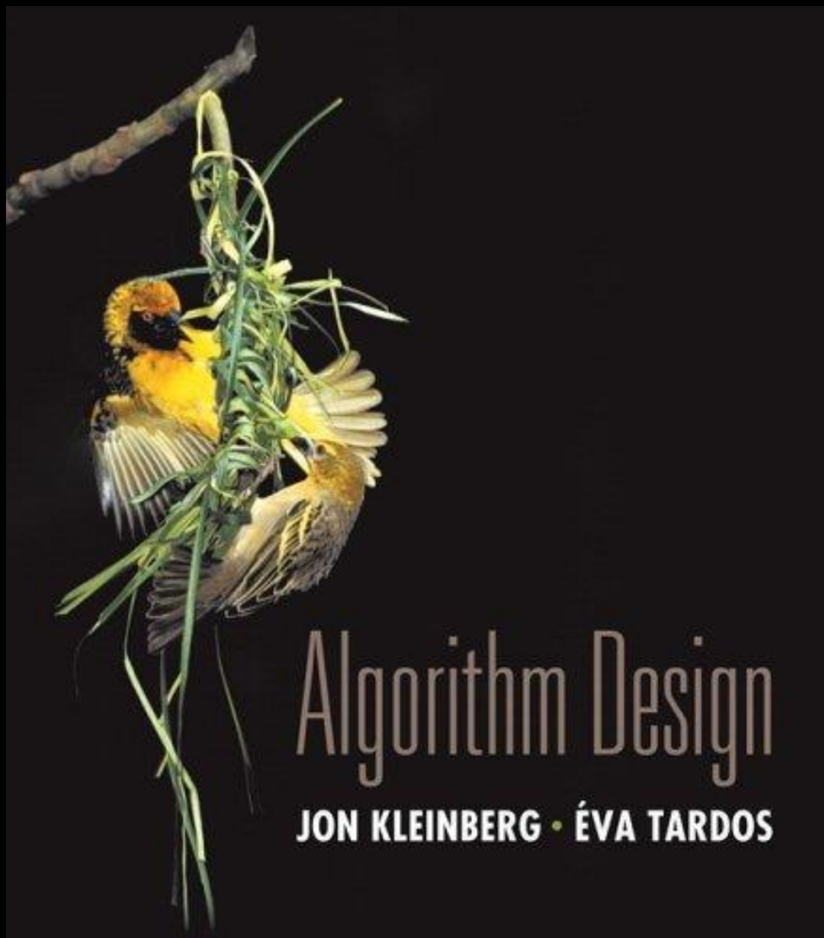


Chapter 2

Basics of Algorithm Analysis

2.1 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

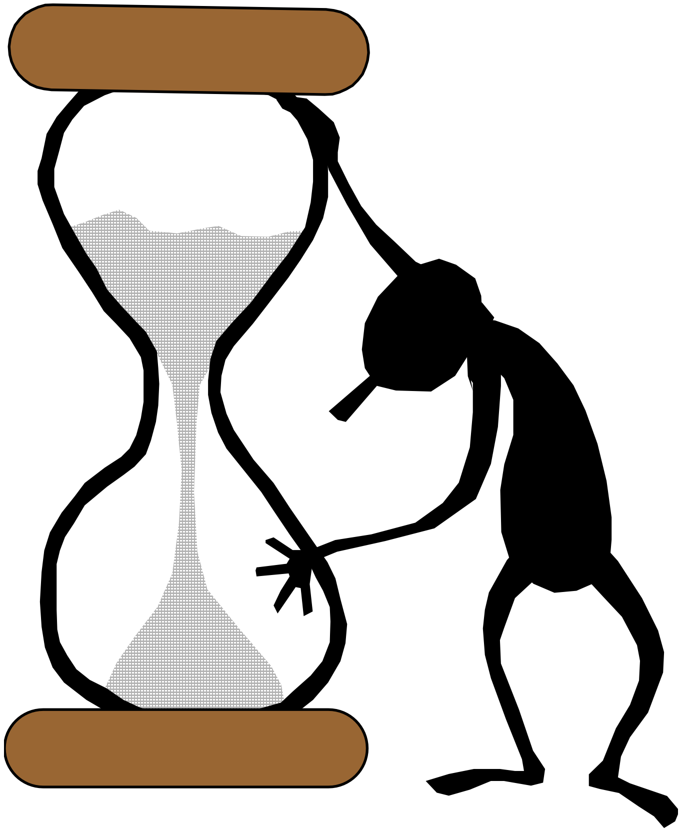
Time complexity is a function: Specifies how the running time depends on the size of the input.

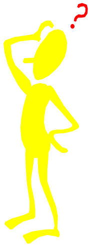
Function mapping

"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:

- # of bits: 17 bits - Formal
- # of digits: 5 digits - Reasonable: #bits and #digits are always within constant factor $\approx \log_2 10 \approx 3.32$
- Value: 83920

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

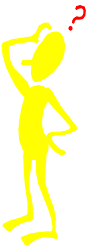
Ex 2:

10

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

- # of elements = 10

Is this reasonable?



Size of input instance

Ex 2:

10

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

- # of elements = 10 - Reasonable: if each number is stored into, say, into 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

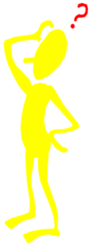
# of bits n to represent input

↓

# 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

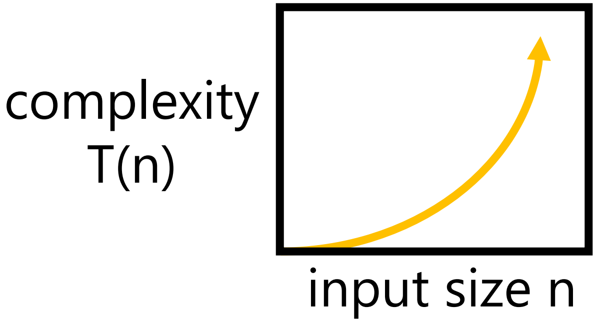
Worst-case running time. Consider the instance where the algorithm uses largest number of basic operations

- 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
- ≈ 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 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
- $\approx$ 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**

2.2 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 and algorithm with complexity

T(n) = 9n+6?

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 have seen, 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)=$30n^2 + 10$ will become T(n)= $\theta(n^2)$

We just want to differentiate T(n)=

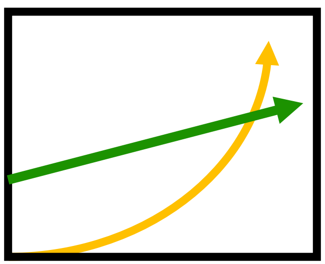
$\sim n^2$ vs ~n vs ~n log n vs $\sim 2^n$ ....

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

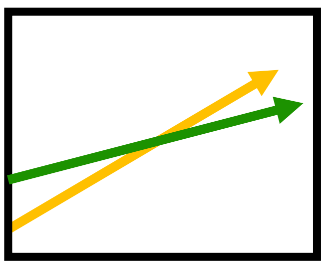
Asymptotic Order of Growth

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) is O(f(n))



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

both grow at same order of magnitude

Asymptotic Order of Growth

Upper bounds

Formal: T(n) is O(f(n)) if there exist a constant $c \geq 0$ 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 $c \geq 0$ such that

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

Asymptotic Order of Growth

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

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

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

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

Asymptotic Bounds for Some Common Functions

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

can avoid specifying the base

Logarithms. For every x > 0, log n is $O(n^x)$

log grows slower than every polynomial

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

every exponential grows faster than every polynomial

Asymptotic Bounds for Some Common Functions

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

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

Asymptotic Bounds for Some Common Functions

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

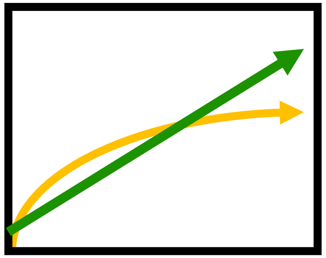
- O(n^2) ? Yes
- O($n^{1.1}$) ? Yes
- O(n) ? No

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 $T(n) \geq c \cdot f(n)$.

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 $\Omega(f(n))$.

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 $\Rightarrow$ $T(n)$ is $O(n)$
- The algorithm makes at least n operations $\Rightarrow$ $T(n)$ 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
```

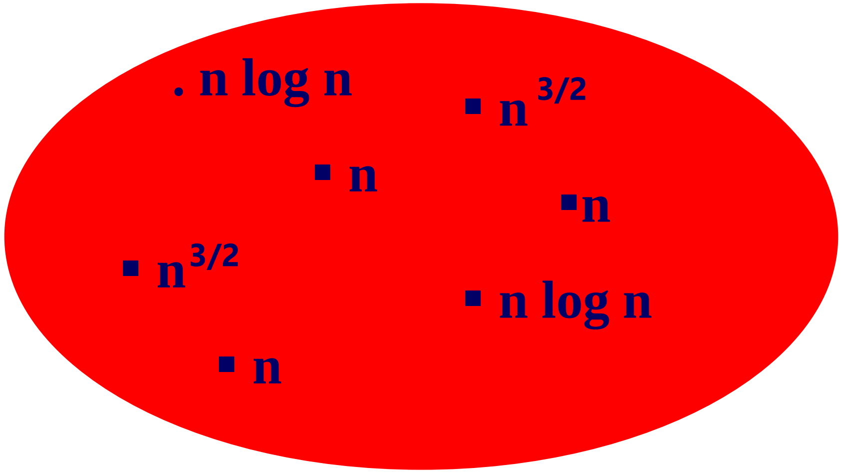
Q: What is the time complexity $T(n)$ of this algorithm?

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
- The algorithm makes at most 5n operations ⇒ $T(n)$ is $O(n)$
- The algorithm makes at least n operations (remember worst instance) ⇒ $T(n)$ is $\Omega(n)$
- So $T(n) = \theta(n)$

Back to algorithms

inputs of size n
for algorithm A
(cartoon)

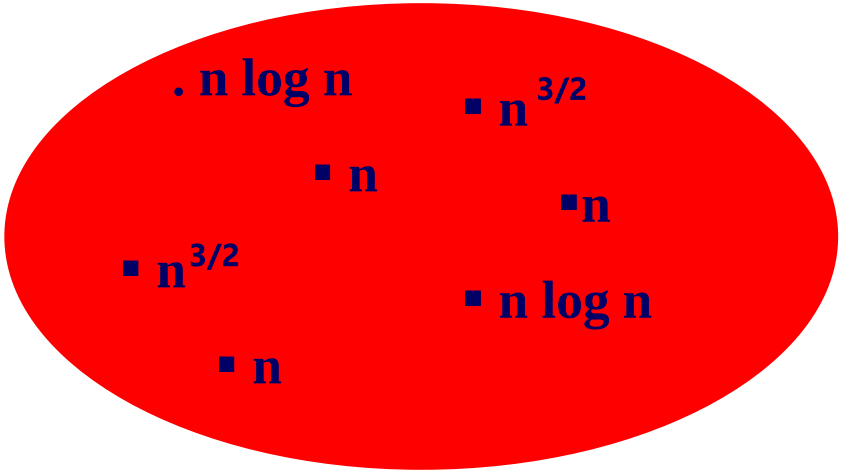


Can we say that the time complexity of A is?

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

Back to algorithms

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

Implication of Asymptotic Analysis

Hypothesis

- Basic operations (addition, comparison, shifts etc) takes at least 10ms and at most 50ms seconds

Algorithms

- Algorithm A executes 20n operations for the worst instance (O(n))
- Algorithm B executes n^2 operations for the worst instance $(\Omega(n^2))$

Conclusion

- For a instance of size n, A spends **at most** 1000n ms
- For the worst instance of size n, B spends **at least** 10 n^2 ms
- For n>100, A is faster than B in the worst case, regargless of which operations they execute

Allows us to tell which algorithm is faster (for large instances)

Exercises

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

```
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
```

Exercises

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

```
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

Exercícios Kleinberg & Tardos, cap 2 da lista de exercícios

2.4 A Survey of Common Running Times

Linear Time: O(n)

Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Remark. For all instances the algorithm executes a linear number of operations

Linear Time: O(n)

Linear time. Running time is at most a constant factor times the size of the input.

Finding an item x in a list. Test if x is in the list $a_1, \ldots, a_n$

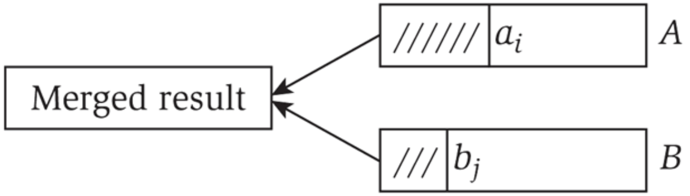
```
Exist ← false
for i = 1 to n {
   if (a_i== x)
      Exist ← true
      break

}
```

Remark. For some instances the algorithm is sublinear (e.g. x in the first position)

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_k$ with $B = b_1, b_2, \ldots, b_k$ (increasing order) into sorted whole.




```
i = 1, j = 1
while (I <= |A| and j <= |B|) {
   if (aᵢ ≤ bⱼ) append aᵢ to output list and increment i
   else        append bⱼ to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size k takes O(n) time (n=total size=2k).

Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the distance of the closest pair.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n-1 {
   for j = i+1 to n {
      d ← (xi - xj)^2 + (yi - yj)^2
      if (d < min)
         min ← d
   }
}
```

← don't need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← see chapter 5

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Let S_1, …, S_n be subsets of {1, 2, …, n}. Is there a disjoint pair of sets?

Set Representation. Assume that each set is represented as an incidence vector.

n=8 and S={2,3,6}, S is represented by (0,1,1,0,0,1,0,0)

n=8 and S={1,4}, S is represented by (1,0,0,1,0,0,0,0)

Algorithm:

```
For i=1…n-1
        For j=i+1…n
                If Disjoint(i, j)
                    Return 'There are disjoint sets'
                End If
        End For
End For
Return 'There are no disjoint sets'
```

Disjoint(i, j):

```
k←1
While k<=n
    If S_i(k)=S_j(k)=1 Return False
    k++
End While
Return True
```

Cubic Time: $O(n^3)$

1. A complexidade de tempo do algoritmo é $O(n^3)$?

2. A complexidade de tempo do algoritmo é $\Omega(n^3)$?

Cubic Time: $O(n^3)$

1. A complexidade de tempo do algoritmo é $O(n^3)$? SIM

2. A complexidade de tempo do algoritmo é $\Omega(n^3)$? SIM

"Bad" instance: all sets are equal to {n} => algoritm makes $\Omega(n^3)$ basic operations

Exponential Time

Independent set. Given a graph, find the largest independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← ϕ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

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

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice…
- … but almost always the poly-time algorithms people develop have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Polynomial Time

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	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

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 $O(n^2)$

Exercício

Exercício 1. 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

Exercício - Solução

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

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

Exercício - Solução

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

2.5 A First Analysis of a Recursive Algorithm: Binary Search

Binary Search

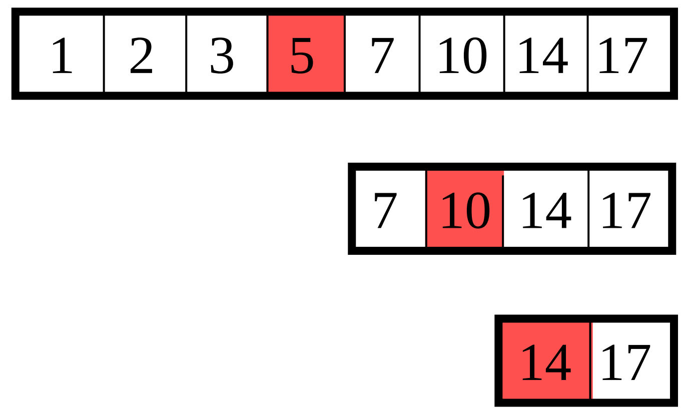
Problem: Given a sorted list of numbers (increasing order) a1,...an, decide if number x is in the list

```
Function bin_search(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 x
   else if x < a_mid
      return bin_search(i, mid-1, x)
   else if x > a_mid
      return bin_search(mid+1, j, x)
   end if
```

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

Ex: x=14



Binary Search Analysis

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

we will always ignore floor/ceiling

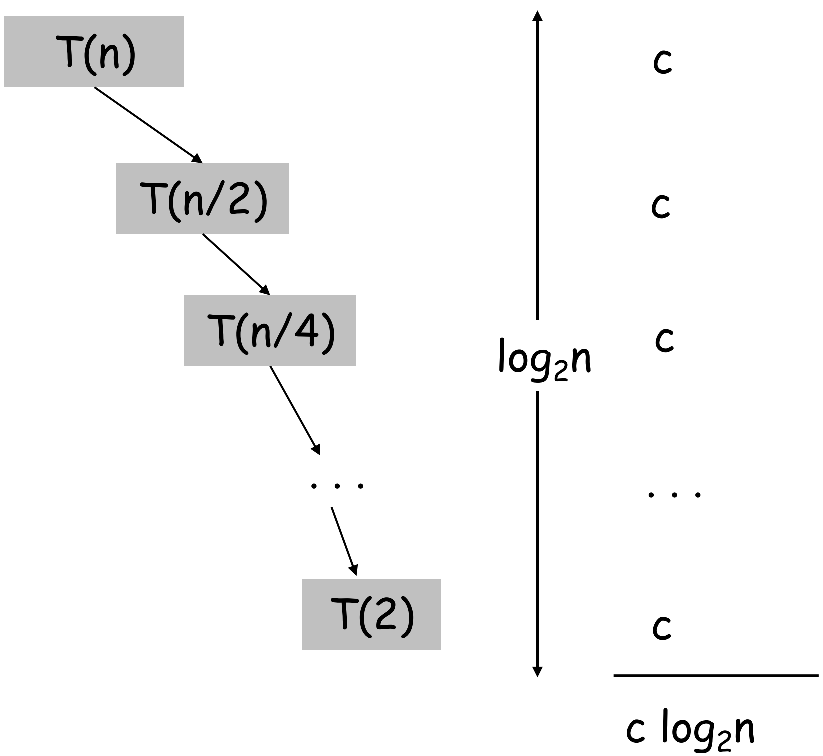
(the "sorting" slides has one slide that keeps the ceiling, so you can see that it works)

Binary Search Analysis

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

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

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



Binary Search Analysis

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

Claim: The time complexity T(n) of binary search is at most c*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

Recursive Algorithms

```
Proc  Pot(x,n)
      Se n=0 return 1
      Se n=1 return x
      Se n é par
          tmp←Pot(x,n/2)
          Return tmp*tmp
      Senão n é ímpar
          tmp←Pot(x,(n-1)/2)
           Return x*tmp*tmp
      Fim Se
Fim
```

Análise:

$T(n) = c + T(n/2) \Rightarrow T(n)$ é $O(\log n)$