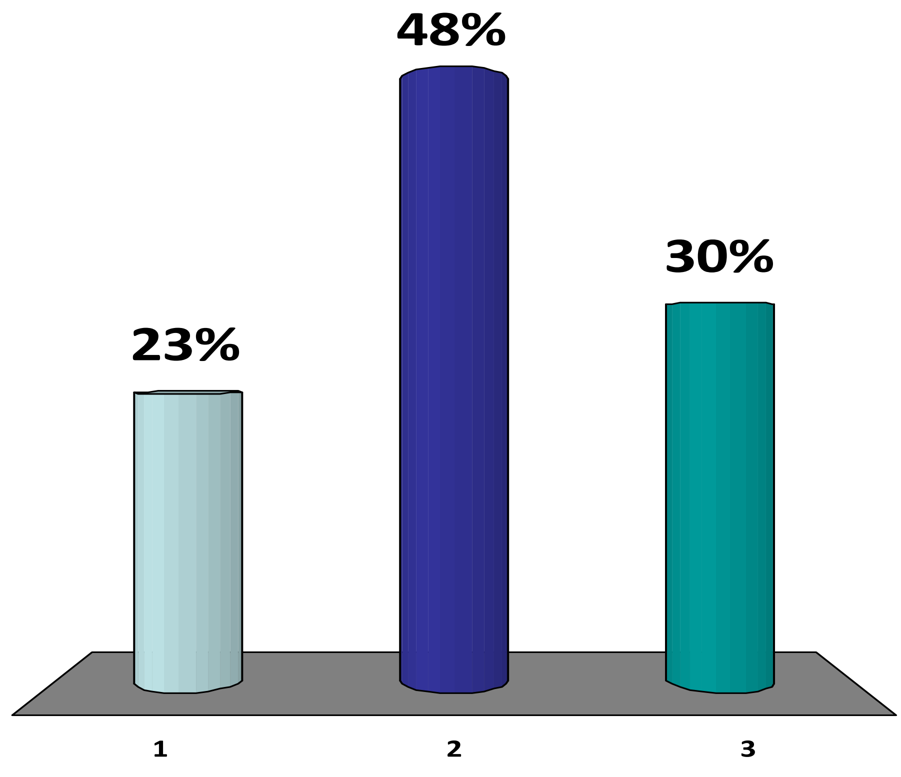
Algorithm 1-2D

Step 1: Find global max for each column.

Step 2: Find peak in the max array.

Is this algorithm correct?

1. Yes
2. No
3. I'm confused...



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

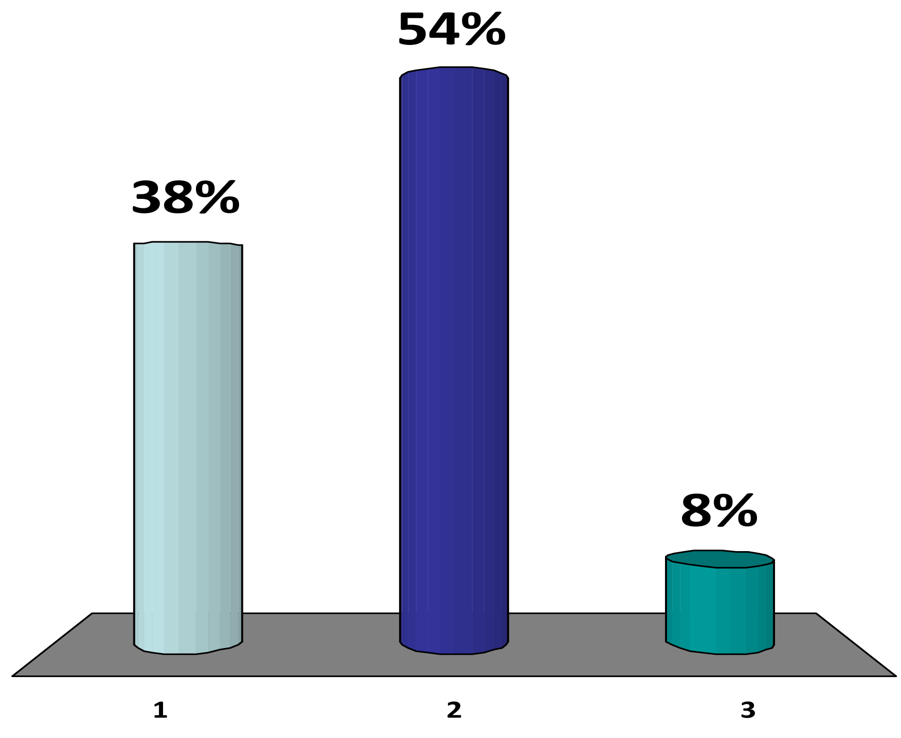
Algorithm 2-2D

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

Step 2: Find peak in the max array.

Is this algorithm correct?

1. Yes
2. No
3. I'm confused...



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.

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 <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: 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.

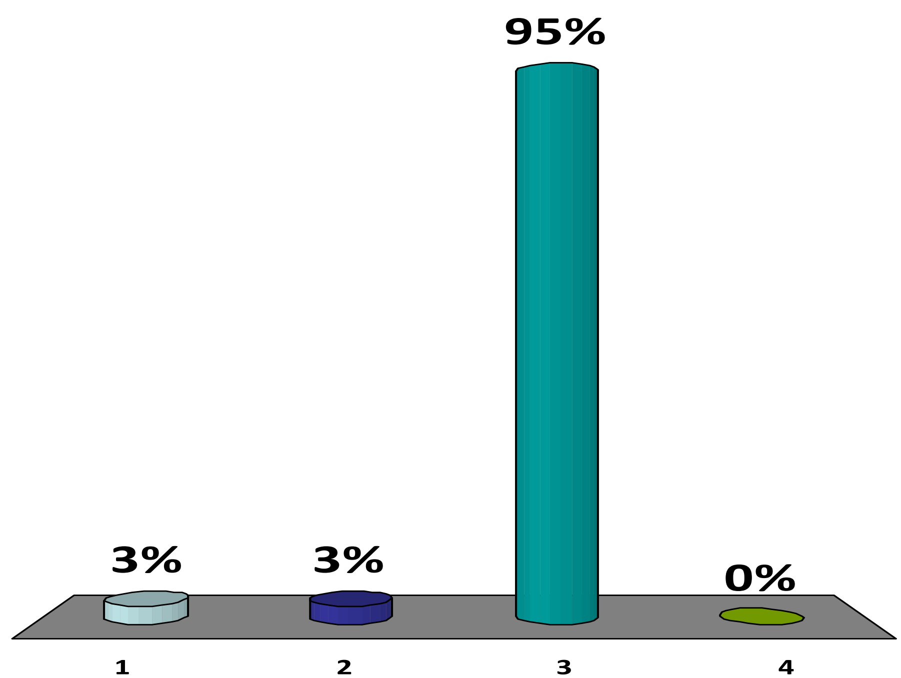
Step 2: Recurse left/right half.

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: $\mathbf{O(n \log m)}$

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

2D Algorithm 3

Any ideas??

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.

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

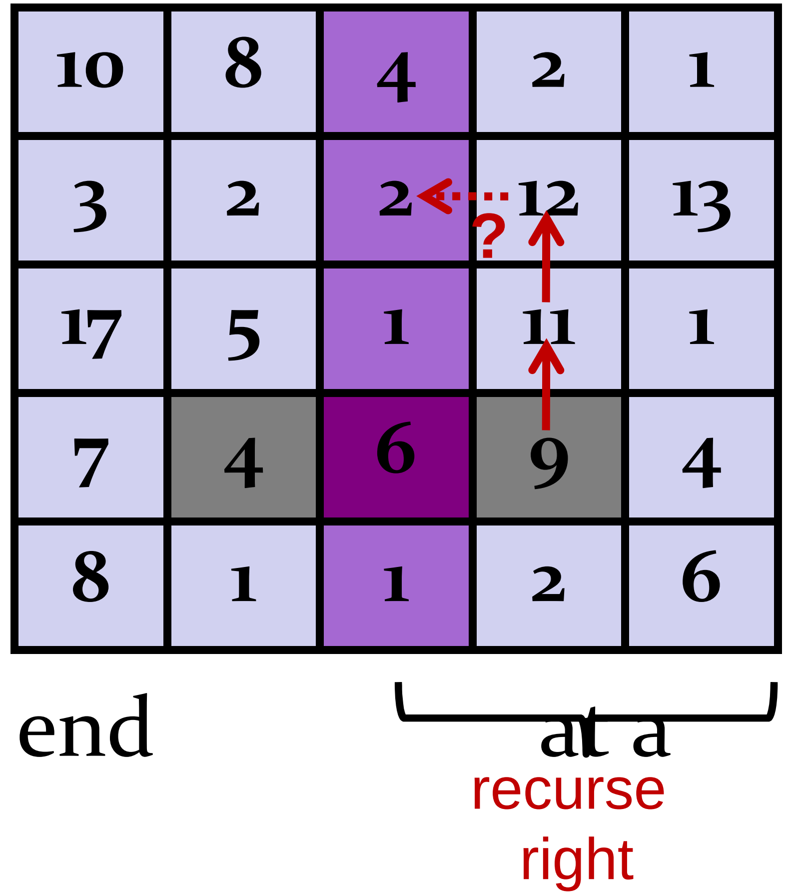
recurse right

2D Algorithm 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.

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

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

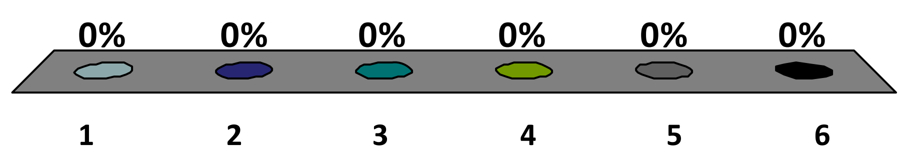
Recurrence Analysis

Response Counter

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

$T(n) = ??$

1. O(log n)
2. O(log m)
3. O(nm)
4. O(n log m)
5. O(m log n)
6. O(n! cos(Π/m))



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)

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

We want to do better than O(n log m)…

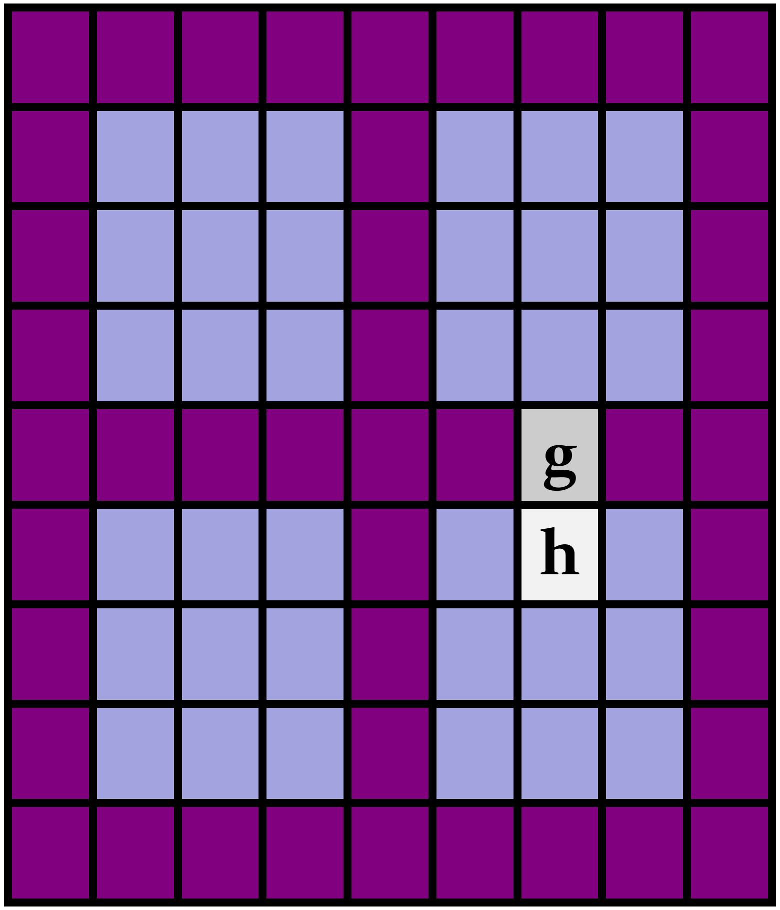
Any ideas??

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

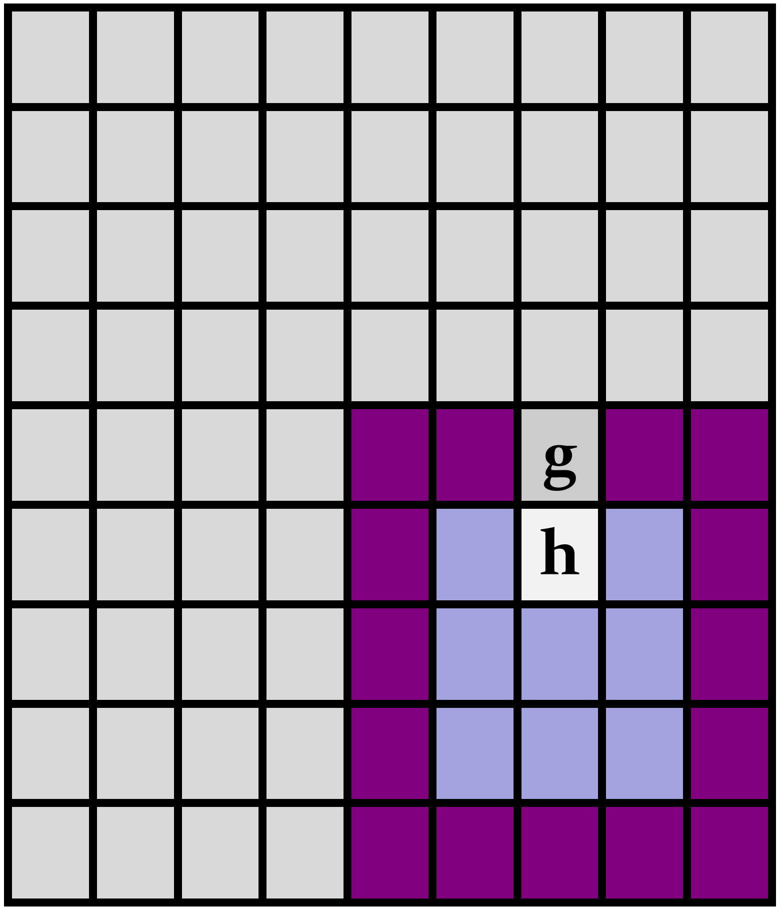
h > g

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

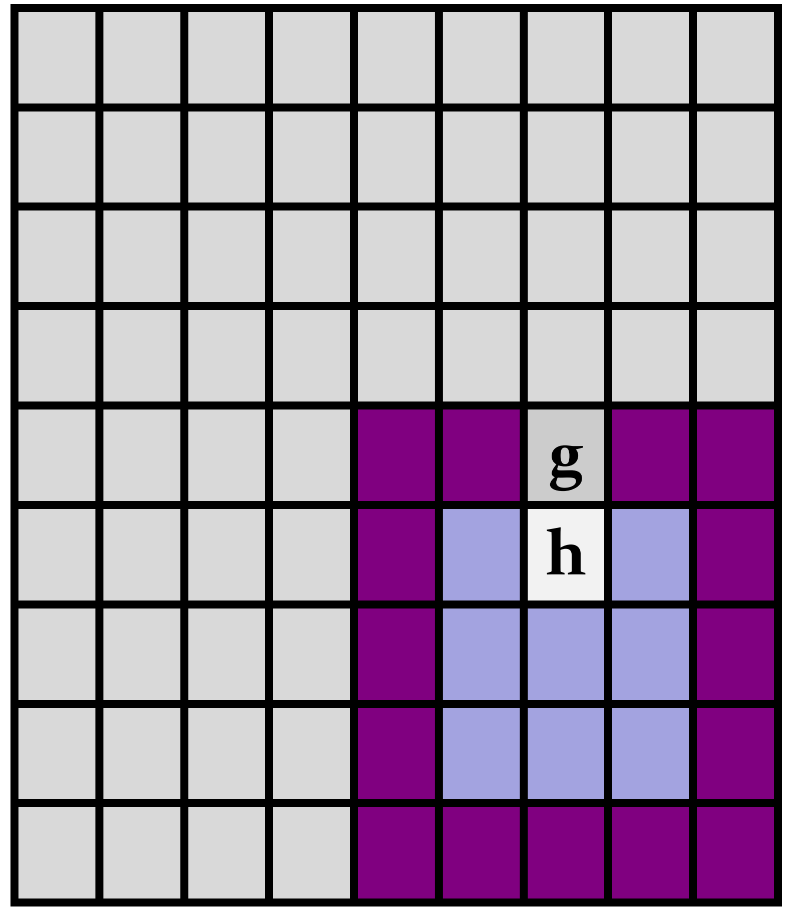
2D Algorithm 4

Correctness

1. The quadrant contains a peak.

   Proof: as before.

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



2D Algorithm 4

Correctness

1. The quadrant contains a peak.

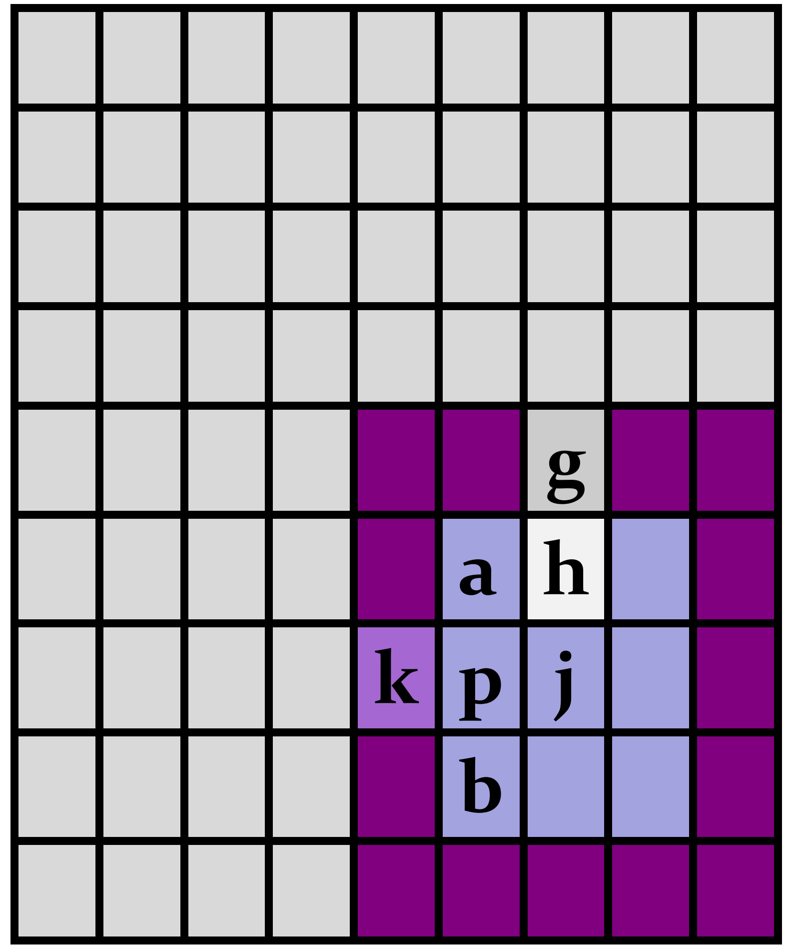
   Proof: as before.

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

   Example: k > p > j

   p > a

   p > b



2D Algorithm 4

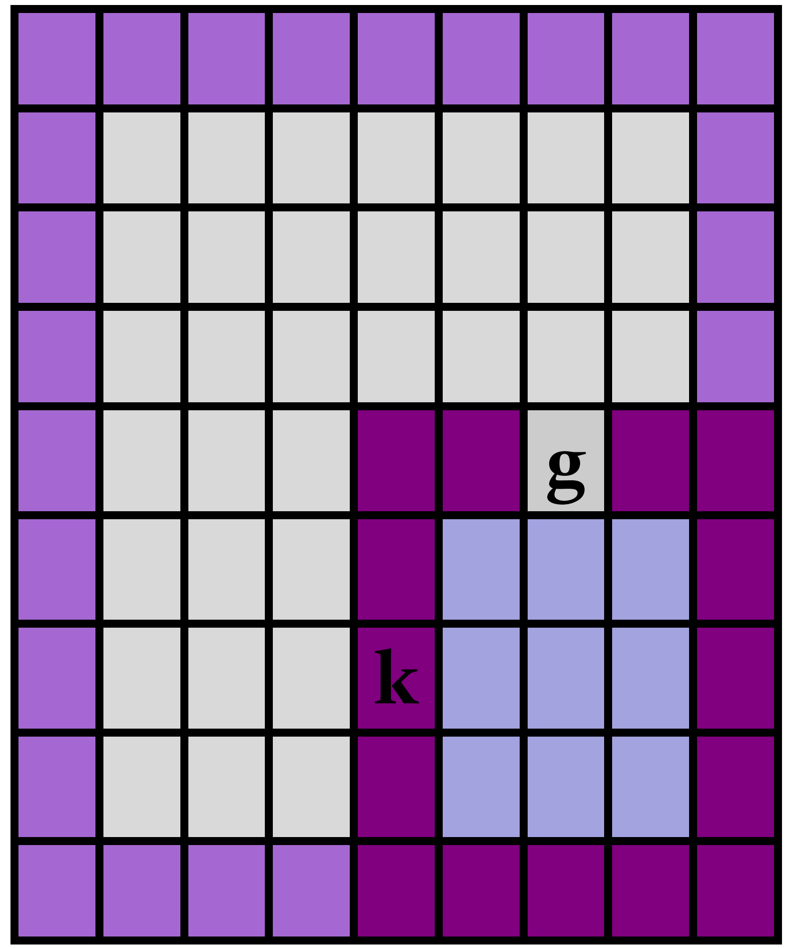
Correctness

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

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

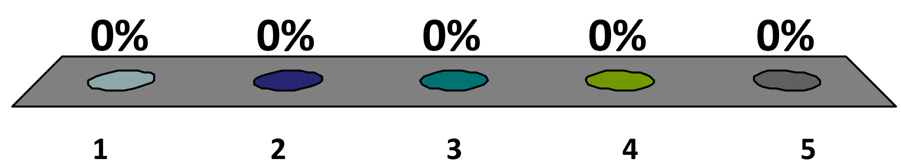
Recurrence Analysis

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

$T(n) = ??$

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

Response Counter



Recurrence Analysis

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

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)