

ALGORITMOS

Parte I:

- 1. O que são?**
- 2. O que os caracteriza**

Parte II:

- 3. Algoritmos e computadores**
- 4. O processo de compilação**
- 5. Algoritmos e linguagens de programação**

Parte III:

- 6. Algoritmos resolvendo problemas**
- 7. Algoritmos e correção**
- 8. Resolvem qualquer problema?**
- 9. Adianta executá-los?**
- 10. Nossa ignorância**

- Algoritmo é uma receita para resolução de um problema
- Exemplo:
 - Problema: preparar “bifes à milanesa”
 - Algoritmo: precisamos descrever a receita

“Bife à milanesa”:

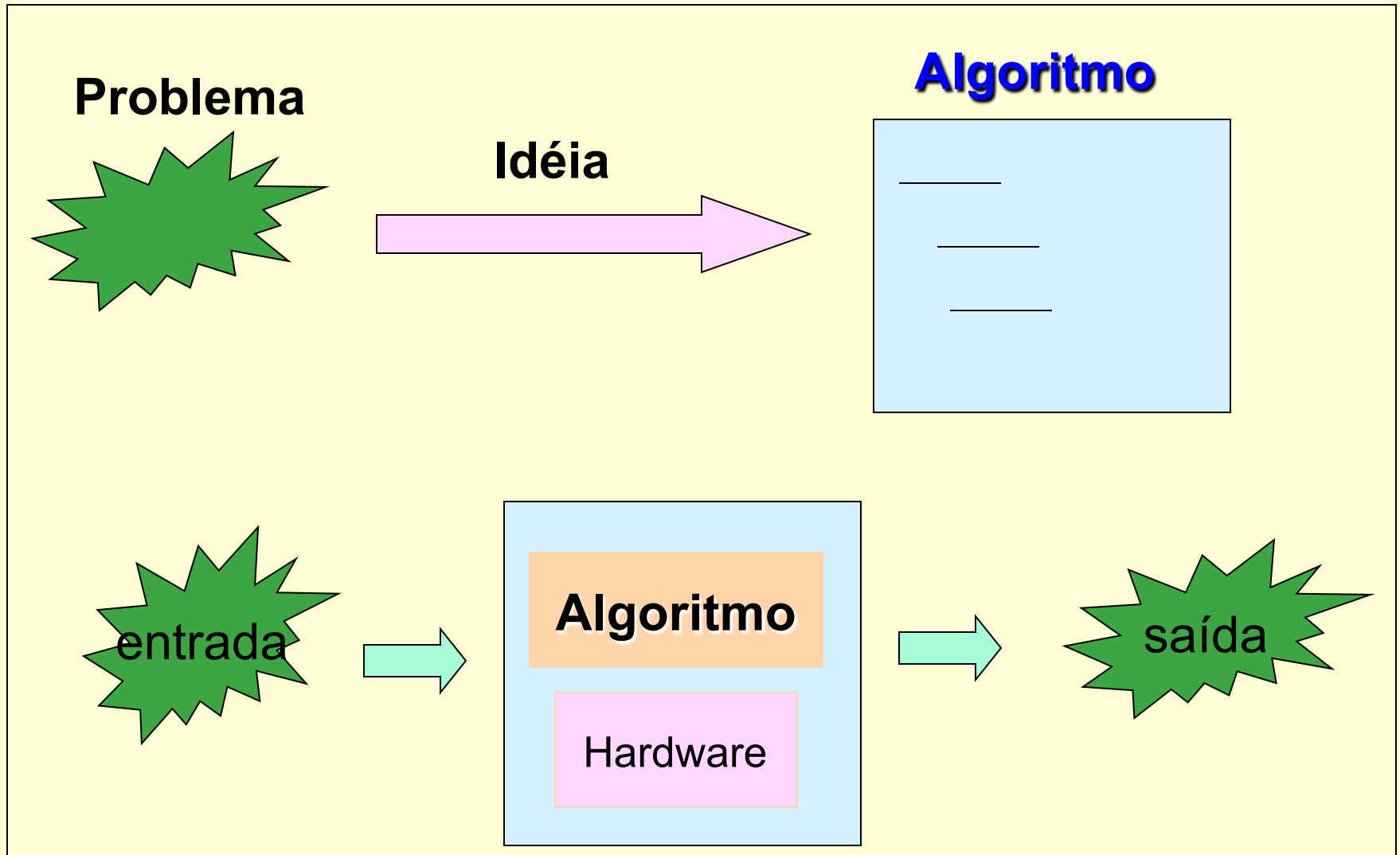
1. Limpar a peça de carne
2. Fatiar a carne em bifes
3. Colocar farinha de rosca em um prato
4. Bater 2 ovos em outro prato
5. Repetir, para cada bife:
 - 5.1) passar cada lado do bife nos ovos;
 - 5.2) passar cada lado do bife na mistura de farinha;
 - 5.3) levar o bife à frigideira;
 - 5.4) aguardar dourar, virando ambas as faces;
 - 5.5) retirar bife e colocar sobre papel toalha até secar;
 - 5.6) retirar do papel toalha e juntar numa travessa;
6. Decorar a travessa com folhas de alface
7. Servir

- Objetos de “consumo” (entrada):
 - carne
 - farinha
 - ovos
 - alface
- Objetos de “apoio” (atores, executores):
 - faca
 - travessa
 - fogão
 - cozinheiro

- Objetos “**produzidos**” (saída):
 - bifes
- Objeto que “**controla**” o processo (receita):
 - **algoritmo**

Algoritmos

O que são?



Algoritmos

O que são?

Um dos primeiros algoritmos:

Euclides (300 . . . 400 BC): algoritmo para obter
o máximo divisor comum de dois inteiros positivos

Século IX (800-899 DC), península arábica/Pérsia:

Matemático **Mohammed al-Khowârizmî**

Cria regras passo-a-passo para se fazer aritmética com algarismos
decimais

Em latim:

al-Khowârizmî  **algorismus** 
algoritmo, algorithm, . . .

Exemplo:

Problema:

achar o máximo divisor comum (MDC) de dois inteiros positivos dados: **M** e **N**.

Idéia ?? ...

Como aprendemos na escola ...

Algoritmo:

1. Se **M=N**, então MDC é **M** (ou **N**); pare
2. Caso a):
 - se (**M>N**) então
 - substitua **M** por (**M-N**) e repita a partir do passo 1
3. Caso b):
 - se (**N>M**) então
 - substitua **N** por (**N-M**) e repita a partir do passo 1

Dados:

Dois números inteiros, $M \geq 1$ e $N \geq 1$

Saída:

Um número inteiro Z , tal que $Z = \text{MDC}(M, N)$

Apoio, executores:

Lápis, papel, borracha, humano

Algoritmos

O que são?

Executando a receita: caso particular no qual **M=36** e **N=21**

Passo	M	N	Comentários
---	36	21	
1	36	21	$36 \neq 21$
2	15	21	$36 - 21 = 15$
1	15	21	$15 \neq 21$
2	15	21	não executado: $15 < 21$
3	15	6	$21 - 15 = 6$
1	15	6	$15 \neq 6$
2	9	6	$15 - 6 = 9$
1	9	6	$9 \neq 6$
2	3	6	$9 - 6 = 3$
1	3	6	$3 \neq 6$
2	3	6	$3 < 6$; não executado
3	3	3	$6 - 3 = 3$
1	3	3	MDC é 3. Pare.

Algoritmo

1. Se **M=N**, então MDC é **M** (ou **N**); páre
2. Caso a):
se (**M>N**) então
 substitua **M** por (**M-N**) e repita a
 partir do passo 1
3. Caso b):
se (**N>M**) então
 substitua **N** por (**N-M**) e repita a
 partir do passo 1

1. Algoritmo é formado por um **texto finito**:

É a receita dada.

2. O texto é composto por **instruções elementares**:

Elementar depende do contexto:

- “ ... **juntar dois ovos** ... ” é **elementar** para um cozinheiro
- “ ... **substituir M por $(M-N)$** ... ” é **elementar** para quem domina aritmética básica
- “ ... **se hoje você puder *provar* que a cotação do dólar vai subir 10% no próximo mês, compre \$ 1.000,00** ... ” **não é elementar** para mortais normais

3. O texto é uma receita **metódica**, passo-a-passo:

- Passo inicial
- Passo final
- Executado um passo, qual o seguinte?

4. Ao executar:

- partindo de **dados válidos**, deve sempre **terminar**.
- partindo de **dados não-válidos**, pode produzir **lixo**, ou mesmo **não terminar**.
- parte **difícil de garantir**.

Exemplo: algoritmo do MDC sempre pára quando $M \geq 1$ e $N \geq 1$:

- A cada execução dos passos 2 ou 3, M ou N diminuem; logo $(M+N)$ diminui.
- M e N sempre são ≥ 1 iniciam assim;
 $M - N \geq 1$, se $M > N$
 $N - M \geq 1$, se $N > M$
- Não podemos passar de $M=N=1$
Nesse caso, $MDC = 1$ e **para**.

1. Se $M=N$, então MDC é M (ou N); páre
2. Caso a):
se $(M > N)$ então
substitua M por $(M-N)$ e repita a partir do passo 1
3. Caso b):
se $(N > M)$ então
substitua N por $(N-M)$ e repita a partir do passo 1

Exemplo: e com dados não válidos?

Iniciando com $M = 3$ e $N = -1$

Passo	M	N	Comentários
1	3	-1	$-1 \neq 3$
2	4	-1	$3 > -1$; $3 - (-1) = 4$
1	4	-1	$-1 \neq 4$
2	5	-1	$4 \neq -1$; $4 - (-1) = 5$
1	5	-1	$-1 \neq 5$
2	6	-1	$5 \neq -1$; $5 - (-1) = 6$
...			repete esse padrão
não pára			não vai parar nunca

1. Se $M=N$, então MDC é M (ou N); páre
2. Caso a):
 - se ($M > N$) então
substitua M por $(M-N)$ e repita a partir do passo 1
3. Caso b):
 - se ($N > M$) então
substitua N por $(N-M)$ e repita a partir do passo 1

- **Algoritmo:**
programa, software ...
- **Computador, HD, disquete, ... :**
hardware, executores, atores
- **Entrada:**
teclado, mouse, sensores, ...
- **Saída:**
monitor, impressora, ...

Características dos algoritmos como software:

1. Texto **finito**:

todo programa tem um texto (talvez muitas linhas) **finito**

2. Instruções **elementares**:

elementares para o **computador** onde o software vai **executar**.

Dificuldades:

- Cada computador tem um **particular** conjunto de instruções básicas
- Instruções do computador são **muito primitivas**

Solução: escrever algoritmos em uma **linguagem de programação** (**C**, **C++**, **JAVA**, **FORTRAN**, . . .)

 **Programa (software): texto** escrito numa particular **LP**

Características dos algoritmos como software (cont):

3. Receita **metódica**:

texto escrito numa LP é **preciso** e **sem ambigüidades**

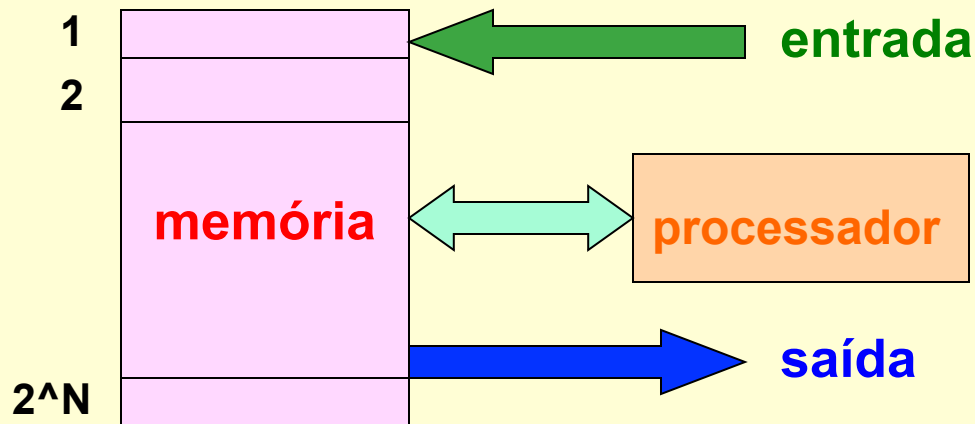
4. **Terminação**:

1. **Grande desafio**: texto escrito numa LP **não deixa isso claro**.
2. **Problema** no desenvolvimento de software: execução sem terminação (i.e. sem “loops”).
 - » Outro problema: termina com a solução errada ou tem interrupção abrupta.

Um computador é, essencialmente, uma máquina programável muito “**primitiva**”

- **instruções elementares** (de máquina) primitivas
- lidam apenas com **cadeias** de “bits”
- realizam **operações muito simples** sobre essas cadeias de bits
 - **trocar um bit** (de 0 para 1, ou de 1 para 0), dependendo do valor atual de outro bit
 - **armazenar na memória** uma cadeia de bits
 - **recuperar da memória** uma cadeia de bits

Arquitetura básica simplificada:



Memória

N	2^N	Bytes
10	1024	1 K
20	1048576	1 M
27	134217728	128 M
30	1073741824	1 G
32	4294967296	4 G

1 byte = 8 bits

Arquitetura básica simplificada (cont):

- Formado por **memória** (grande), **processador** e circuitos de **entrada** e de **saída**
- Processador executa **instruções elementares específicas dessa máquina**
 - inclusive armazenamento e recuperação de dados da memória
- Processador e circuitos de e/s **recebem dados** das **entradas** e **exibem dados** nas **saídas**

Compilação:

- Processo de traduzir programas escritos em uma **particular LP** para código em **instruções básicas** de uma **máquina específica**
- Tradução de **LP** para **linguagem de máquina**:
 - **Dificuldades**: processo laborioso, entediante e sujeito a erros.
 - **Solução**: Escrever **um programa** para fazer a tradução



Esse programa é um

compilador

➡ Para cada **LP** e cada **computador** (**processador**), um **compilador específico**

Existem milhares de **LPs**:

- **FORTRAN**: científica, mais antiga
- **ALGOL**, **C**, **PASCAL**: estruturadas, generalistas
- **C++**, **C#**, **JAVA**: lidam com tecnologia de objetos
- **LISP**, **PROLOG**: voltadas para IA
- ... (muitas outras)

O processo de compilação/edição/execução:



- **Codificação/compilação/execução:**
 - permite **executar** um **algoritmo** em um **computador**
 - obtém **solução** para problemas
 - rapidamente (?!)

- **Algoritmo:**
 - **centro** do universo
 - demais componentes são **acessórios**
 - “Ciências da Computação”: estudo de **algoritmos**

- Pessoas codificam **algoritmos** em **LPs**
- Que tipos de **instruções** (em geral) estão presentes em uma **LP**?
 - **Atribuição:**
A = E;
 - **Seqüenciamento:**
... faça A; faça B; ...
...
faça A;
faça B;
...

- Tipos de **instruções** (em geral) em uma **LP** (cont):
 - **Desvio condicional:**
 - ... se (A é verdade) então (faça B) senão (faça C); ...
 - ... se (A é verdade) então (faça B); ...
 - **Iterações:**
 - ... faça A exatamente N vezes; ...
 - ... repita A até que (Q seja verdade); ...
 - ... enquanto (Q é verdade) faça A; ...
 - **Inúmeras outras**, mais sofisticadas, dependendo da LP

Como dados são representados e manipulados em LPs?

- Valores dos dados são armazenados na memória;
- A memória é referenciada por meio de variáveis:
 - Variável: um nome simbólico que designa uma, ou mais, posições na memória
 - Exemplo: X, Z, D3, CASA_DE_PEDRA, . . .
 - A cada variável está associado um tipo de dados:
 - O número de posições de memória ocupadas pela variável depende do seu tipo;
 - As operações e manipulações permitidas com o valor e uma variável dependem de seu tipo.

- Entender bem o problema a ser resolvido:
 - separar dados de entrada válidos dos que não são válidos
 - definir como será representada a solução na saída
- Criar uma idéia para resolver o problema
 - desenvolver o algoritmo (processo criativo, rascunho, lápis e papel)
 - simular execução do algoritmo em casos de fronteira
 - verificar correção e término
- Traduzir a idéia para uma LP, escrevendo um programa
 - restrito aos comandos e tipos de dados da LP
- Editar/compilar/executar o programa
 - processo iterativo para retirar erros (algoritmo e código)

O **algoritmo** corretamente **soluciona** o problema?

Provar um teorema (como em matemática) mostrando que o **algoritmo é correto**:

- processo de “**convencimento dos pares**”
- possível exibir uma “**prova formal**” da correção?
 - **Dificuldades:**
 - **precisão e rigor** ao descrever a **execução** do algoritmo
 - **especificar formalmente dados** de entrada e a saída

Questão:

Dado um problema **P**, sempre haverá um **algoritmo** que **resolva P corretamente**?

- **P** deve ser um problema **prático, fácil de enunciar**
 - ordenar um conjunto de números,
 - calcular produto de matrizes, . . .
- O **algoritmo A** que **resolve P** deve **funcionar corretamente** em **todas as** (infinitas) **entradas** de **P**
 - todos os conjuntos de inteiros quaisquer
 - quaisquer duas matrizes de quaisquer dimensões compatíveis entre si

Algoritmos

... resolvem qualquer problema?

A Ciência da Computação **tem a resposta** para a questão

SURPRESA !!!

NÃO. Há **problemas** para os quais **NÃO EXISTE** algoritmos capazes de resolvê-los **corretamente**.

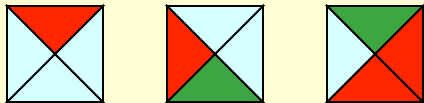
Com **mais tecnologia** (computadores mais rápidos, mais memória) e dado tempo suficiente para rodar o programa, poderemos **resolver** esses problemas, **no futuro**, certo?

NÃO

Computador nenhum vai **resolver** esses problemas, **nem hoje, nem amanhã, nem nunca**; **nem aqui, nem em Marte, em lugar algum**; rodando **qualquer programa** por **quanto tempo quiser** (anos, séculos, ...)

Um problema **indecidível** (**insolúvel**) [Harel, “Computers Ltd.”]:

Dado um conjunto finito **T** de ladrilhos quadrados:



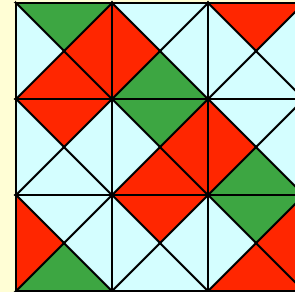
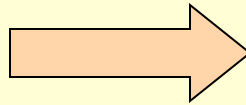
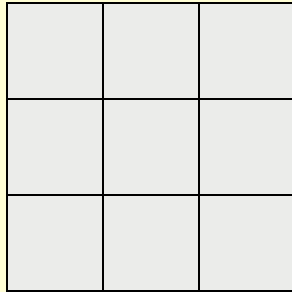
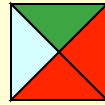
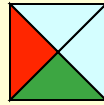
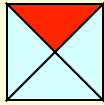
Problema: podemos ladrilhar **qualquer** grade quadrada com ladrilhos de tipo **T**, casando as cores das faces que se tocam? **SIM** ou **NÃO**?

- pode usar quantos ladrilhos quiser, de cada tipo
- os ladrilhos não podem ser girados

Algoritmos

... resolvem qualquer problema?

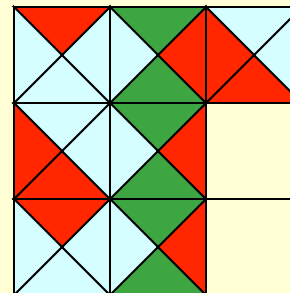
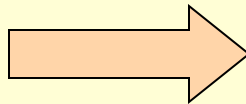
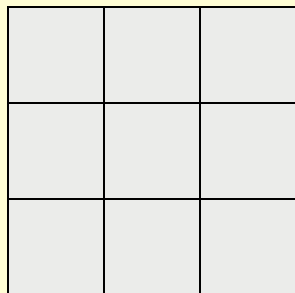
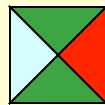
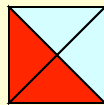
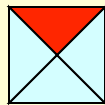
Exemplo 1:



consegue para toda
região do plano

SIM!

Exemplo 2:



todas as outras
possibilidades **falham**
nessa região

NÃO!

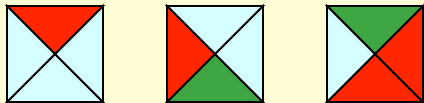
Problema de ladrilhar toda região do plano:

NENHUM computador **JAMAIS** vai conseguir resolver esse problema, nem agora, nem nunca, nem com **QUALQUER** melhoria de tecnologia, nem com **QUALQUER** tamanho de memória, nem com **QUALQUER** tempo de execução

Podemos demonstrar isso, matematicamente!

Um problema **decidível** (**solúvel**) [Harel, “Computers Ltd.”]:

Dado um conjunto finito **T** de ladrilhos quadrados:



Dadas duas posições (“**I**” e “**F**”) na grade infinita do plano

Problema: podemos ladrilhar um caminho na grade, partindo de “**I**” e chegando em “**F**”, com ladrilhos de tipo **T**, e casando as cores das faces que se tocam? **SIM** ou **NÃO**?

- pode usar quantos ladrilhos quiser, de cada tipo
- os ladrilhos não podem ser girados

Problema de ladrilhar um caminho entre duas posições:

EXISTE um **algoritmo** que **decide se há**, ou **se não há**, o caminho entre as duas posições dadas, usando ladrilhos do conjunto dado

Podemos **exibir** o **algoritmo** e **mostrar** sua **correção** e **terminação**, não importa qual o conjunto **T** dado e não importa quais as posições **I** e **F** dadas

Dado um **problema** **P**, e conseguido um **algoritmo** **A** para **P**, então podemos **resolver** qualquer **instância** de **P**, com **dados de entrada** **E**, executando **A** sobre os dados **E**.

CORRETO?

NEM SEMPRE:

- ao executar sobre **E**, o **algoritmo** **A** pode precisar de um **tempo muito longo** (anos, séculos, milhões de séculos, ...)
- ao executar sobre **E**, o **algoritmo** **A** pode precisar de um **muita memória** (vários GBs, muitos milhões de GBs, ...)

Nesses casos, **A** é um **algoritmo imprestável!**

Algoritmos

... adianta executá-los?

Se **A** é um **algoritmo** ruim, então basta criar outro **algoritmo** para **P**, que seja **mais eficiente** (em tempo e/ou em memória)

CORRETO?

SURPRESA !!!

Pode ser que **NÃO EXISTA** um **algoritmo mais eficiente** do que **A** para resolver **P** e podemos **provar** isso matematicamente!

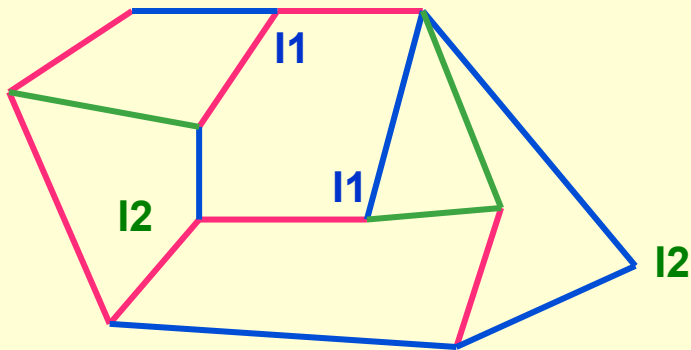
P é um problema **computável**, porém **intratável**

Algoritmos

... adianta executá-los?

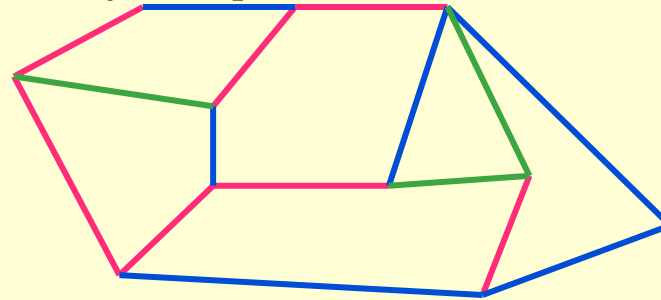
Exemplo: O jogo do bloqueio [Harel, op. cit.]

Dado um mapa rodoviário:



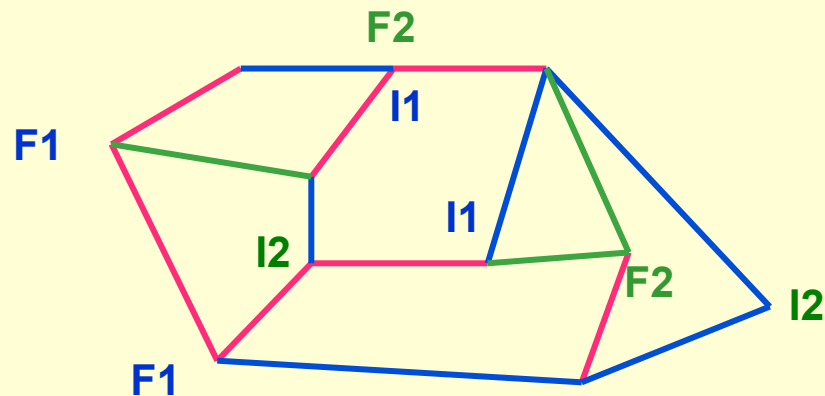
Posições finais de **J1**: **F1**

Posições finais de **J2**: **F2**



Posições iniciais de **J1** (jogador 1): **I1**

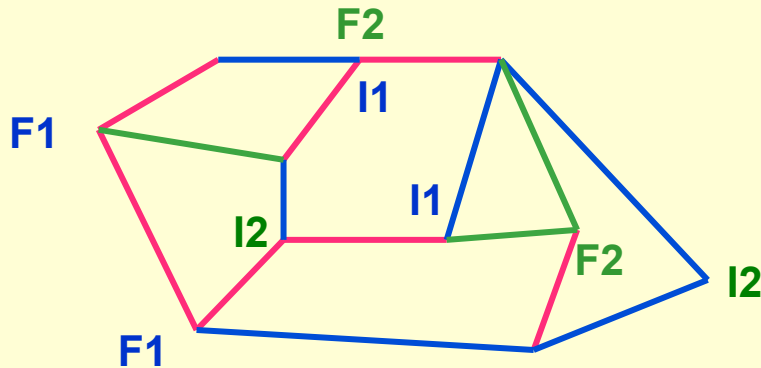
Posições iniciais de **J2** (jogador 2): **I2**



Problema: Será que o jogador **J1** tem uma **estratégia vencedora**?

Regras de movimentos:

- **J1 inicia**; depois os jogadores se **alternam**.
- Em **um movimento**, um jogador pode percorrer **qualquer trecho** (concatenado) de **mesma cor**, partindo de uma posição **ocupada por si**
 - não pode passar por intersecções ocupadas (por si ou pelo adversário)
 - ponto final deve estar também desocupado
- **Vencedor**: aquele jogador que chegar a um **ponto final** primeiro



Nesse **mapa**, nessas posições **iniciais** e **finais**

J1 TEM estratégia (seq. de movimentos) sempre **vencedora**

Jogo do bloqueio:

Algoritmo A:

- De forma **sistemática**, tente **todas as possibilidades** de seqüências alternadas de movimentos, começando com **J1**

Possível:

- número **finito** de possibilidades

Dificuldade:

- o número de possibilidades é **enorme**
 - para **cada** movimento de **J1**, deve tentar **todos** os movimentos de **J2**
 - para **cada** movimento de **J2**, deve tentar **todos** os movimentos de **J1**

Estimando o tempo de execução do algoritmo **A**:

Uma quantidade, **N**, mede o “tamanho” (num. de bits na representação) da entrada

-- por exemplo, **N** pode ser o número de intersecções, ou o número de vias, ou

Tempo de execução dado pela contagem do número de passos elementares que **A** executa, no pior caso, para entradas de tamanho **N** é dado pela função $f(N) = 2^N$.

Algoritmos

... adianta executá-los?

Tempo de execução do algoritmo A:

Em um computador que realiza **1 milhão** de passos elementares **por segundo**, o **tempo de execução** de **A** seria

n	10	20	50	60	100	200
$f(n)=2^n$	1 ms	1 s	35.7 anos	3.000 anos	+ 400 trilhões anos	num. séc. tem 45 dígitos!

Impraticável para 50 ou mais cidades

Rodando **A** em um computador **10.000** vezes mais rápido

n	50	60	100	200
$f(n)=2^n$	1,29 dias	3 anos	+ 40 bilhões séc.	num. séc. tem 41 dígitos!

Impraticável para 60 ou mais cidades: **quase nada muda!**

Algoritmo A não é bom para o problema do bloqueio:

Precisamos de outro **algoritmo**, **mais eficiente**

O que é um algoritmo “**eficiente**”?

- **N**: é o tamanho de uma entrada válida
- **f(N)**: quantos passos, no máximo, **A** executa com entradas de tamanho **N**

Algoritmos

... adianta executá-los?

com um milhão de passos por segundo

$f(N) \backslash N$	10	20	50	100	200
N^2	0,1 ms	0,4ms	2,5ms	10ms	40s
N^5	0,1s	3,2s	5,2m	2,8h	3,7dias
2^N	1ms	1s	35,7anos	séculos 400 trilhões	séculos 45 dígitos
N^N	2,8s	3,3 trilhões anos	séculos 70 dígitos	séculos 185 dígitos	séculos 445 dígitos

Tempos de execução, de pior caso:

- Polinomiais: resultam em algoritmos eficientes
- Exponenciais: resultam em algoritmos não eficientes

Problemas tratáveis: têm algoritmos polinomiais

Problemas intratáveis: não têm algoritmos polinomiais

Problema do bloqueio: é intratável

Dado um problema **P**, como saber se é **tratável**?

SURPRESA!!!

Para **muitos** problemas de **grande interesse prático**,
não sabemos se são tratáveis ou não!

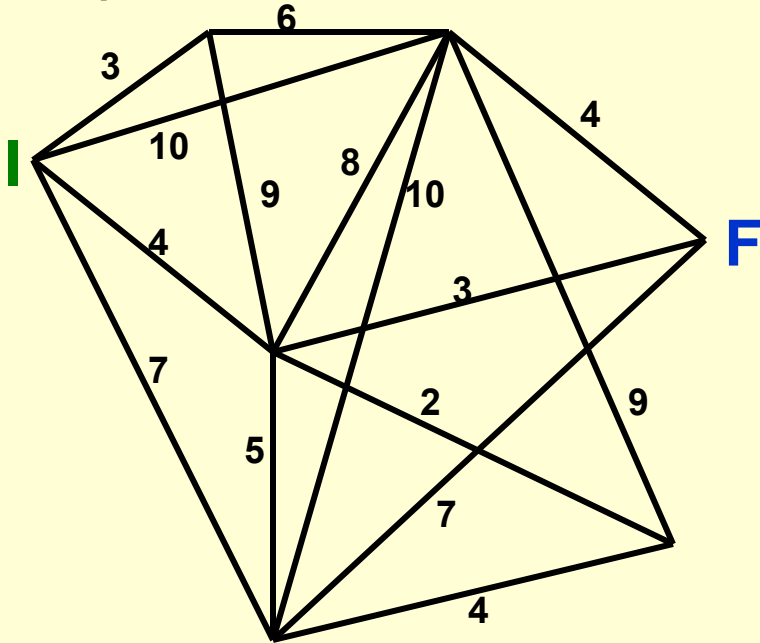
Essa é um das **maiores questões em aberto** em
Ciências da Computação!

Exemplo: problema do caixeiro viajante

- **Dado:**
 - mapa de cidades, com custo de viagem entre cada par de cidades
 - cidade de início, **I**, cidade de término, **F**
 - um valor **K**
- **Problema:**
 - existe rota, de **I** até **F**, visitando as cidades exatamente uma vez
 - com custo no máximo **K**? **SIM/NÃO**?
- **A notar:**
 - problema de grande interesse prático
 - problema simples de entender

Instância:

O mapa:



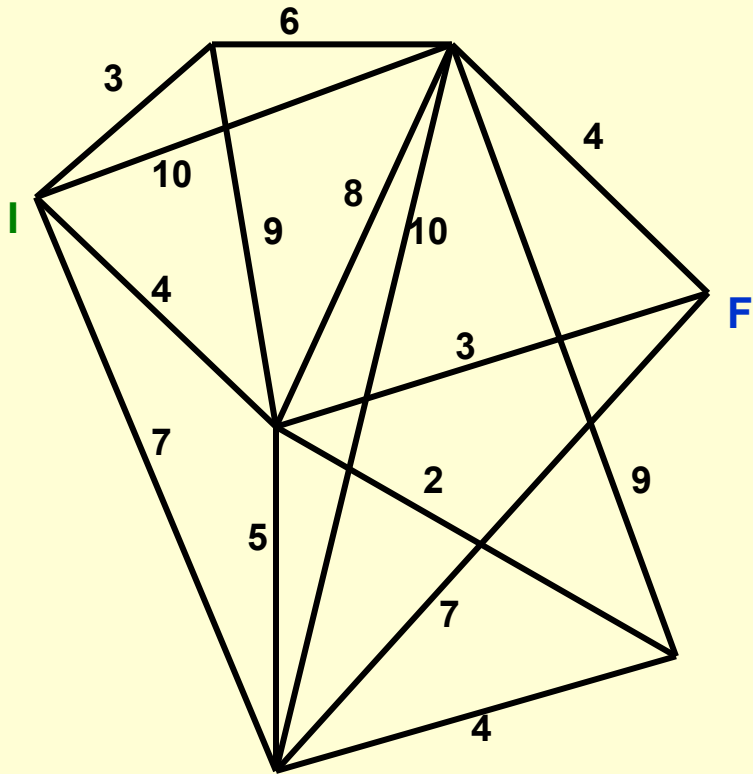
O valor máximo do
percurso: **29**

EXISTE um percurso?
SIM/NÃO

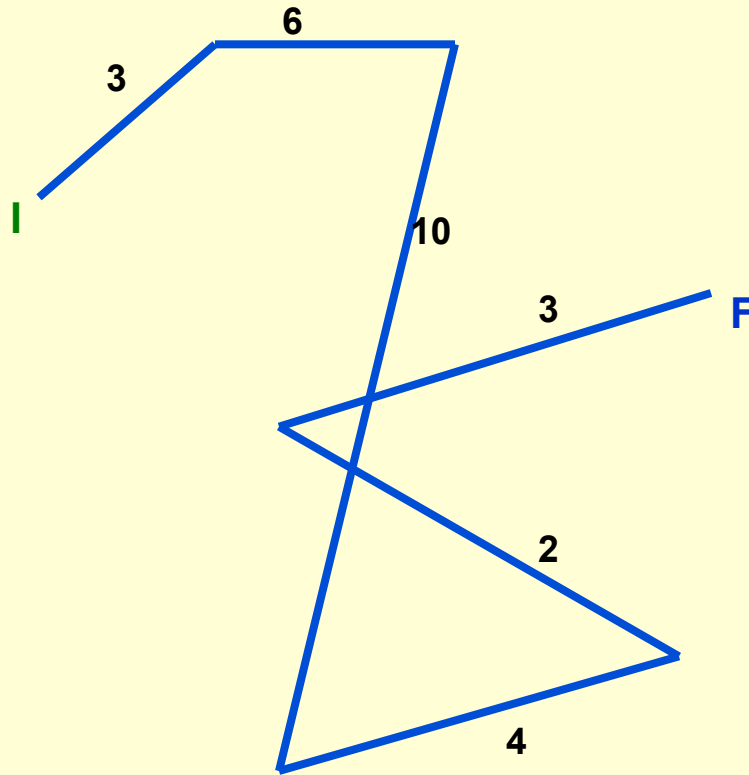
Os pontos **inicial** e **final**

Algoritmos

... nossa ignorância

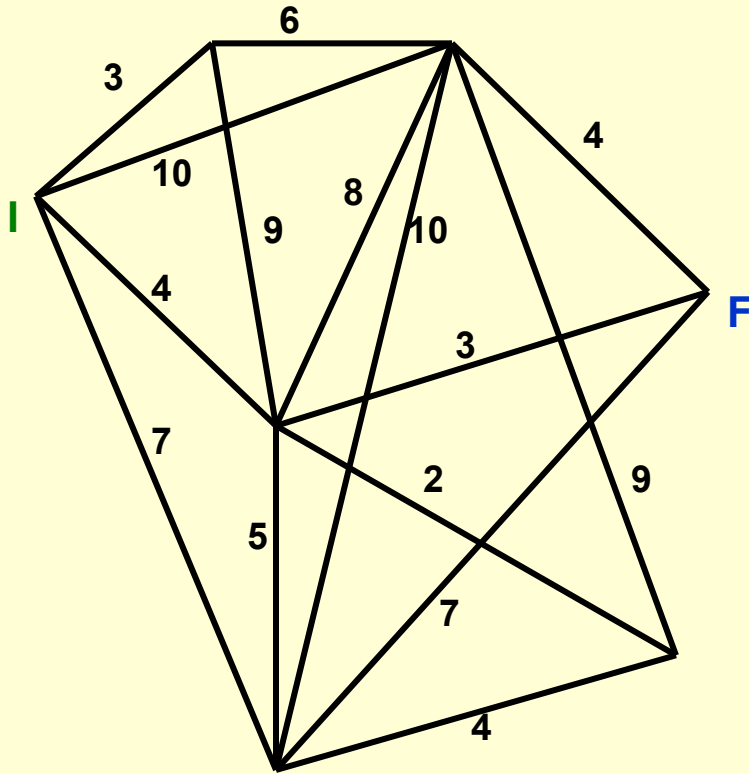


K=29

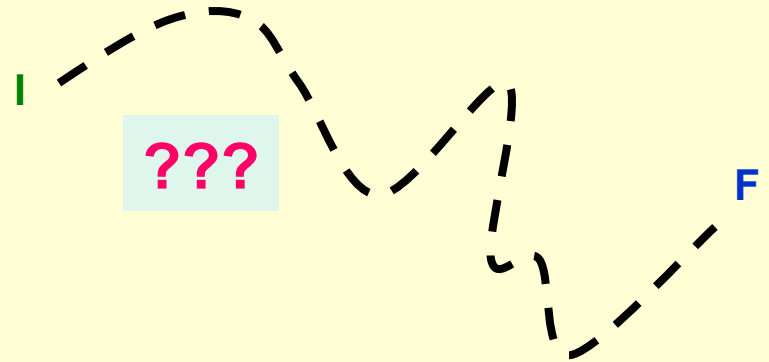


$$3+6+10+4+2+3 = 28$$

SIM



K=25



Com esse custo não é possível

NÃO

Algoritmos para o problema do **caixeiro viajante**:

- Partindo da posição **I**, tente todas as possibilidades que fiquem dentro do custo **K**
 - Se achar um caminho até **F**, responda **SIM**
 - Se não achar, responda **NÃO**
- Número de possibilidades é **finito**
 - **algoritmo** corretamente **resolve** o problema
- Número de possibilidades é **muito grande**
 - tempo de execução é **exponencial** no **número de cidades**
- O algoritmo é **impraticável**

Problema do **caixeiro viajante**:

Existe um algoritmo **mais eficiente** (polinomial no **número de cidades**)?

SURPRESA !!!

NÃO SABEMOS !?!

Esse é o caso com **MUITOS** outros problemas **de interesse prático** (classe NP)

O que podemos fazer, por ora?

- **Uso de heurísticas:**
 - obtém “boas” soluções, sem garantia de otimalidade
- **Algoritmos randomizados:**
 - dão a resposta correta quase sempre
- **Algoritmos aproximativos:**
 - dão solução com garantia de proximidade da ótima
- **Computação quântica:**
 - baseado na mecânica quântica, nova maneira de programar
- **Computação molecular:**
 - paralelismo maciço usando reações moleculares

■ ■ ■