

- 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 bifés.**
- 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.**

Algoritmos

O que são?

- Objetos de “consumo” (entrada):

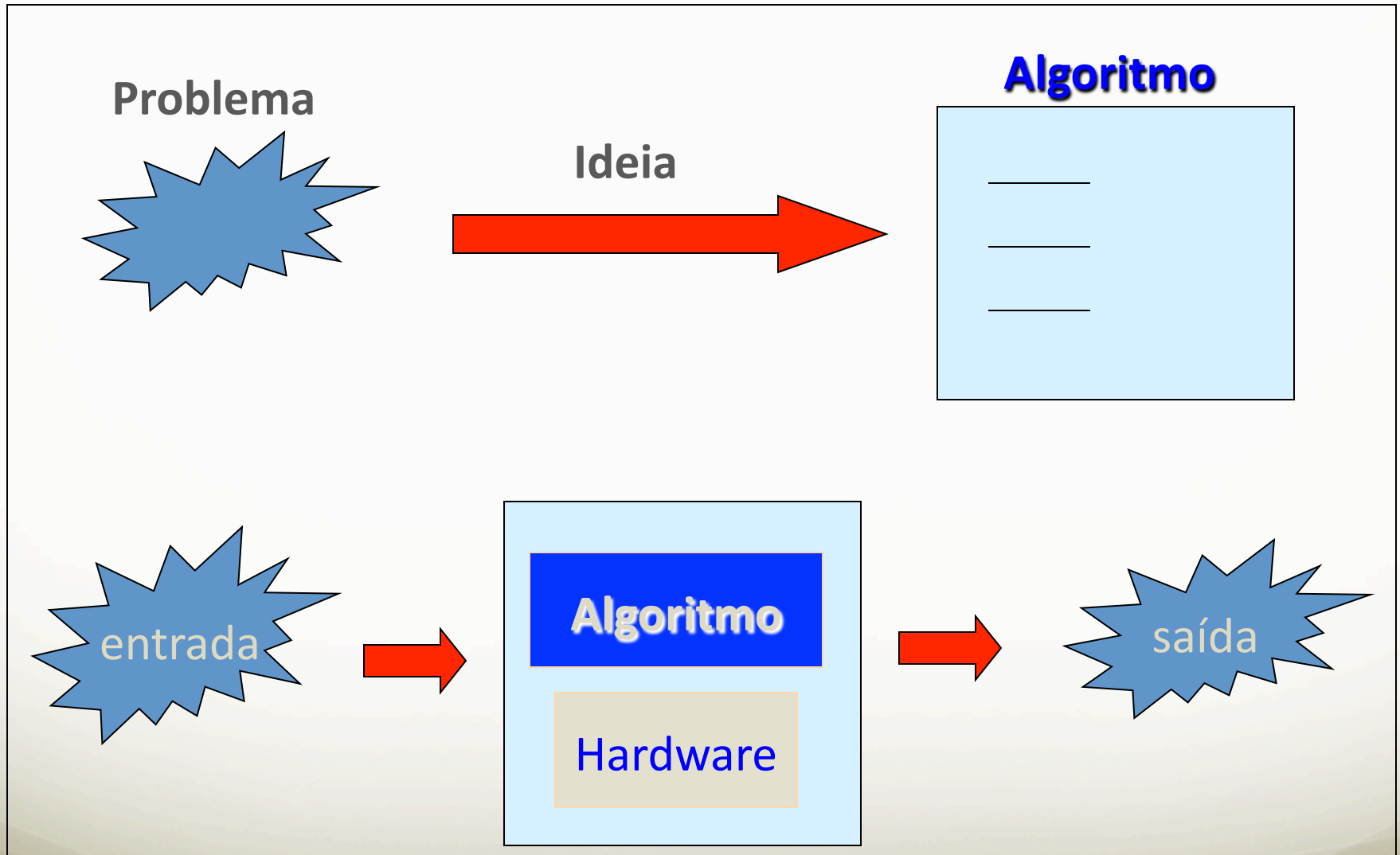
- ✓ carne;
- ✓ farinha;
- ✓ ovos;
- ✓ alface.

- Objetos de “apoio” (atores, executores):

- ✓ faca;
- ✓ travessa;
- ✓ fogão;
- ✓ cozinheiro.

- Objetos “produzidos” (saída):
✓ Bifes.

- Objeto que “controla” o processo (receita):
✓ **Algoritmo.**



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, estabelece claramente que é 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**.

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

Características dos algoritmos como software.

1. Texto **finito** (talvez muitas linhas).
2. Instruções **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 em uma LP é **preciso** e **sem ambiguidades**.

4. **Terminação**:

1. **Grande desafio**: texto escrito em uma LP **não deixa isso claro**.

2. **Problemas** no desenvolvimento de software:

- execução sem terminação (i.e. com “loops”);
- termina com a solução errada ou tem interrupção abrupta.

Um computador é 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);
 - ✓ **armazenar na memória** uma cadeia de bits;
 - ✓ **recuperar da memória** uma cadeia de bits.

- 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!**

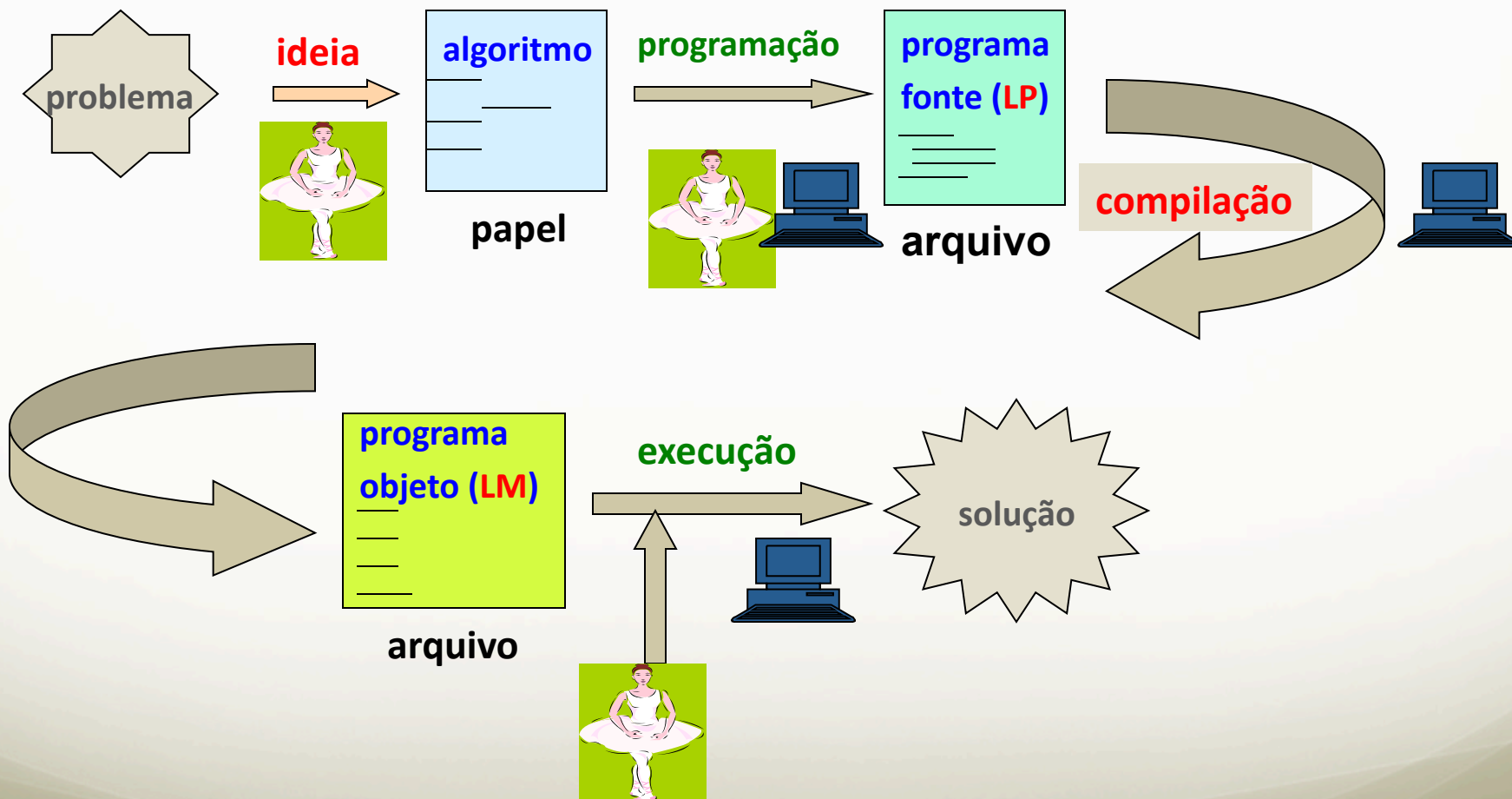


Para cada **LP** e cada **computador (processador)**, é necessário um **compilador específico**.

Existem muitas **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:



- 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 ideia para resolver o problema:
 - desenvolver o **algoritmo** (processo criativo, lápis e papel);
 - simular execução do algoritmo em casos de contorno;
 - verificar correção e término.
- Traduzir a idéia para uma **LP**, escrevendo um programa:
 - é 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.

- possível exibir uma “**prova formal**” da correção?
 - Dificuldades:
 - ✓ **precisão e rigor** ao descrever a **execução** do algoritmo;
 - ✓ **especificação formal** dos dados de entrada e 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 inteiros;
 - calcular produto de matrizes.
- Um **algoritmo** que resolva **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.

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

SURPRESA !

Não! Há **problemas** para os quais **não existem** 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.

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

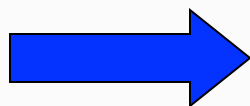
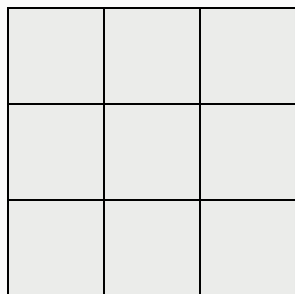
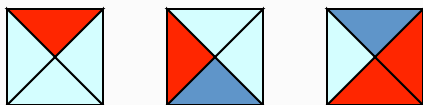
Dado um conjunto finito **T** de ladrilhos quadrados:



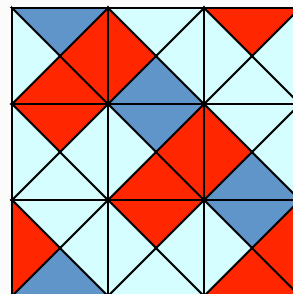
Problema: podemos ladrilhar **qualquer** grade quadrada com ladrilhos de tipo **T**, casando as cores das faces que se tocam?
SIM ou **NÃO**?

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

Exemplo 1:

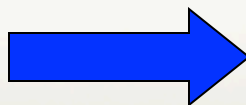
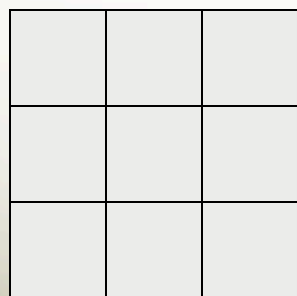
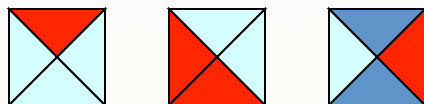


SIM!

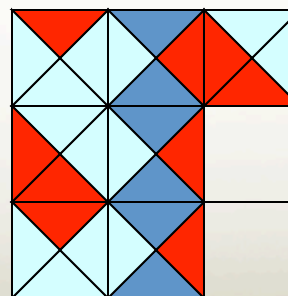


consegue para toda região.

Exemplo 2:



NÃO!



todas as outras possibilidades falham nessa região do plano.

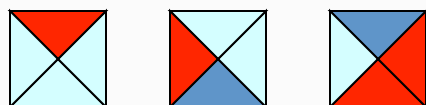
Problema de ladrilhar toda região do plano

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:

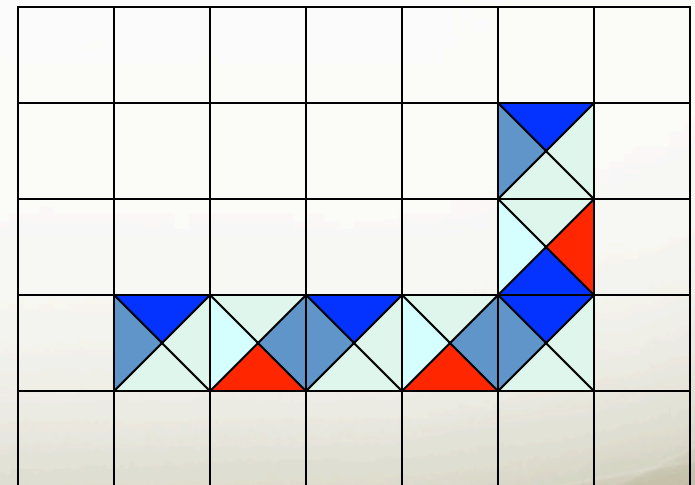
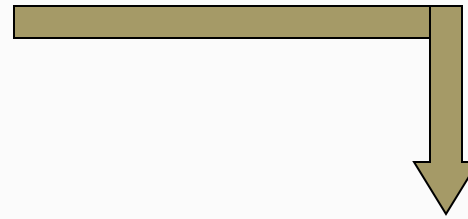
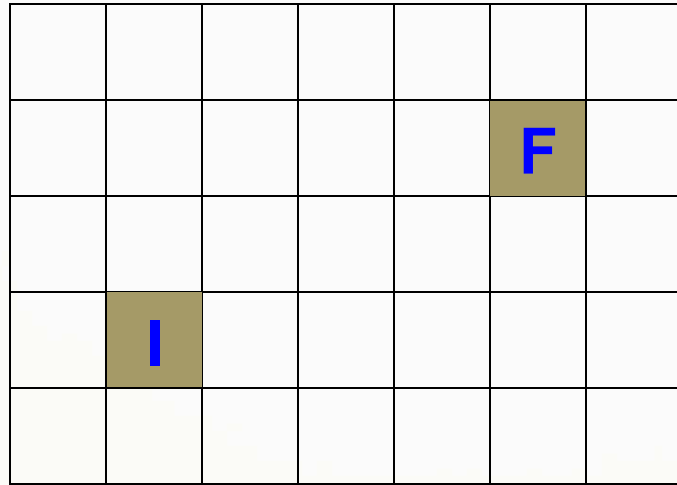
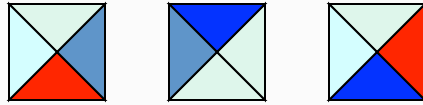


Dadas duas posições (“I” e “F”) na grade infinita do plano

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.

Exemplo:



Com **esses** ladrilhos,
com **essas** posições **I** e **F**:

SIM!

Problema de ladrilhar um caminho entre duas posições

EXISTE um **algoritmo** que **decide se há**, ou **se não há**, um 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 importam 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!**

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 !

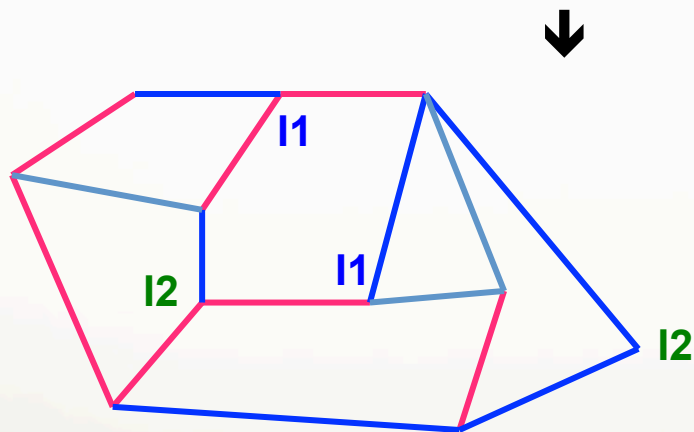
Pode ser que **não exista** um **algoritmo mais eficiente** do que **A** para resolver **P**. Talvez possamos **provar** isso matematicamente!

Neste caso, **P** é um problema **computável**, porém **intratável**.

Exemplo: O jogo do bloqueio [Harel].

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

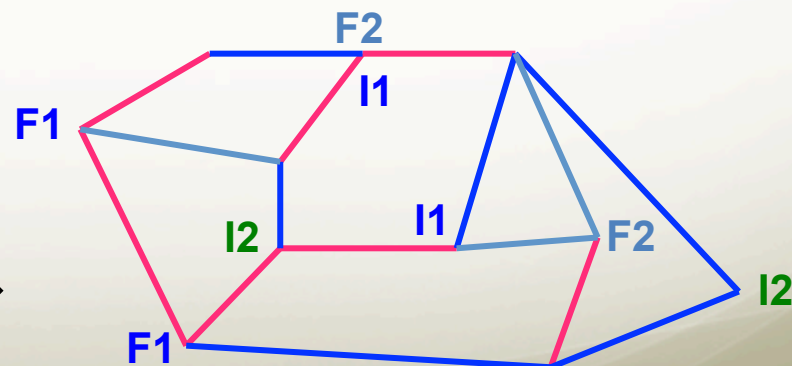
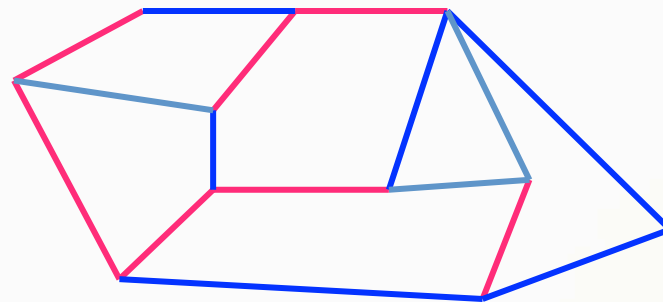
Posições iniciais de **J2** (jogador **2**): **I2**.



Posições finais de **J1**: **F1**.

Posições finais de **J2**: **F2**.

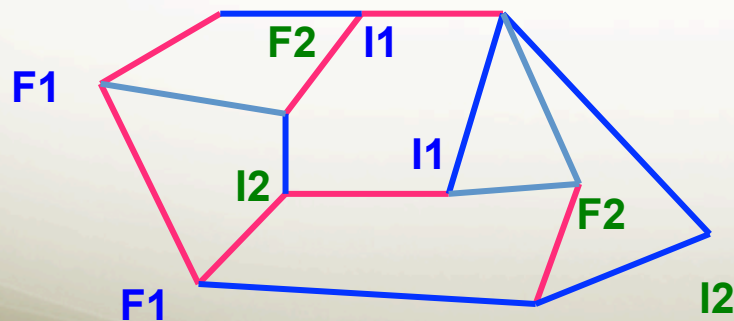
Dado um mapa rodoviário.



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.

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



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 sequências alternadas de movimentos, começando com **J1**.

Possível:

- número **finito** de possibilidades.

Dificuldade:

- o número de possibilidades é **enorme** pois:
 - 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***;
 - ✓ neste caso, é dado pela função **$f(n) = 2^n$** .

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** .

com **um milhão** de passos por segundo:

$f(n)$ \ 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!

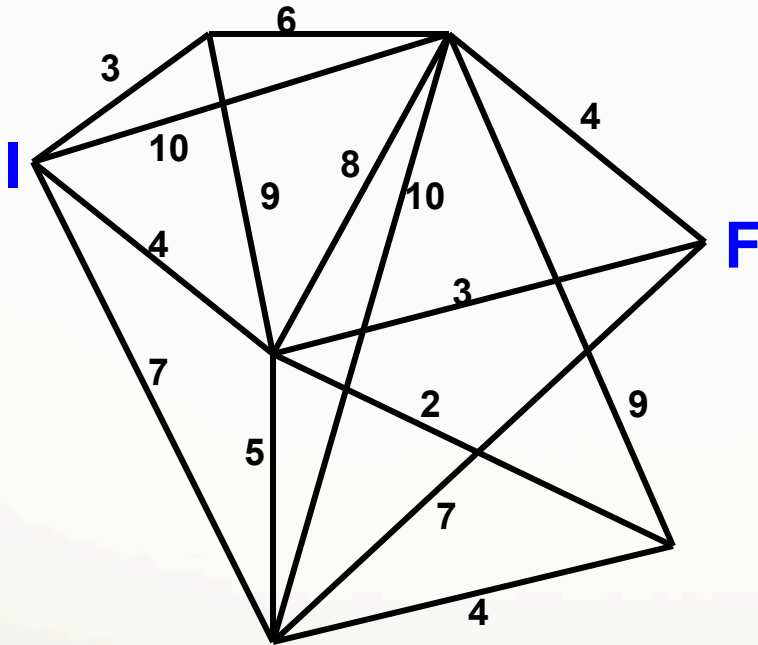
Para muitos problemas de grande interesse prático,
não sabemos se são tratáveis ou não!

Essa é um das maiores questões em aberto em
Ciência 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 todas as cidades exatamente uma vez, com custo no máximo **K**?

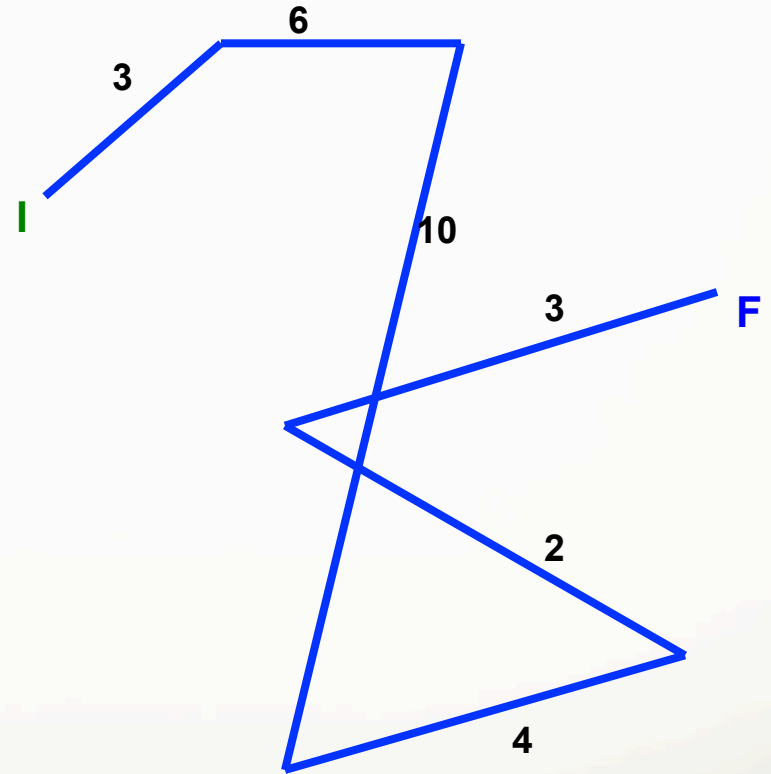
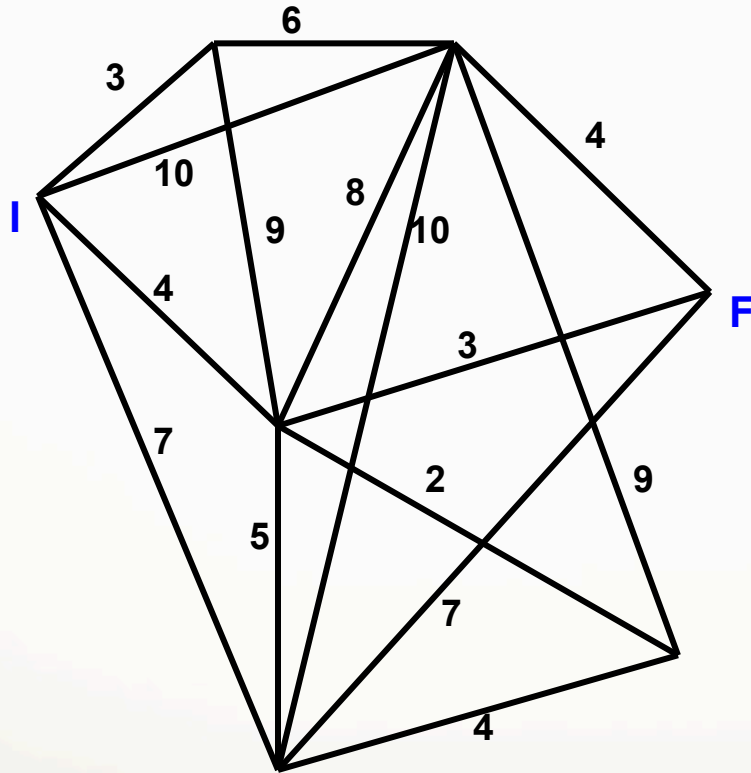
Instância:



O valor máximo do percurso: **29**

Existe um percurso?

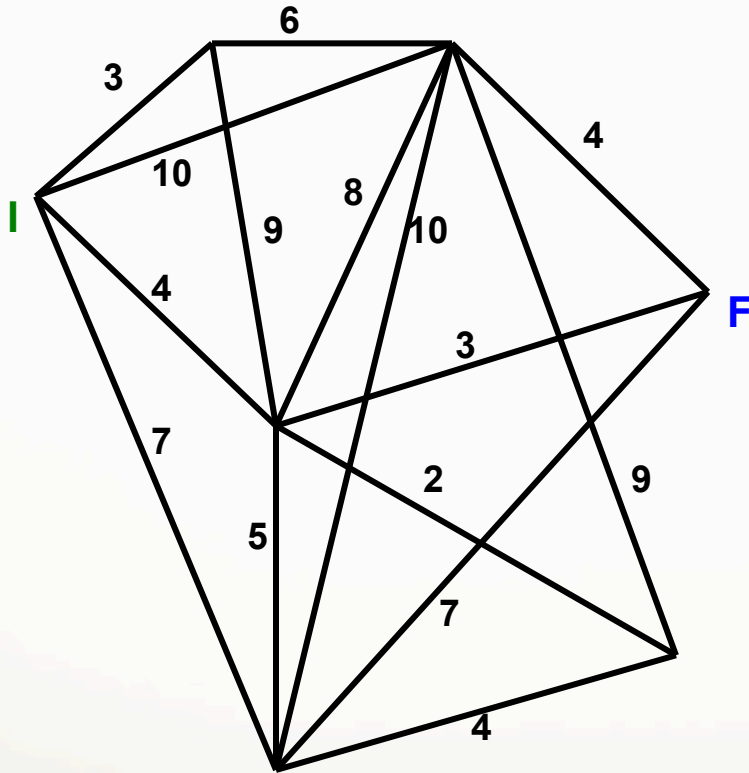
SIM / NÃO



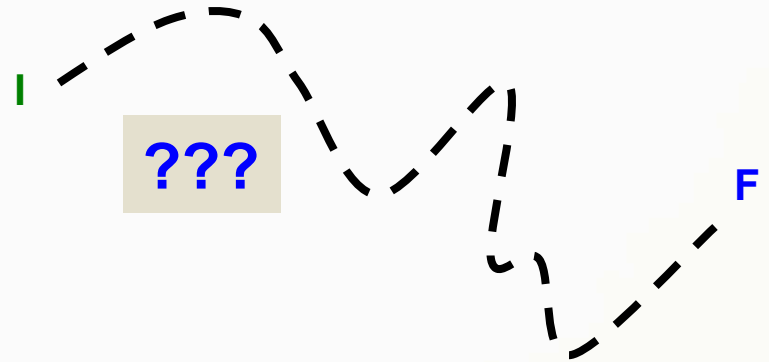
K=29

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

SIM



K=25



Com esse custo não é possível!

NÃO

Algoritmo 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)?

NÃO SABEMOS !

Esse é o caso de **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.

...