# Algorithms and Data Structures

U S - Cela

# Syllabus

- Introduction (1)
- Correctness, analysis of algorithms (2,3,4)
- Sorting (1,6,7)
- Elementary data structures, ADTs (10)
- Searching, advanced data structures (11,12,13,18)
- Dynamic programming (15)
- Graph algorithms (22,23,24)
- Computational Geometry (33)
- NP-Completeness (34)

# Data Structures and Algorithms

- Algorithm
  - Outline, the essence of a computational procedure, step-by-step instructions
- Program – an implementation of an algorithm in some programming language
- Data structure
  - **Organization** of data needed to solve the problem

# Algorithmic problem

Specification of input → **?** → Specification of output as a function of input

- Infinite number of input *instances* satisfying the specification.

divide(int x, int y)

{ return $\frac{x}{y}$ ;

# Algorithmic Solution

Input instance, adhering to the specification

Algorithm

Output related to the input as required

- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

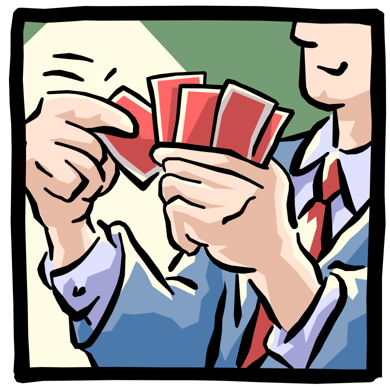
# Example: Sorting

**INPUT**

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



**OUTPUT**

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

**Correctness**

For any given input the algorithm halts with the output:

- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

**Running time**

Depends on

- number of elements ($n$)
- how (partially) sorted they are
- algorithm

# Insertion Sort

**A**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 6 | 8 | 9 | 7 | 2 | 5 | 1 |

1 j → n

← i

**Strategy**

- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

```
for j=2 to length(A)
   do key=A[j]
   "insert A[j] into the
   sorted sequence A[1..j-1]"
     i=j-1
     while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
     A[i+1]:=key
```

# Analysis of Algorithms

- Efficiency:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - Number of data elements (numbers, points)
  - A number of bits in an input number

# The RAM model

- Very important to choose the level of detail.
- The RAM model:
  - Instructions (each taking constant time):
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
  - Data types – integers and floats

# Analysis of Insertion Sort

- Time to compute the **running time** as a function of the **input size**

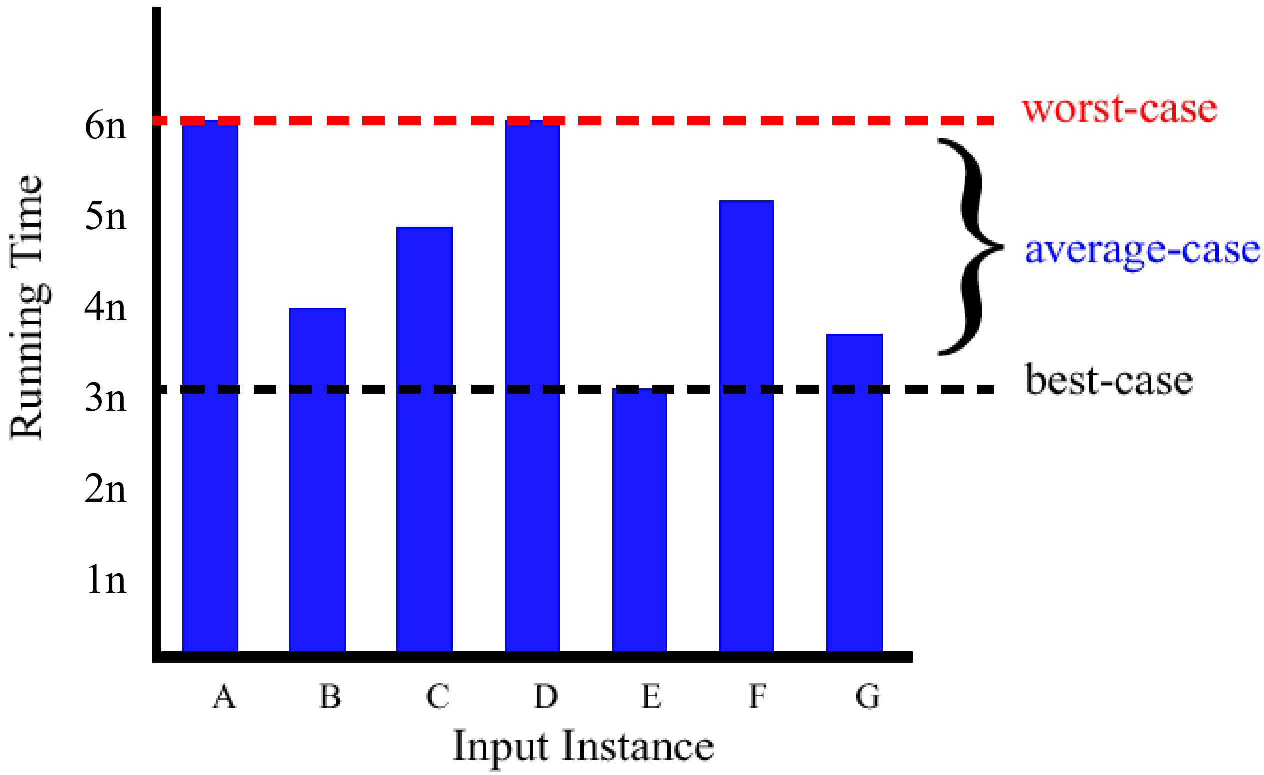
| | cost | times |
|---|---|---|
| **for** j=2 **to** *length*(A) | $c_1$ | n |
| **do** key=A[j] | $c_2$ | n-1 |
| "insert A[j] into the sorted sequence A[1..j-1]" | 0 | n-1 |
| i=j-1 | $c_3$ | n-1 |
| **while** i>0 **and** A[i]>key | $c_4$ | $\sum_{j=2}^{n} t_j$ |
| **do** A[i+1]=A[i] | $c_5$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| i-- | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| A[i+1]:=key | $c_7$ | n-1 |

# Best/Worst/Average Case

- **Best case**: elements already sorted $\rightarrow$ $t_j=1$, running time = *f(n)*, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order
  $\rightarrow$ $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

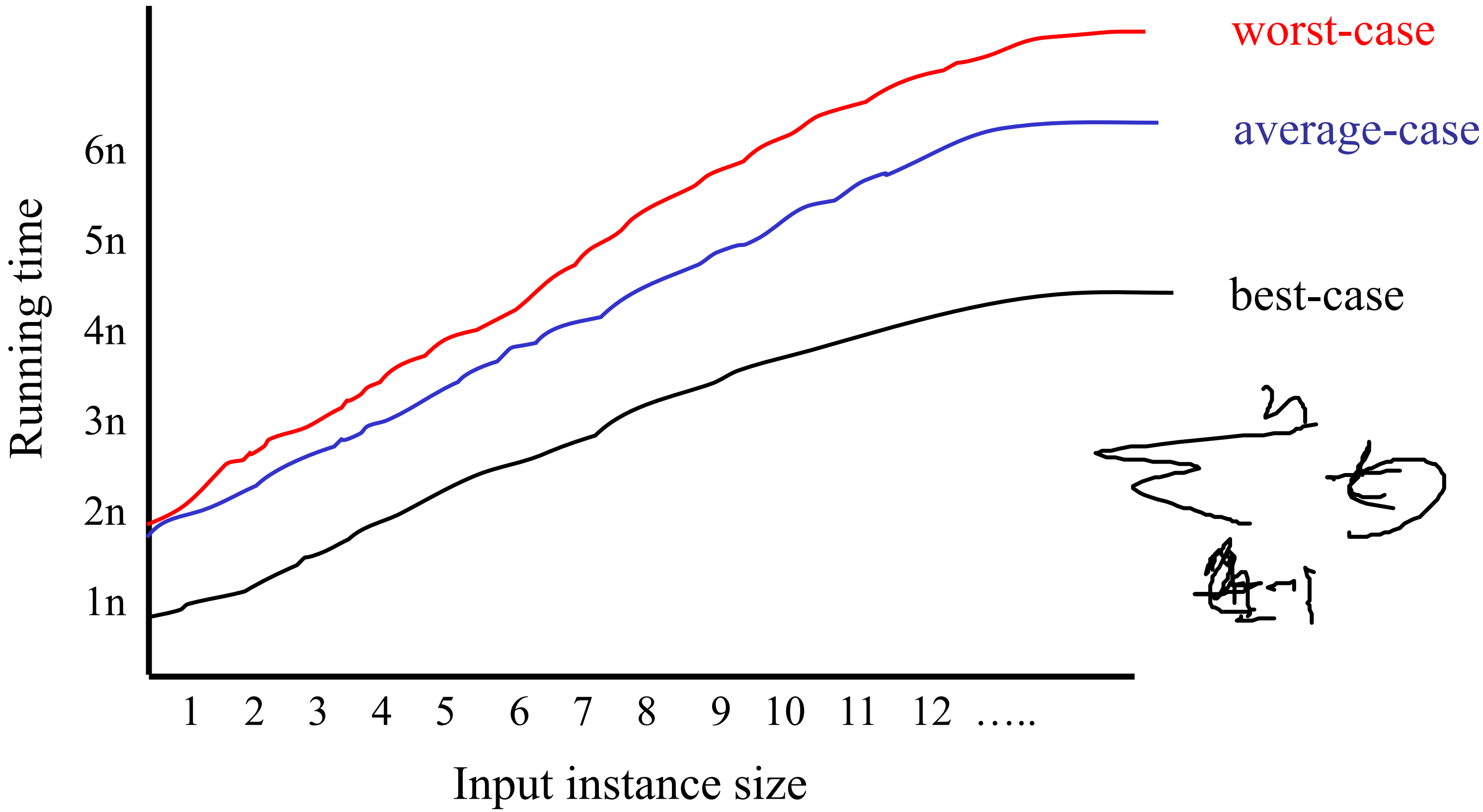
# Best/Worst/Average Case (2)

- For a specific size of input $n$, investigate running times for different input instances:



# Best/Worst/Average Case (3)

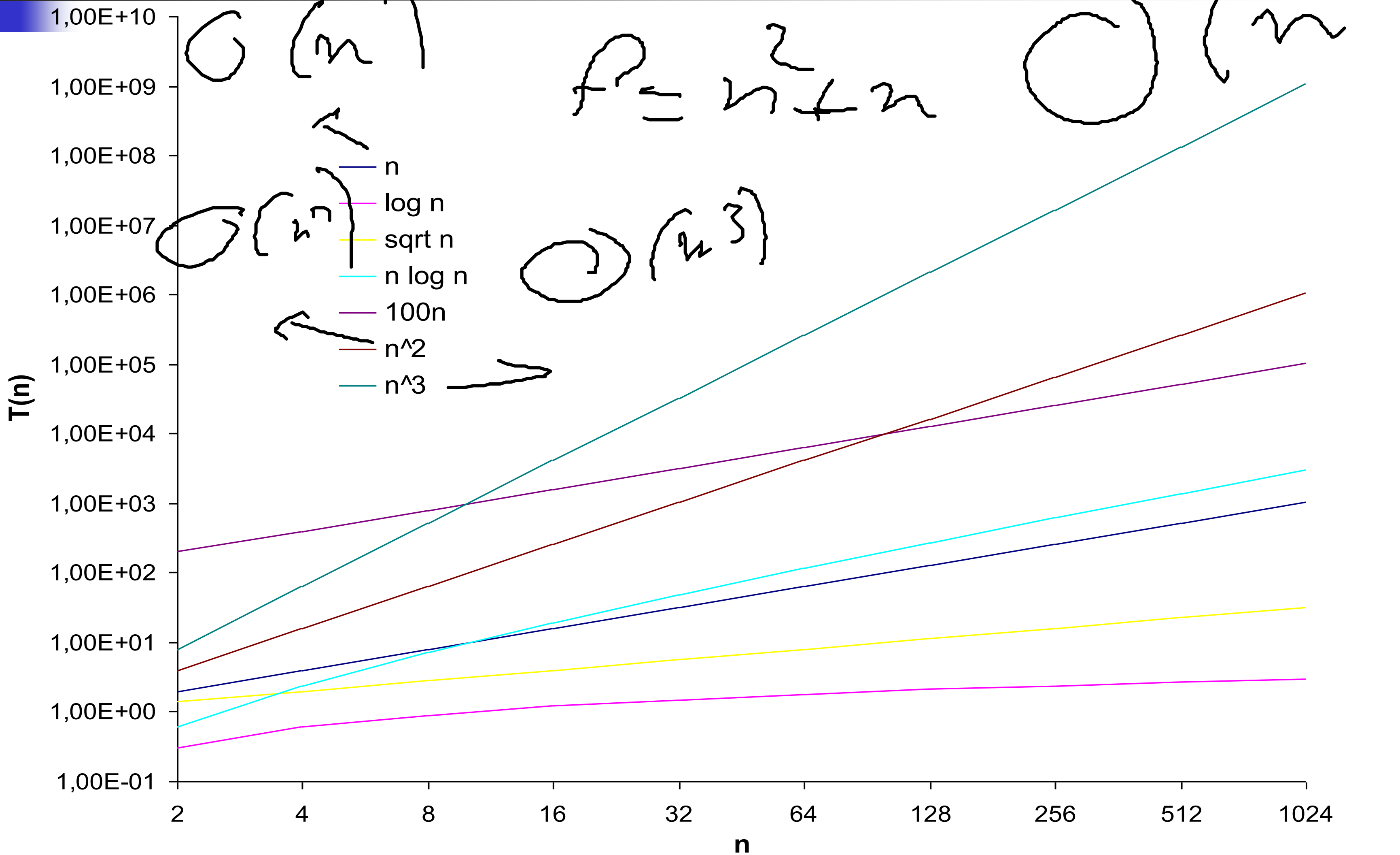
- For inputs of all sizes:



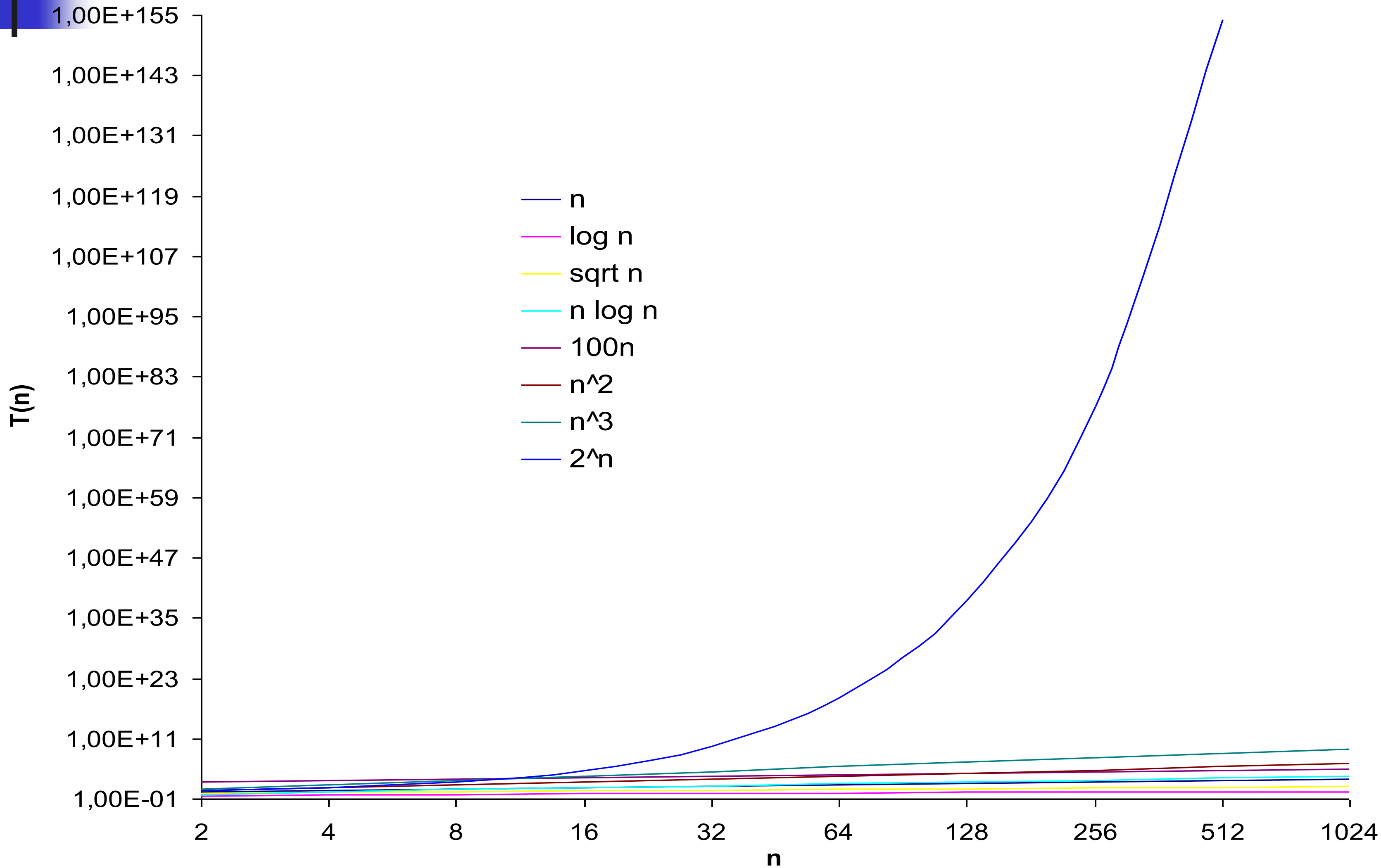
# Best/Worst/Average Case (4)

- **Worst case** is usually used:
  - It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
  - For some algorithms **worst case** occurs fairly often
  - The **average case** is often as bad as the **worst case**
  - Finding the **average case** can be very difficult

# Growth Functions



# Growth Functions (2)



# That's it?

- Is **insertion sort** the best approach to sorting?
- Alternative strategy based on divide and conquer
- MergeSort
  - sorting the numbers <4, 1, 3, 9> is split into
  - sorting <4, 1> and <3, 9> and
  - merging the results
  - Running time *f(n log n)*

# Example 2: Searching

## INPUT

- sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \ldots, a_n; \; q$ $\longrightarrow$ $j$

2 5 4 10 7; 5 $\longrightarrow$ 2

2 5 4 10 7; 9 $\longrightarrow$ *NIL*

## OUTPUT

- an index of the found number or *NIL*

# Searching (2)

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst-case running time: *f(n)*, average-case: *f(n/2)*
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

# Example 3: Searching

## INPUT

- sorted non-descending sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \ldots., a_n;\ q$ $\longrightarrow$

2 4 5 7 10; 5

2 4 5 7 10; 9

## OUTPUT

- an index of the found number or *NIL*

$\xrightarrow{j}$

2

*NIL*

# Binary search

- Idea: Divide and conquer, one of the key design techniques

```
left=1
right=length(A)
do
   j=(left+right)/2
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return NIL
```

# Binary search – analysis

- How many times the loop is executed:
  - With each execution its length is cult in half
  - How many times do you have to cut *n* in half to get 1?
  - *lg n*