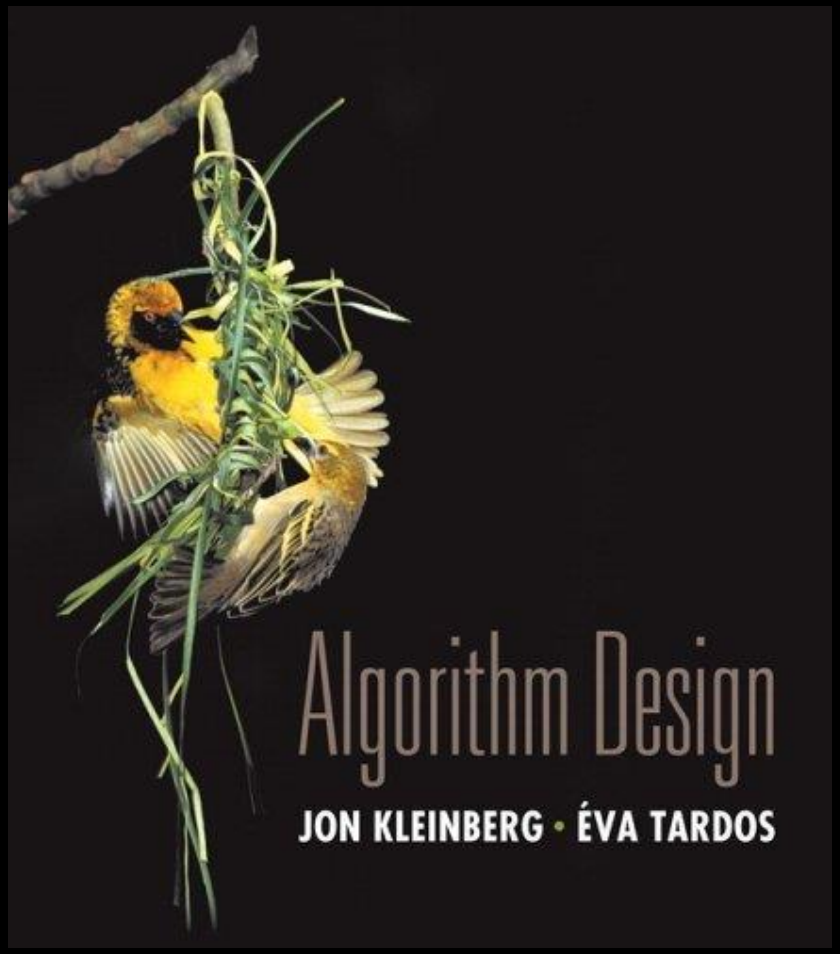


Chapter 2

Basics of Algorithm Analysis

The Time Complexity of an Algorithm

Specifies how the running time depends on the size of the input.

Purpose?

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 help focus on the parts of code that are executed the largest number of times.
- To choose an algorithm for an application.

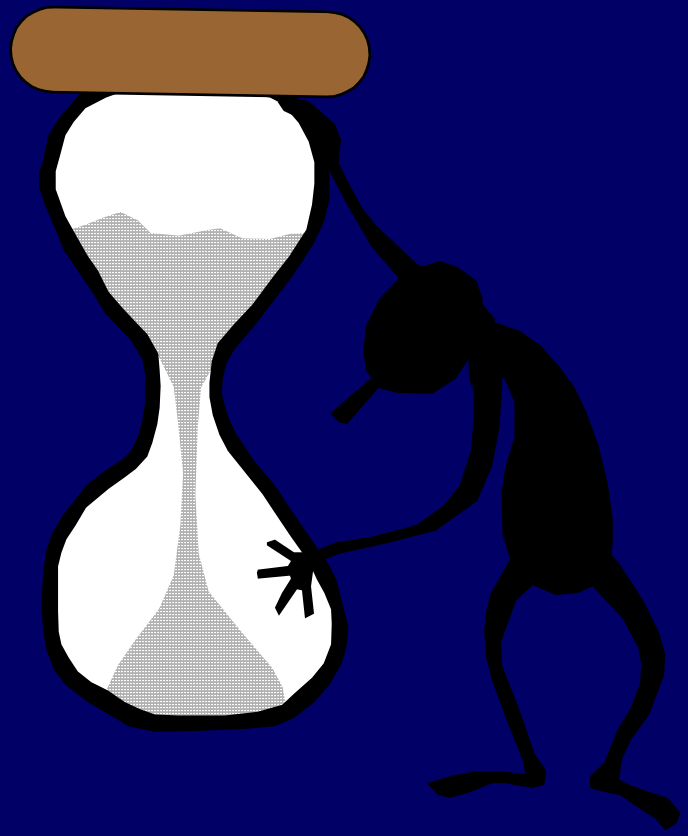
Time Complexity Is a Function

- Specifies how the running time depends on the size of the input.

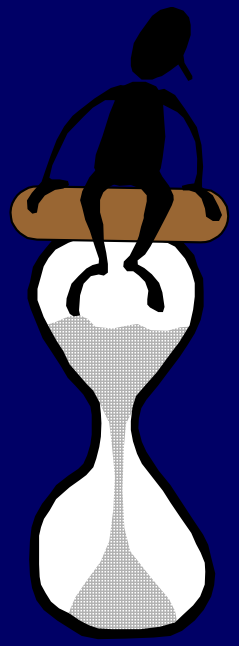
- A function mapping
  - "size" of input

  - "time" T(n) executed .

Definition of Time?



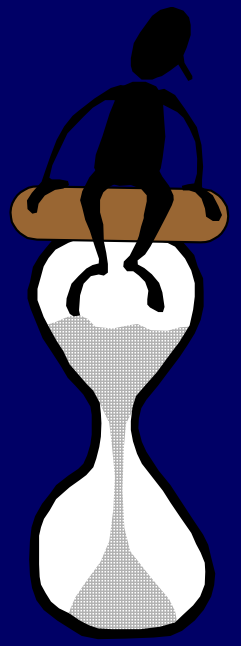
Definition of Time



- # of seconds (machine dependent).
- # lines of code executed.
- # of times a specific operation is performed (e.g., addition, comparison).

- Which?

Definition of Time

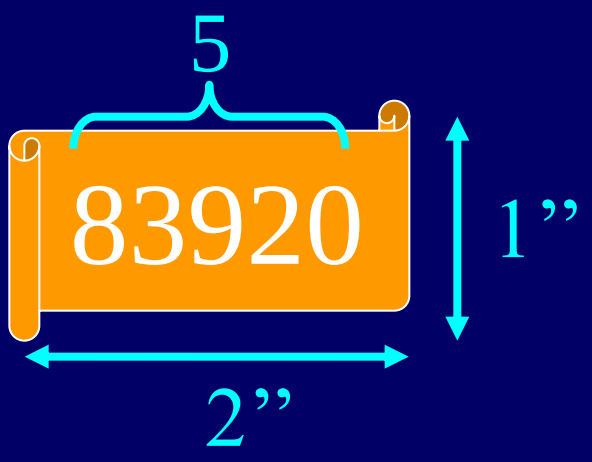


- # of seconds (machine dependent).
- # lines of code executed.
- # of times a specific operation is performed (e.g., addition).

- These are all reasonable definitions of time, because they are within a constant factor of each other.

Size of Input Instance?

Size of Input Instance

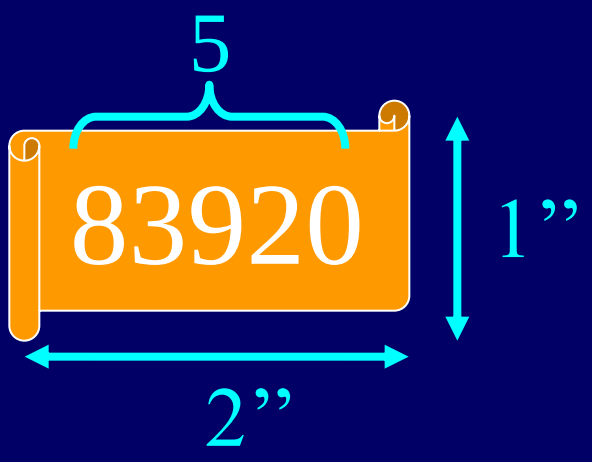


- # of bits
- # of digits
- Value

- - n = 17 bits
- - n = 5 digits
- - n = 83920

Which are reasonable?

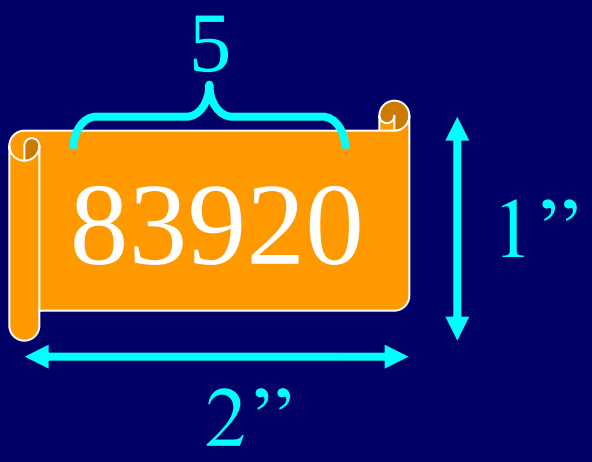
Size of Input Instance



- # of bits
- # of digits
- Value

- - n = 17 bits
- - n = 5 digits
- - n = 83920

Size of Input Instance

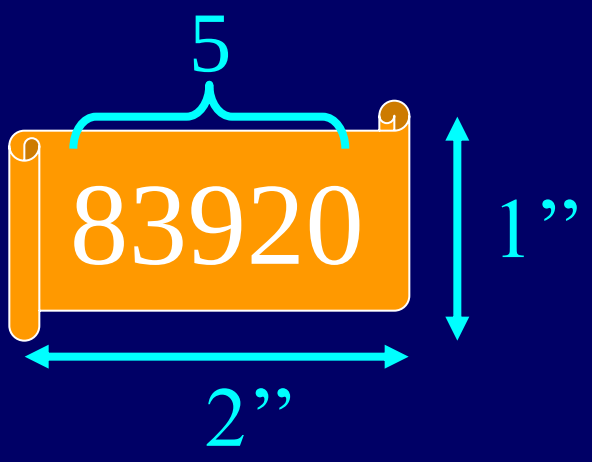


- # of bits
- # of digits
- Value

- - n = 17 bits
- - n = 5 digits
- - n = 83920

- Formal

Size of Input Instance



- # of bits
- # of digits
- Value

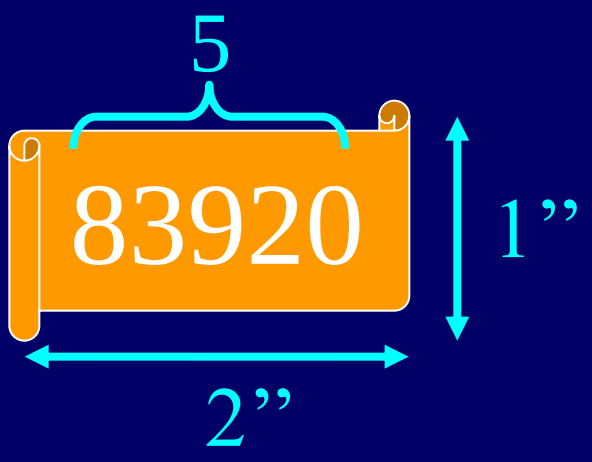
- - n = 17 bits
- - n = 5 digits
- - n = 83920

- Formal
- Reasonable

  # of bits =

  3.32 * # of digits

Size of Input Instance



- # of bits
- # of digits
- Value

- - n = 17 bits
- - n = 5 digits
- - n = 83920

- Formal
- Reasonable
- Unreasonable

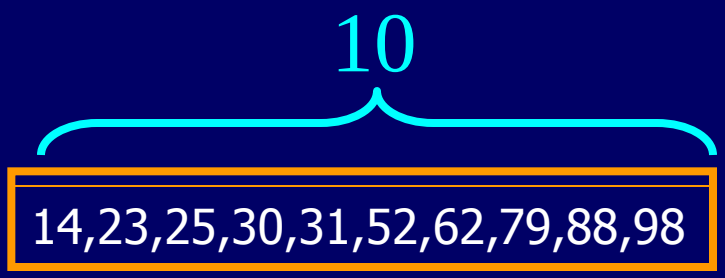
# of bits = $\log_2(\text{Value})$

Value = $2^{\text{\# of bits}}$

Size of Input Instance?

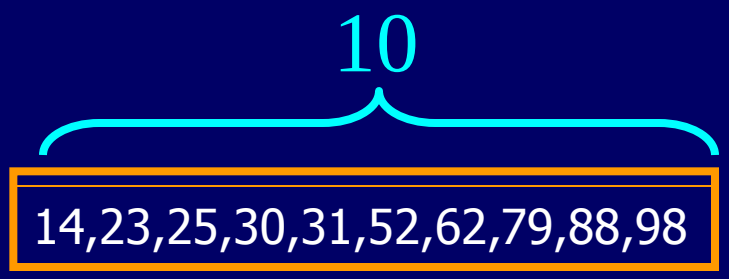
14,23,25,30,31,52,62,79,88,98

Size of Input Instance



- # of elements - n = 10 elements

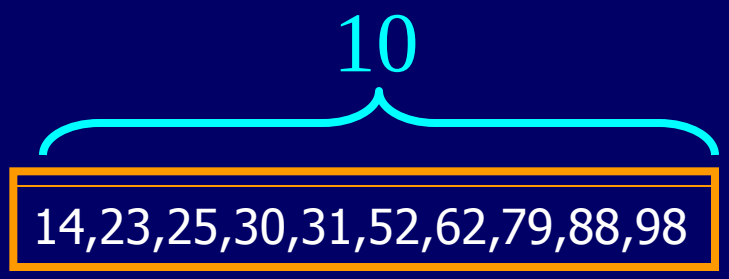
Size of Input Instance



- # of elements - n = 10 elements

Is this reasonable?

Size of Input Instance



- # of elements - n = 10 elements ~ Reasonable

If each element has c bits

# of bits = c * # of elements

Time Complexity Is a Function

- Specifies how the running time depends on the size of the input.

- A function mapping
  - # of bits n needed to represent the input

  - # of operations T(n) executed .

Time Complexity Is a Function

- Common use
  - The input has n elements
  - Each element fits in a word memory

Which Input of size n?

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

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Draconian (pessimistic) view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

Worst-Case Polynomial-Time

We consider an algorithm efficient if its running time 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.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method

Why It Matters

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

2.2 Asymptotic Order of Growth

Asymptotic Order of Growth

Prog 1

```
X←rand(100)
For i=1...N
        j←j+1
        If  x>50 then
                x←x+3
        End If
 End For
```

Análise de pior caso

- 3N +1 atribuições
- N comparações
- 2N adições

Total 6N+1 operações elementares

Asymptotic Order of Growth

Prog 2

```
X←rand(20)
For i=1...N
        x←3x
 End For
```

Análise de pior caso

- 2N +1 atribuições
- N multiplicações

Total 3N+1 operações elementares

Podemos afirmar que Prog2 vai executar mais rápido do que Prog1 ?

Asymptotic Order of Growth

- Podemos afirmar que a instância mais lenta de Prog2 vai executar mais rápido do que a instância mais lenta Prog1?

  - Não, isso vai depender dos tempos de executar comparações, multiplicações, adições e atribuições na máquina

Não vale a muita a pena complicar a metodologia estimando as constantes

Asymptotic Order of Growth

Upper bounds.

T(n) is O(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

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

Lower bounds.

T(n) is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

T(n) is $\Omega(f(n))$ if there exist constant $c>0$

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

Asymptotic Order of Growth

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

Asymptotic Order of Growth

Exercício: $T(n) = 32n^2 + 17n + 32$.

- Responda se T(n) é
- $O(n^2)$?
- $O(n^3)$?
- $\Omega(n)$?
- $\Omega(n^2)$?
- $O(n)$?
- $\Theta(n^2)$?
- $\Omega(n^3)$?
- $\Theta(n)$?
- $\Theta(n^3)$?

Asymptotic Order of Growth

Exercício: $T(n) = 32n^2 + 17n + 32$.

- Responda se T(n) é
- $O(n^2)$? Sim
- $O(n^3)$? Sim
- $\Omega(n)$? Sim
- $\Omega(n^2)$? Sim
- $O(n)$? não
- $\Theta(n^2)$? sim
- $\Omega(n^3)$? não
- $\Theta(n)$?
- $\Theta(n^3)$?

Asymptotic Order of Growth

Solução: $T(n) = 32n^2 + 17n + 32$.

- Para provar que $T(n)$ é $O(n^2)$, considere $c = 33$ e $n_0 = 19$
- Para provar que $T(n)$ é $\Omega(n^2)$, considere $c = 1$ e $n_0 = 0$
- Como $T(n)$ é $O(n^2)$ e $T(n)$ é $\Omega(n^2)$ então $T(n)$ é $\Theta(n^2)$
- Como $T(n)$ é $\Theta(n^2)$ então $T(n)$ é $O(n^3)$ e $\Omega(n)$ e não é $O(n)$, $\Omega(n^3)$, $\Theta(n^3)$ e $\Theta(n)$

Asymptotic Order of Growth

Exercício: Responda se :

- 2^{n+1} é $O(2^n)$?
- 2^{2n} é $O(2^n)$?

Asymptotic Order of Growth

Solucão:

- 2^{n+1} é $O(2^n)$. Considere $c=2$ e $n_0>0$
- 2^{2n} não é $O(2^n)$. De fato, para $2^{2n} < c.2^n$, devemos ter $c>2^n$. Logo, c não é constante

Asymptotic Order of Growth

Exercício: Mostre que log(n!) é Θ(nlog n)

Asymptotic Order of Growth

Solução:

- Primeiro mostramos que log (n!) = O(n log n)

$$\log (n!) = \log n + \log (n-1) + \ldots \log 1 < n. \log n,$$

já que a função log é crescente

- Agora mostramos que log (n!) = Ω(n log n)

$$\log (n!) = \log n + \log (n-1) + \ldots \log 1 >$$

$$n/2. \log (n/2) = n/2 (\log n - 1)$$

Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Asymmetric:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Asymptotic Order of Growth

Exercício resolvido 1 [Kleinberg & Tardos, cap 2]

Why 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.
- Algorithm B executes n^2 operations for the worst instance.

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

Why asymptotic analysis?

General Conclusion

- $t_A(n)$ and $t_B(n)$ time complexities of algos. A and B
- $t_B(n)$ **is not** $O(t_A(n))$. Then, for

sufficiently large n (n larger than n_0)

the worst instance for A of size n executes faster than the worst instance for B of size n

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 ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```

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 $a_1, \ldots, a_n$

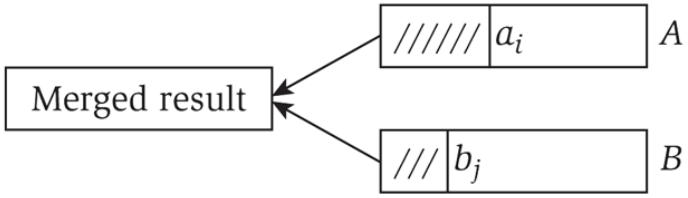
```
Existe ← false
for i = 1 to n {
   if (a_i == x)
      Exitse ← true
      break

}
```

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

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else           append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.

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), …, (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 $S_1, \ldots, S_n$ be subsets of $\{1, 2, \ldots, n\}$. Is there a disjoint pair of sets?

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

- n =8 e S= {2,3,6}, S é representado por (0,1,1,0,0,1,0,0)

  n =8 e S= {1,4}, S é representado por (1,0,0,1,0,0,0,0)

Algoritmo.

```
Para i=1...n-1
        Para j=i+1...n
            Se Disjunto(i, j)
                Return 'Existem conjuntos disjuntos'
            Fim Se
        Fim para
Fim Para
Return 'Não existem conjuntos disjuntos'

Disjunto (i, j)
      k←1
      Enquanto k<=n
          Se Si(k)=Sj (k) =1  return False
          k++
      Fim Enquanto
      Return True
```

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

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

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

   SIM

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

Cubic Time: $O(n^3)$

Mostrar que é $\Omega(n^3)$

Entrada: n vetores com 1 elemento {1,n} {2,n},…,{n-1,n},{n}

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
   }
}
```

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

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

- T(n)= n log n is considered polynomial time

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

can avoid specifying the base

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

log grows slower than every polynomial

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

every exponential grows faster than every polynomial

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

Exercício

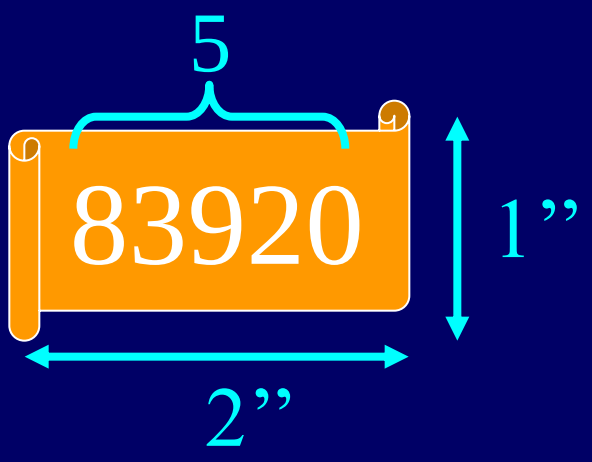
Exercício 2. Projete um algoritmo que recebe com entrada um numero real x e um inteiro positivo n e devolve x^n. O algoritmo deve executar O(log n) somas e multiplicações

Exercício: Solução Recursiva

```
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) => T(n) é O(log n)

Size of Input Instance



- # of bits
- Value

- - n = 17 bits
- n = 83920

- Formal
- Unreasonable

# of bits = $\log_2$(Value)

Value = $2^{\text{\# of bits}}$

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + …

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm:Try N/2, N/3, N/4, …. ?

Time?

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + … Time = N

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm:Try N/2, N/3, N/4, …. ? Time = N

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + … Time = N

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ? Time = N

Is this reasonable?

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + … Time = N

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ? Time = N

No!

One is considered fast and the other slow!

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + …

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ?

Size of Input Instance?

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + … size n = N

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ? size n = log N

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + … size n = N

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ? size n = log N

Time as function of input size?

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + …

  size n = N
  Time = N = n

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ?

  size n = log N
  Time = N = 2^n

Two Example Algorithms

- Sum N array entries:

  A(1) + A(2) + A(3) + …

  size n = N
  Time = N = n

- Factor value N:

  Input N=5917 & Output N=97*61.

  Algorithm N/2, N/3, N/4, …. ?

  size n = log N
  Time = N = 2^n

Linear vs Exponential Time!

Time Complexity of an Algorithm
X
Time Complexity of a Problem

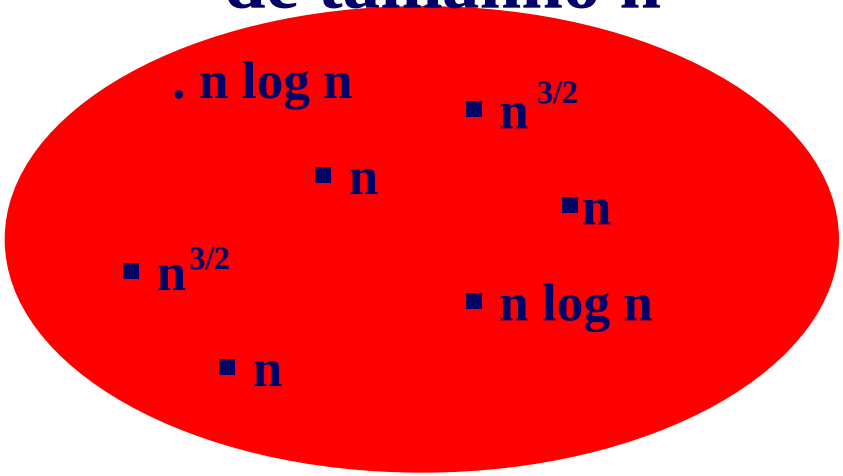
Time Complexity of Algorithm

The time complexity of an algorithm is the largest time required on any input of size n.

- $O(n^2)$: Prove that for every input of size n, the algorithm takes no more than cn^2 time.
- $\Omega(n^2)$: Find one input of size n, for which the algorithm takes at least this much time.
- $\theta(n^2)$: Do both.

Complexidade de Tempo de um Algoritmo

Entradas p/ algoritmo A de tamanho n



Seria correto afirmar que:

•A é O(n^2) ? Sim, pois a maior complexidade do algoritmo é $n^{3/2}$ que é menor que n^2

•A é Ω(n^2) ? Não, pois o algoritmo não possui entradas que possuem esta complexidade.

• A é O($n^{3/2}$) ? Sim

•A é Ω (n) ? Sim

•A é O (n) ? Não, pois tem entradas que gastam mais que isso

•A é Ω ($n^{3/2}$) ? Sim

Time Complexity of Problem

The time complexity of a problem is the time complexity of the **fastest** algorithm that solves the problem.

- $O(n^2)$: Provide an algorithm that solves the problem in no more than this time.
- $\Omega(n^2)$: Prove that no algorithm can solve it faster (Usually hard to prove).
- $\theta(n^2)$: Do both.