1.
Basics of Algorithm Analysis

Time Complexity of an Algorithm

Purpose

- To estimate how long a program will run
- To estimate the largest input that can reasonably be given to the program
- To compare the efficiency of different algorithms
- To choose an algorithm for an application

Time complexity is a function

Time for a sorting algorithm is different for sorting 10 numbers and sorting 1,000 numbers

Time complexity is a function: Specifies how the running time depends on the size of the input.

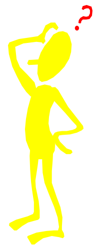
Function mapping

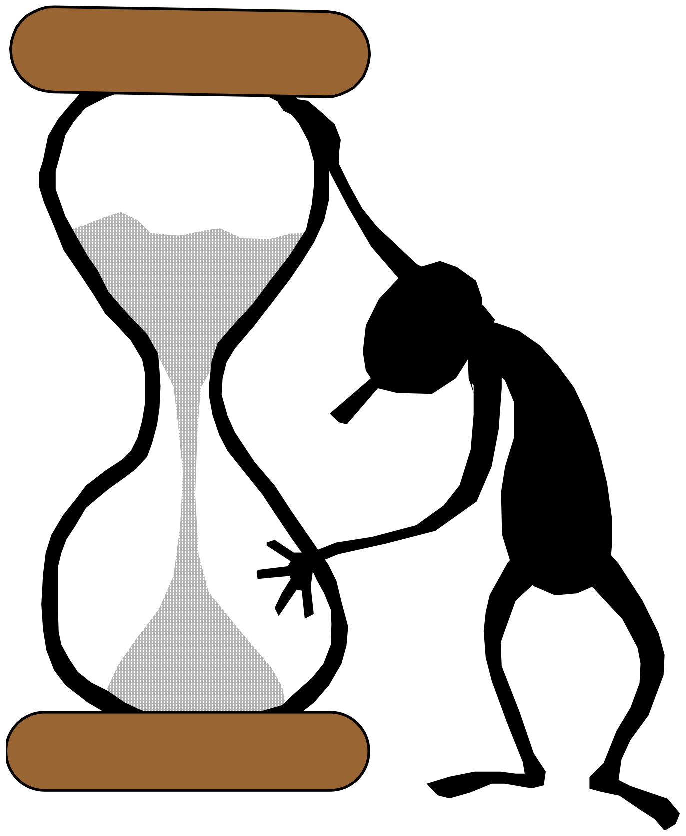
"size" n of input

↓

"time" $T(n)$ executed by algorithm

Definition of time?





Definition of time?

- # of seconds
- # lines of code executed
- # of simple operations performed

Definition of time?

- # of seconds   Problem: machine dependent
- # lines of code executed   Problem: lines of diff. complexity
- # of simple operations performed

this is what we will use

Size of input instance?

Formally: Size n is number of bits to represent instance

But we can work with anything reasonable

reasonable = within a constant factor of number of bits

Size of input instance

Ex 1:

83920

- # of bits: 17 bits - Formal
- # of digits: 5 digits - Reasonable: #bits and #digits are always within constant factor:
- Value: 83920

$$\#\text{bits} = (\log_2 10) \cdot \#\text{digits}$$
$$= \quad \sim 3.22 \cdot \#\text{digits}$$

Size of input instance

Ex 1:

- # of bits: 17 bits - Formal
- # of digits: 5 digits - Reasonable
- Value: 83920 - Not reasonable: $\approx 2^{\#\text{bits}}$, much bigger

Size of input instance

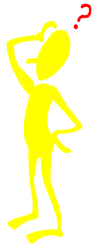
Ex 2:

10

14,23,25,30,31,52,62,79,88,98

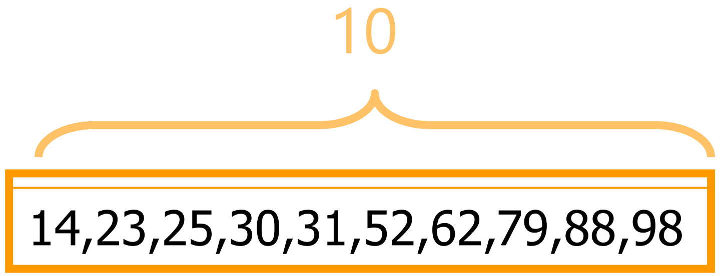
- # of elements = 10

Is this reasonable?



Size of input instance

Ex 2:



- # of elements = 10 - Reasonable if each number is, say, a 32-bit word, total number of bits is

#bits = 32 * #elements

Time complexity is a function

Time complexity is a function: Specifies how the running time depends on the size of the input

Function mapping

# of bits n to represent input

⬇

# of basic operations $T(n)$ executed by the algorithm

Which input of size n?

Q: There are 2^n inputs of size n. Which do we consider for the time complexity T(n)?

Worst instance

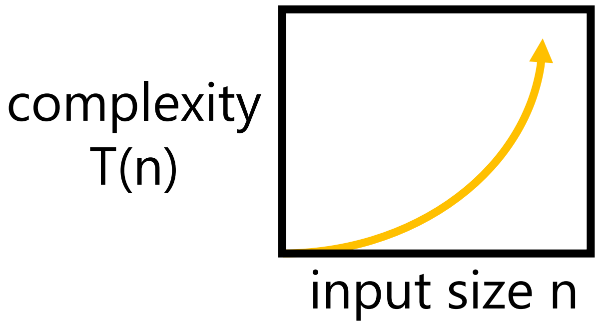
Worst-case running time. Consider the instance where the algorithm uses largest number of basic operations

- Generally captures efficiency in practice
- Pessimistic view, but hard to find better measure

Time complexity

We reach our final definition of time complexity:

T(n) = number of basic operations the algorithm takes over the worst instance of bit-size n



Example 1

```
Func Algorithm 1(A)        #A is array of bits
    x=20
    For i=1...len(A)
        x=3x
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) \approx 2n + 1$

- Input A of bit-size *n* has *n* entries
- $\approx$ 2 simple operations per step of **for**, +1 for "x=20"

(ignoring extra operations that make up the **For**)

Example 2

```
Func Algorithm 2(A)          #A is array of bits
    x=1
    For i=1…len(A)
      x=x+1
      If x>50 then
        x=x+3
      End If
    End For
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) \approx 5n - 99$

- Input A of bit-size *n* has *n* entries
- +1 for initialization “x=1”
- … ≈ 2+1 per iterations of **for** in the first 50 iterations
- … ≈ 2+1+2 per iterations of **for** in the other (n-50) iterations

(assuming $n \geq 50$)

Example 3

```
Func Algorithm 3(A)        #A is array of 32-bit numbers
  For i=1 to len(A)
    print "oi"
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) \approx (n/32)$

- A of bit-size *n* has *n/32* numbers
- ≈ 1 simple operations per iteration of **for**

Point: Understand input size

Example 4

```
Func Find10(A)            #A is array of 32-bit numbers
    For i=1 to len(A)
        If A[i]==10
            Return i
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) \approx (n/32) + 1$

- Worst instance: the only "10" is in the last position
- A of bit-size *n* has *n/32* numbers
- $\approx$ 1 simple operations per **for**, +1 for **Return**

Point: Complexity of algo is **always** about the **worst instance**

Asymptotic Order of Growth

Asymptotic Order of Growth

Motivation: Determining the exact time complexity T(n) of a real algorithm is very hard and often does not make much sense

In particular, can we say that an algorithm with complexity

T(n) = 10n

will run slower than an algorithm with complexity

T(n) = 9n?

No; for example, maybe the opertions in the first algorithm are slightly faster then in the second
(e.g., addition vs. multiplication)

But as we will see, for large instances there is a difference between $\approx n$ and $\approx n^2$
(e.g., ≈ 1.000 vs $\approx 1.000.000$)

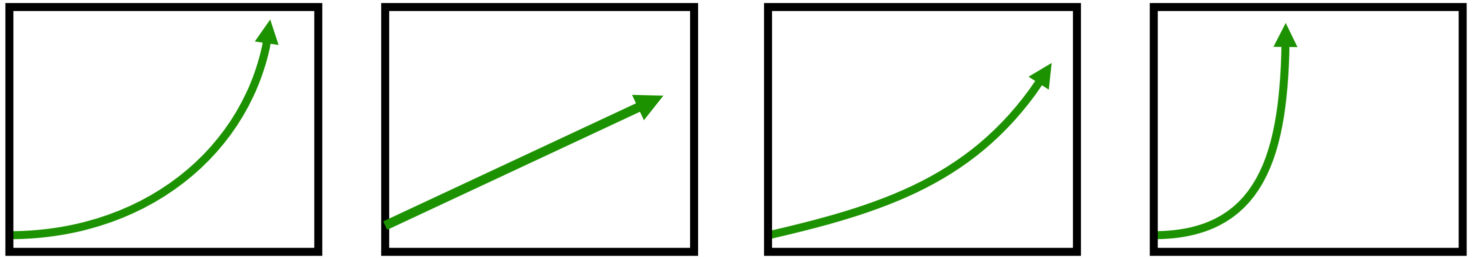
Asymptotic Order of Growth

We will focus on the asymptotic order of growth of the complexity T(n)

So T(n)=$30n^2 + 7n + 10$ will become T(n)= $\theta(n^2)$

We just want to differentiate T(n)=

$\sim n^2$ vs ~n vs ~n log n vs $\sim 2^n$ ....



Asymptotic Order of Growth

Actually, to compute the asymptotic order of growth of T(n) we will compute upper and lower bounds for T(n):

Ex: T(n) grows at most (not faster) like n^2
T(n) grows at least like n^2

⬇

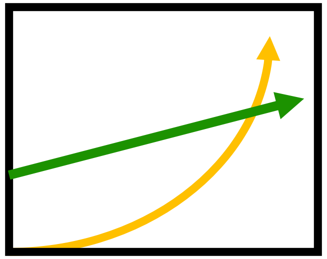
T(n) grows just like n^2

Asymptotic Order of Growth: Upper Bounds

Upper bounds

Informal: T(n) is O(f(n)) if T(n) grows with at **most** the same order of magnitude as f(n) grows:

$$T(n) \lesssim f(n)$$



T(n) is O(f(n))

Asymptotic Order of Growth: Upper Bounds

Upper bounds

Formal: T(n) is O(f(n)) if there exist a constant $c > 0$ such that for all $n \geq 1$ we have

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

Equivalent: T(n) is O(f(n)) if there exists c > 0 such that

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} \leq c$$

Asymptotic Order of Growth: Upper Bounds

Exercise 1: $T(n) = 32n^2 + 17n + 32$.

Say if T(n) is:

- $O(n^2)$?
- $O(n^3)$?
- $O(n)$?

Asymptotic Order of Growth: Upper Bounds

Exercise 1: $T(n) = 32n^2 + 17n + 32$.

Say if T(n) is:

- $O(n^2)$? Yes
- $O(n^3)$? Yes
- $O(n)$? No

Solution: To show that T(n) is $O(n^2)$ we can:

- Use the first definition with c = 1000
- Use limits: $\lim_{n \to \infty} \frac{T(n)}{n^2} = 32$, which is a constant

Asymptotic Order of Growth: Upper Bounds

Exercise 2:

- $T(n) = 2^{n+1}$, is it $O(2^n)$?
- $T(n) = 2^{2n}$, is it $O(2^n)$?

Asymptotic Order of Growth: Upper Bounds

Exercise 2:

- T(n) = 2^{n+1}, is it O(2^n) ? Yes
- T(n) = 2^{2n}, is it O(2^n) ? No

Solution (second item): $\lim_{n\to\infty} \frac{T(n)}{2^n} = \lim_{n\to\infty} 2^n = \infty$ is not constant

Solution 2 (second item): To have $2^{2n} < c.2^n$ we need $c > 2^n$. So c is not a constant

Upper Bounds Involving log/exp

Logarithms. $\log_a n$ is $O(\log_b n)$ for any constants $a, b > 0$

can avoid specifying the base

Logarithms. For every constant $d > 0$, $\log n$ is $O(n^d)$

log grows slower than every polynomial

Exponentials. For every constants $r > 1$ and $d > 0$, n^d is $O(r^n)$

every exponential grows faster than every polynomial

Upper Bounds Involving log/exp

Exercise: is T(n) = 21*n*log n

- O(n^2) ?
- O($n^{1.1}$) ?
- O(n) ?

Upper Bounds Involving log/exp

Exercise: is T(n) = 21*n*log n

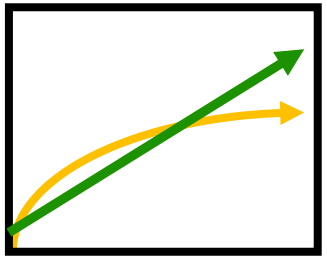
- O(n^2) ? Yes
- O($n^{1.1}$) ? Yes
- O(n) ? No

Solution (first item): Comparing 21*n*log n vs. n^2 is the same as comparing 21*log n vs. n, and we know log n grows slower than n

Solution 2 (first item): $\lim_{n\to\infty} \frac{T(n)}{n^2} = \lim_{n\to\infty} \frac{21 \log n}{n}$, which is at most a constant since log n grows slower than n

Lower Bounds

Informal: T(n) is Ω(f(n)) if T(n) grows with at **least** the same order of magnitude as f(n) grows



Formal: T(n) is Ω(f(n)) if there exist constants c > 0 such that for all n we have T(n) ≥ c · f(n).

Equivalent: T(n) is Ω(f(n)) if there exist constant c>0

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} \geq c$$

Tight Bounds

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$

T(n) grows at most as fast as f(n) — $T(n)$ is $O(f(n))$

T(n) grows at least as fast as f(n) — $T(n)$ is $\Omega(f(n))$

⬇

T(n) grows just like f(n) — $T(n)$ is $\Theta(f(n))$

Lower and Tight Bounds

Exercise: $T(n) = 32n^2 + 17n + 32$

Is T(n):

- $\Omega(n)$?
- $\Omega(n^2)$?
- $\Theta(n^2)$?

- $\Omega(n^3)$?
- $\Theta(n)$?
- $\Theta(n^3)$?

Lower and Tight Bounds

Exercise: $T(n) = 32n^2 + 17n + 32$

Is T(n):

- $\Omega(n)$?
- $\Omega(n^2)$?
- $\Theta(n^2)$?

- $\Omega(n^3)$?
- $\Theta(n)$?
- $\Theta(n^3)$?

Lower and Tight Bounds

Exercise: $T(n) = 32n^2 + 17n + 32$

Is T(n):

- $\Omega(n)$? Yes
- $\Omega(n^2)$? Yes
- $\Theta(n^2)$? Yes

- $\Omega(n^3)$? No
- $\Theta(n)$? No
- $\Theta(n^3)$? No

Solution (second item): $\lim_{n\to\infty} \frac{T(n)}{n^2} = 32$ is constant > 0

Solution 2 (second item): To show T(n) is $\Omega(n^2)$ use c = 1

Back to algorithms

```
Func Algorithm 1(A)        #A is array of bits
  x=20
  For i=1...len(A)
      x=3x
```

Q: What is asymptotic time complexity $T(n)$ of this algorithm?

A: $T(n) = \Theta(n)$

- Just notice/remember $T(n) \approx 2n + 1$

Back to algorithms

```
Func Algorithm 1(A)        #A is array of bits
    x=20
    For i=1...len(A)
        x=3x
```

Q: What is asymptotic time complexity $T(n)$ of this algorithm?

A: $T(n) = \Theta(n)$

- Input A of bit-size *n* has *n* entries, so *n* iterations of **for**
- The algorithm makes at most 10n operations ⇒ $T(n)$ is $O(n)$
- The algorithm makes at least n operations ⇒ $T(n)$ is $\Omega(n)$
- So $T(n) = \Theta(n)$

Back to algorithms

```
Func Find10(A)            #A is array of 32-bit numbers
    For i=1 to len(A)
        If A[i]==10
            Return i
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) = \Theta(n)$

- Remember need to look at worst instance to get T(n)
- Notice/remember that $T(n) \approx (n/32) + 1$

Back to algorithms

```
Func Find10(A)            #A is array of 32-bit numbers
    For i=1 to len(A)
        If A[i]==10
            Return i
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) = \Theta(n)$

- Remember need to look at worst instance to get T(n)
- Worst instance: the only "10" is in the last position
- A of bit-size *n* has *n/32* numbers (using formal definition)
- The algorithm makes at most 5n operations $\Rightarrow$ $T(n)$ is $O(n)$
- The algorithm makes at least n/32 operations (remember worst instance) $\Rightarrow$ $T(n)$ is $\Omega(n)$
- So $T(n) = \Theta(n)$

Complexity of Algorithm vs Complexity of Problem

There are many different algorithms for solving the same problem

Showing that an algorithm is $\Omega(n^3)$ does not mean that we cannot find another algorithm that solves this problem faster, say in $O(n^2)$

Exercises

Exercicio 1. Analise a complexidade de pior caso do algoritmo abaixo. Ou seja, encontre uma funcao $f(n)$ tal que $T(n) = \Theta(f(n))$. Justifique.

Pseudo1 (A) #A é vector com n bits

```
t ← 0
Cont ← 1
Para i=1 até n
    Cont ← cont+1
Fim Para
Enquanto cont ≥ 1
    Cont ← cont/2
    Para j = 1 a n
        t++
    Fim Para
Fim Enquanto
```

- t ← 0, Cont ← 1: cst
- Para i=1 até n … Fim Para: cst*n
- Para j = 1 a n … Fim Para: cst*n
- Enquanto … Fim Enquanto: log(n) iterations * [cst*n per iteration] = cst*n*log(n)

Solucao: O algoritmo é $\Theta(n \log n)$

Exercises

Exercício 2. Considere um algoritmo que recebe um número real x e o vetor $(a_0, a_1, \ldots, a_{n-1})$ como entrada e devolve

$$a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$$

a) Desenvolva um algoritmo para resolver este problema que execute em tempo **quadrático**. Faça a análise do algoritmo

b) Desenvolva um algoritmo para resolver este problema que execute em tempo **linear**. Faça a análise do algoritmo

Exercises

Solução Exercício 2

a)

sum = 0
Para i= 0 até n-1 **faça**
    aux ← a_i
    Para j:=1 até i
        aux ← x . aux
    Fim Para
    sum ← sum + aux
Fim Para
Devolva sum

- Análise

Número de operações elementares é igual a

$$1+2+3+ \ldots + n-1 = n(n-1)/2 = O(n^2)$$

Exercises

b)

sum = a_0

pot = 1

Para i= 1 até n-1 **faça**

pot ← x.pot

sum ← sum + a_i.pot

Fim Para

Devolva sum

- Análise

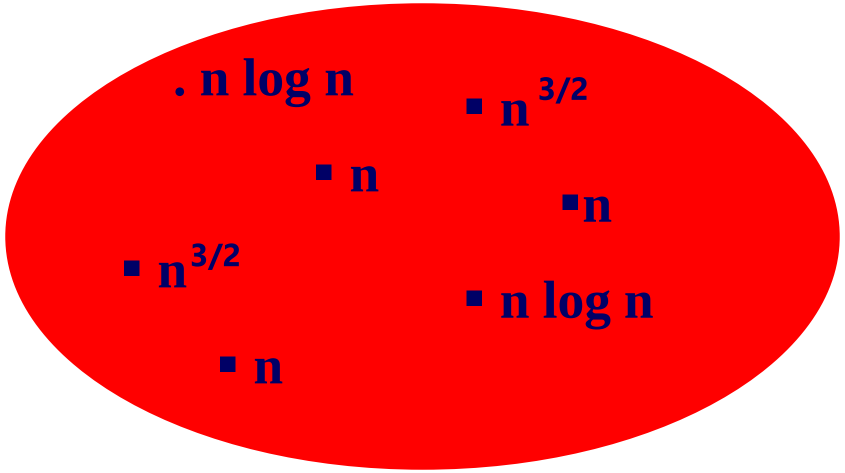
A cada loop são realizadas O(1) operações elementares. Logo, o tempo é linear

Exercises

Exercícios Kleinberg & Tardos, cap 2 da lista de exercícios

A high-level view

inputs of size n
for algorithm A
(cartoon)

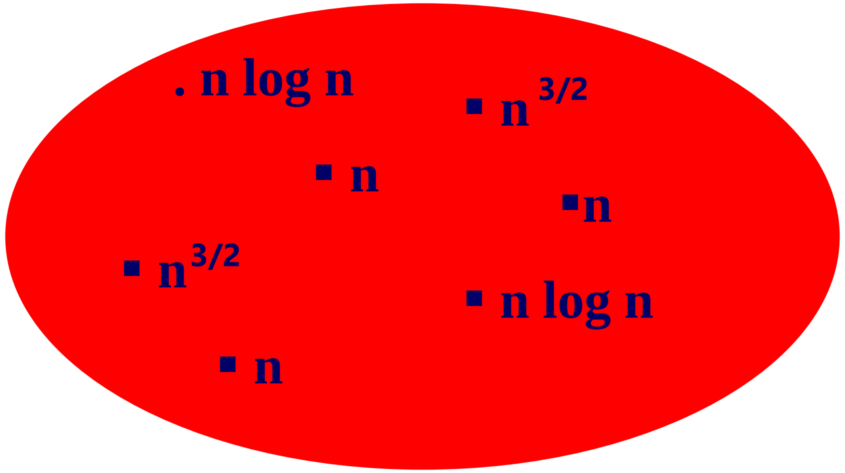


Can we say that the time complexity of A is?

- O(n^2) ?
- Ω(n^2) ?
- Ω (n) ?
- O (n) ?
- Ω ($n^{3/2}$) ?

A high-level view

inputs of size n
for algorithm A
(cartoon)



Can we say that the time complexity of A is?

- O(n^2) ? Yes, beccause largest complexity of algorithm is at most n^2
- Ω(n^2) ? No, there is no input where the complexity of the algorithm has order n^2
- Ω (n) ? Yes
- O (n) ? No, there are inputs where complexity has larger order
- Ω ($n^{3/2}$) ? Yes

What does asymptotic analysis give us?

Does not tell that exact constants in the time-complexity of an algo

Does give a good basis of comparison between algorithms

- Even if optimize implementation of $\Theta(n^2)$ algorithm and make it 10x faster, it is probably much slower than a "bad" implementation of a $\Theta(n)$ algorithm (for large instances)

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Polynomial Time

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Ex: $T(n) = 32n^2$ and $T(n) = n \log n$ are polynomial time

We consider an algorithm efficient if time-complexity is polynomial

efficient

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

A First Analysis of Recursive Algorithms: Binary Search

Binary Search

Problem: Given a sorted list of numbers (increasing order) a1,…an, decide if number x is in the list

```
Function bin_search(A,x)
   n = len(A)
   if n = 1
      if A[1] = x return TRUE
      else return FALSE
   end if

   if x = A[n/2]
      return TRUE
   else if x < A[n/2]
      return bin_search(A[1:n/2], x)
   else if x > A[n/2]
      return bin_search(A[n/2:n], x)
   end if
```

Ex: x=14

1	2	3	5	7	10	14	17

7	10	14	17

14	17

Binary Search

Problem: Given a sorted list of numbers (increasing order) a1,…an, decide if number x is in the list

```
Function bin_search(A,i,j,x)
   if i = j
      if A[i] = x return TRUE
      else return FALSE
   end if

   mid = floor((i+j)/2)
   if x = A[mid]
      return TRUE
   else if x < A[mid]
      return bin_search(A, i, mid-1, x)
   else if x > A[mid]
      return bin_search(A, mid+1, j, x)
   end if
```

```
Function bin_search_main(A, x)
   bin_search(A,1,n,x)
```

Ex: x=14

1	2	3	5	7	10	14	17

7	10	14	17

14	17

Binary Search Analysis

Binary search recurrence: $T(n) \leq$

we will always ignore floor/ceiling

(the "sorting" slides has one slide that keeps the ceiling, so you can see that it works ok)

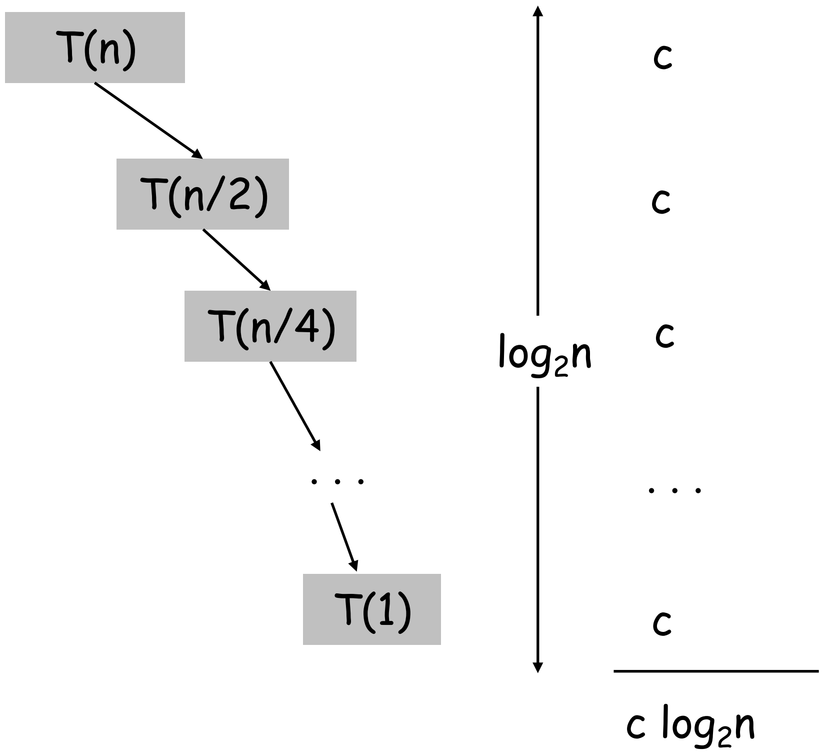
Binary Search Analysis

Binary search recurrence: $T(n) \le c + T\left(\frac{n}{2}\right), \quad T(1) \le c$

Claim: The time complexity T(n) of binary search is at O(log n)

Proof 1: $T(n) \le c + T(n/2) \le c + c + T(n/4) \le \ .... \ \le c + c + ... + T(1) \le \underbrace{c + c + ... + c}_{\text{log n terms}}$

Recursion tree:



Binary Search Analysis

Binary search recurrence: $T(n) \leq c + T\left(\frac{n}{2}\right)$, $T(1) \leq c$

Claim: The time complexity T(n) of binary search is at most O(log n)

Proof 2: (induction) Base case: n=1

Now suppose that for n′ ≤ n – 1, $T(n') \leq c * \log(n')$

Then T(n) ≤ c + T(n/2) ≤ c + c*log(n/2) = c + c*(log n – 1) = c*log n

Recursive Algorithms

Exercício 2. Projete um algoritmo (recursivo) que receba como entrada um numero real x e um inteiro positivo n e devolva x^n. O algoritmo deve executar O(log n) somas e multiplicações