PANC: Projeto e Análise de Algoritmos

Aula 15: Programação Dinâmica e Teoria da Complexidade Computacional (Resumido)

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Revisão de Conteúdo
- Programação Dinâmica
- Abordagens de Programação Dinâmica: *Top-Down* vs. *Bottom-Up*
- Ilustração com o Problema de Fibonacci
- O problema da Mochila 0-1
- Teoria da Complexidade Computacional
- Máquinas de Turing
- Classes de Problemas: P, NP, P e NP, NP-Completo e NP-Difícil
- Reduções de Problemas
- Teorema de Cook
- Discussões

Recapitulando...

- O **Backtracking** busca por todas as soluções possíveis (explícita ou implicitamente) e seleciona a melhor
  - Possui corretude, mas é inviável para problemas grandes
- **Algoritmos Gulosos** tomam sempre a melhor decisão a cada instante
  - Sem provas de corretude, falham em obter a solução ótima
- **Divisão e Conquista** divide o problema original em subproblemas menores e “mais fáceis”
  - No entanto, pode resolvê-los repetidamente
  - O balanceamento pode ser primordial

Programação Dinâmica (1)

- *“Dynamic programming is a fancy name for [recursion] with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table*
- *The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table*
- *Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often*
- *In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed”*

- Ian Parberry, *Problems on Algorithms*

Programação Dinâmica (2)

- A palavra **programação** na expressão "programação dinâmica" não tem relação direta com programação de computadores

- Ela significa **planejamento** e refere-se à construção da tabela que armazena as soluções dos subproblemas

- A **Programação Dinâmica (ou PD)** é talvez o paradigma de solução de problemas mais desafiantes dentre os outros vistos
  - Este paradigma pode "parecer mágica", até que tenhamos visto exemplos o suficiente, tornando-o relativamente fácil de ser aplicado.
  - É necessário nos assegurarmos de termos dominado todos os outros paradigmas anteriores antes de continuar
  - Recursões e recorrências também serão vistas novamente

Programação Dinâmica – Características (1)

- Algoritmos de PD são quase sempre definidos por recursividade:
  - Definem a solução para um problema em termos da solução de problemas menores
  - De certa forma, lembram D&C e algoritmos gulosos

- Um possível defeito na busca recursiva é a computação redundante de subproblemas, ou a exploração redundante do espaço de busca
  - Para contornar esta situação, podemos armazenar os resultados dos subproblemas já resolvidos
  - Em parte, por isto o *Backtracking* é ineficiente

Programação Dinâmica – Características (2)

- À medida em que **resolvemos subproblemas**, **armazenamos** os **resultados parciais**, acelerando o algoritmo:
  - Para cada **subproblema inédito**, o **resolvemos** e **armazenamos** o resultado
  - Para os **subproblemas repetidos**, apenas **consultamos** o **resultado**
- É importante primeiro nos certificarmos de que o algoritmo recursivo é correto, depois o aceleramos
- As soluções dos **subproblemas são mantidas em uma tabela**, a qual é consultada a cada subproblema encontrado
- Cada entrada na **tabela** é chamada de **estado** ou **estágio**

Programação Dinâmica – Top-Down vs. Bottom-Up

- Pode ser empregada de duas formas:
  - **Top-Down**:
    - O problema original é decomposto em subproblemas que são resolvidos recursivamente e combinados para obtenção da solução
    - As entradas da tabela são preenchidas de acordo com a necessidade, ao longo da recursão. Pode não preencher toda a tabela
  - **Bottom-Up**:
    - Não se utiliza recursão. Os subproblemas são resolvidos e combinados sucessivamente de maneira a construir a solução do problema original
    - As entradas da tabela são preenchidas "em ordem" e a tabela é totalmente preenchida

O Problema de Fibonacci (1)

$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$

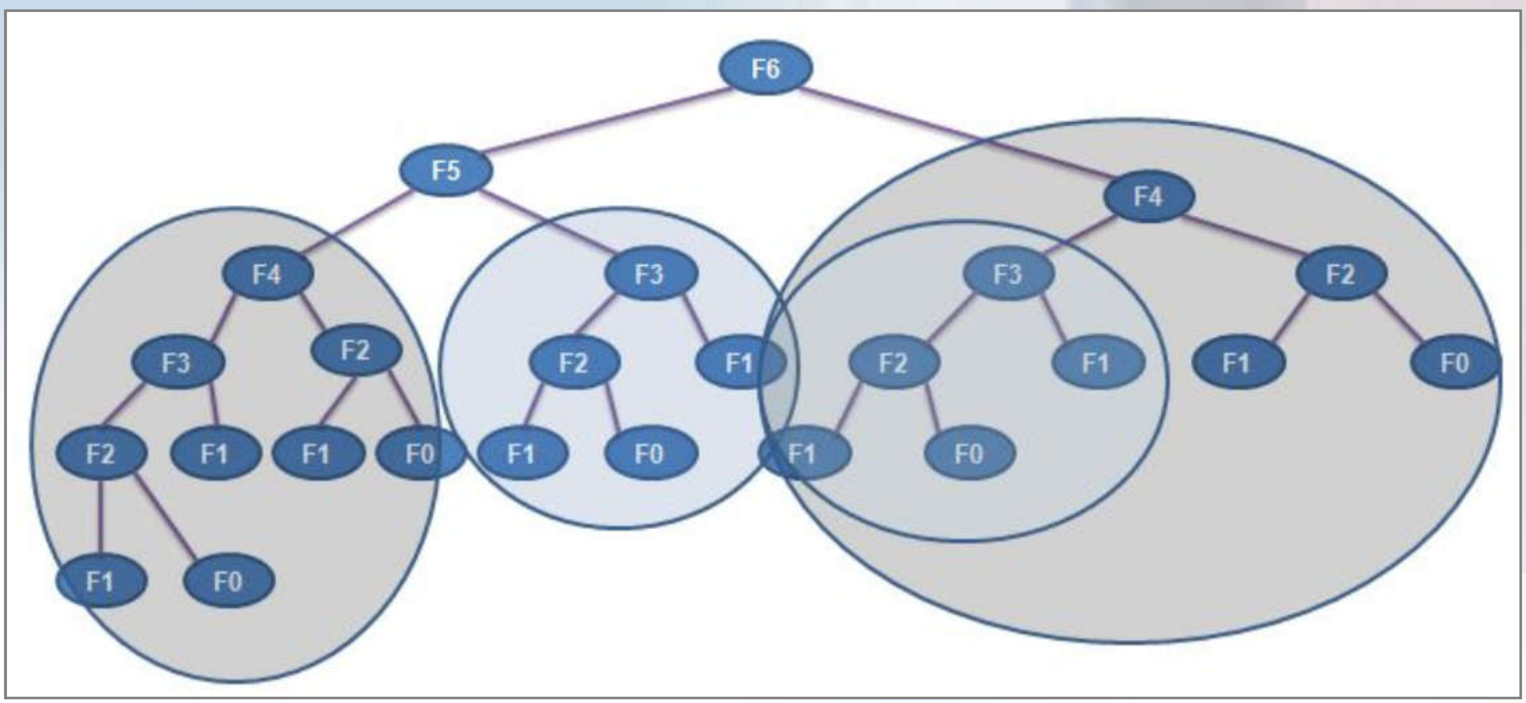
n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

Algoritmo recursivo para F_n:

```
FIBO-REC (n)
se n ≤ 1
    então devolva n
    senão a ← FIBO-REC (n − 1)
          b ← FIBO-REC (n − 2)
          devolva a + b
```

O Problema de Fibonacci (2)

- Consumo de Tempo T(n) é **Exponencial**
- Algoritmo resolve subproblemas muitas vezes
- **Exemplo:** Superposição de subproblemas ao determinar o 6º número de Fibonacci



O Problema de Fibonacci (3)

- Resolução do Fibonacci com **Programação Dinâmica – Bottom-Up:**

FIBO (n)
1 $f[0] \leftarrow 0$
2 $f[1] \leftarrow 1$
3 **para** $i \leftarrow 2$ **até** n **faça**
4 $\quad f[i] \leftarrow f[i-1] + f[i-2]$
5 **devolva** $f[n]$

- Consumo de tempo é O(n)

O Problema de Fibonacci (4)

- Resolução do Fibonacci com **Programação Dinâmica – Top-Down**:

```
MEMOIZED-FIBO (f, n)
  para i ← 0 até n faça
       f[i] ← −1
  devolva LOOKUP-FIBO (f, n)

LOOKUP-FIBO (f, n)
  se f[n] ≥ 0
       então devolva f[n]
  se n ≤ 1
       então f[n] ← n
       senão f[n] ← LOOKUP-FIBO(f, n − 1)
                    + LOOKUP-FIBO(f, n − 2)
  devolva f[n]
```

- Não recalcula valores de Fibonacci

Programação Dinâmica – Top-Down vs. Bottom-Up

- A **Top-Down** utiliza **memorização**, ou seja, registra valores para buscá-los posteriormente se necessário

- A **Bottom-Up** normalmente depende de uma noção de **“tamanho” do problema** e de uma **noção de “ordem” dos subproblemas**, tal que resolver um subproblema em particular depende de termos resolvidos os subproblemas menores anteriores

- As duas abordagens normalmente geram algoritmos com tempo de execução assintoticamente iguais, exceto em circunstâncias em que a Top-Down não examina todos os subproblemas recursivamente

- A PD Bottom-Up possui constantes melhores normalmente, devido ao menor overhead por chamadas recursivas de funções

Estudando problemas clássicos de Programação Dinâmica...

O PROBLEMA DA MOCHILA 0-1 (*KNAPSACK PROBLEM*)

O Problema da Mochila 0-1 (1)

- Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens distintos e únicos (enumerados de 0 a n-1), cada um com um peso w_i (inteiro) e valor c_i associado a cada item i, queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade. Para cada item, devemos escolher se ele estará incluso na solução ou não.

- **Exemplo:**
  - W = 10, n = 4, w = [8, 1, 5, 4] e V = [500, 1000, 300, 210]
  - Resposta: 1510, selecionando os itens 2, 3 e 4 com peso total de 10.

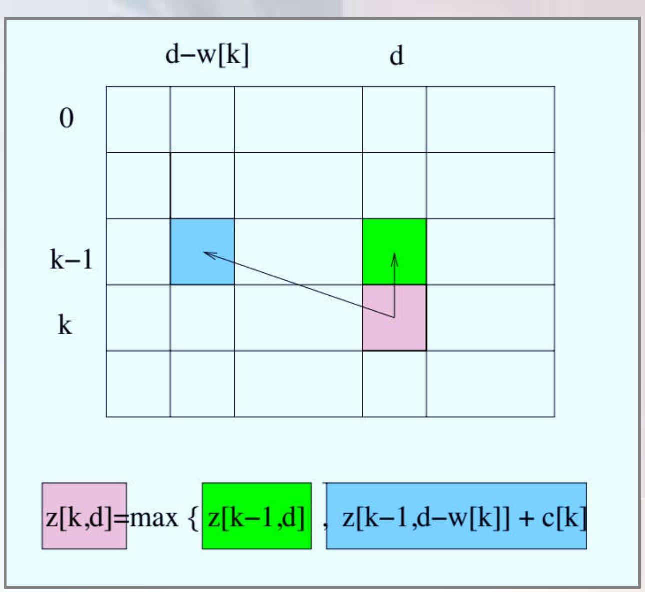
O Problema da Mochila 0-1 (2)

- Como podemos projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens → algoritmo de força bruta é **impraticável**
- É um problema de otimização. Será que tem subestrutura ótima?
  - Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens
  - Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens

O Problema da Mochila 0-1 (3)

- Seja z[k,d] o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original

- **A fórmula de recorrência para computar z[k,d] para todo valor de d e k é**:
  - $z[0,d] = 0$
  - $z[k,0] = 0$
  - $z[k,d] =$
    - $z[k-1, d]$, se $w_k > d$
    - $\max\{z[k-1,d], z[k-1, d-w_k]+c_k\}$, se $w_k \leq d$



O Problema da Mochila 0-1 (4)

- **Algoritmo:**

Mochila(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n.
▷ **Saída:** O valor máximo do total de itens colocados na mochila.

1. **para** $d := 0$ **até** W **faça** $z[0, d] := 0$
2. **para** $k := 1$ **até** n **faça** $z[k, 0] := 0$
3. **para** $k := 1$ **até** n **faça**
4.   **para** $d := 1$ **até** W **faça**
5. $z[k, d] := z[k-1, d]$
6.   **se** $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
7. $z[k, d] := c_k + z[k-1, d-w_k]$
8. **devolva** $(z[n, W])$

O Problema da Mochila 0-1 – Exemplo (1)

- Exemplo: c = {10, 7, 25, 24}, w = {2, 1, 6, 5} e W = 7

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

O Problema da Mochila 0-1 – Exemplo (2)

- **Exemplo: c = {10, 7, 25, 24}, w = {2, 1, 6, 5} e W = 7**

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

O Problema da Mochila 0-1 – Exemplo (3)

- **Exemplo: c = {10, 7, 25, 24}, w = {2, 1, 6, 5} e W = 7**

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

O Problema da Mochila 0-1 – Exemplo (4)

- **Exemplo: c = {10, 7, 25, 24}, w = {2, 1, 6, 5} e W = 7**

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

O Problema da Mochila 0-1 – Exemplo (5)

- **Exemplo: c = {10, 7, 25, 24}, w = {2, 1, 6, 5} e W = 7**

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

O Problema da Mochila 0-1 – Exemplo (6)

- Exemplo: c = {10, 7, 25, 24}, w = {2, 1, 6, 5} e W = 7

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

O Problema da Mochila 0-1: Complexidade (1)

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é O(n.W)
- É um algoritmo **pseudo-polinomial**: sua complexidade depende do valor de W, parte da entrada do problema
  - É importante salientar que o problema da mochila 0-1 é NP-Difícil, e por pseudo-polinomial, significa que é polinomial no valor numérico da entrada, porém, exponencial no comprimento da codificação mesma
  - Especificamente, W não é polinomial em relação ao comprimento da sua codificação.
    - Por exemplo, seja W = 1.000.000.000.000: são necessários 40 bits para representar este número, porém, o tempo de execução considera o fator 1.000.000.000.000, que é $O(2^{40})$, ou seja, $O(2^{\log W})$.
    - Na verdade, a complexidade do algoritmo é mais precisamente expressa por $O(n2^{\log W})$, ou seja, exponencial
- O algoritmo não devolve o subconjunto de valor total máximo, apenas o valor máximo
  - É fácil recuperar o subconjunto a partir da tabela preenchida

Estudando sobre...

TEORIA DA COMPLEXIDADE COMPUTACIONAL

Avisos

- O assunto tratado nesta aula será devidamente aprofundado na Disciplina de Linguagens Formais e Autômatos do 4º Semestre do Bacharelado em Ciência da Computação

- Não pretende ser completo tecnicamente e historicamente

Teoria da Complexidade Computacional (1)

- Quase todos algoritmos vistos até aqui possuem tempo polinomial: em entradas de tamanho n, o tempo de execução no pior caso é $O(n^k)$
- Pode-se pensar, intuitivamente, que todos os problemas podem ser resolvidos em tempo polinomial
- Entretanto, a **resposta é não**. Há problemas que não podem ser resolvidos pelo computador, e outros que podem, porém, não em tempo $O(n^k)$
- Geralmente, designamos os problemas resolvíveis em tempo polinomial como **tratáveis**
- Por outro lado, designamos os problemas resolvíveis apenas em tempo supra polinomial como **intratáveis**, ou **difíceis**

Teoria da Complexidade Computacional (2)

- Um problema é considerado **indecidível** caso não seja possível criar um algoritmo qualquer para sua solução. Exemplo: Problema da Parada
  - “Dadas uma descrição de um programa e uma entrada finita, decida se o programa termina de rodar ou rodará indefinidamente, dada essa entrada.”

- Um problema é considerado **intratável** caso não haja algoritmo em tempo polinomial que o resolva deterministicamente

- Em geral, considera-se que os **problemas resolvíveis em tempo polinomial** são **tratáveis**, mas por razões filosóficas, e não matemáticas:
  - Embora seja razoável considerar intratável um problema que exige tempo $O(n^{100})$, um número muito pequeno de problemas práticos exigem tempo de ordem tão alta

Teoria da Complexidade Computacional (3)

- Uma classe importante de problemas, denominada **NP-completo**, possui **status desconhecido**

- **Não se desenvolveu nenhum algoritmo de tempo polinomial para nenhum problema desta classe**, porém, **não se provou também a impossibilidade de tal algoritmo existir** – eis a famosa questão **P vs. NP?**

- Vários problemas NP-completos são interessantes porque se parecem muito com problemas fáceis:
  - Caminho mais curto vs. Caminho mais longo
  - Ciclo Euleriano vs. Ciclo Hamiltoniano
  - 2-SAT vs. 3-SAT

Teoria da Complexidade Computacional (4)

- A Computabilidade e a Teoria da Complexidade Computacional estudam os limites da computação:
  - Quais problemas jamais poderão ser resolvidos por um computador, independente da sua velocidade ou memória?
  - Quais problemas podem ser resolvidos por um computador, mas requerem um período tão extenso de tempo para completar a ponto de tornar a solução impraticável?
  - Em que situações pode ser mais difícil resolver um problema do que verificar cada uma das soluções manualmente?

Teoria da Complexidade Computacional (5)

- **Problema de Decisão:**
  - Tipo de problema computacional em que a resposta para cada instância é **sim** ou **não**
  - "Dadas uma lista de cidades e as distâncias entre todas, há uma rota que visite todas as cidades e retorne à cidade original com distância total menor do que 500 km?"

- **Problema de Otimização:**
  - Tipo de problema computacional em que é necessário determinar a **melhor** solução possível entre todas as soluções viáveis
  - "Dadas uma lista de cidades e as distâncias entre todas, determine a menor rota que visite todas as cidades e retorne à cidade original."

Teoria da Complexidade Computacional (6)

- **Problemas de Decisão e Otimização:**
  - Geralmente, os problemas de decisão não são mais difíceis que os problemas de otimização
  - Se pudermos caracterizar o problema de decisão como difícil, também estaremos caracterizando o problema de otimização relacionado como difícil
- **NP-Completude:**
  - A teoria da NP-Completude, por conveniência, se aplica a problemas de decisão
  - A razão pela qual nos concentraremos nos problemas de decisão é a propriedade natural de codificação por **linguagens formais (Matéria do Próximo Semestre)**

Teoria da Complexidade Computacional (7)

- Para um programa de computador resolver um problema abstrato, temos de representar as instâncias de um problema de modo que o programa entenda

- Uma **codificação** consiste em um mapeamento de objetos **abstratos para cadeias**, por exemplo, binárias

- Um algoritmo **resolve** um problema em tempo O(T(n)) se, dada uma instância i de comprimento n = |i|, o algoritmo gerar a solução em tempo máximo T(n)

- Utilizando o conceito de **codificação**, pode-se criar uma codificação mais geral e independente para problemas, denominadas **linguagens formais**

Teoria da Complexidade Computacional (8)

- **Linguagens:**
  - Para um conjunto finito de símbolos (ou **alfabeto**) Σ, denotamos por Σ* o conjunto de todas as cadeias finitas de símbolos de Σ
  - Por exemplo, para Σ = {0, 1}, temos que Σ* é composto por: Null (Vazio), 0, 1, 00, 11, e todas as cadeias finitas de 0s e 1s
  - Um subconjunto L de Σ* é chamado de **linguagem** sobre o alfabeto Σ

- **Aceitação e Rejeição:**
  - Um algoritmo A **aceita** uma cadeia x ϵ Σ* se dada a entrada x, a saída do algoritmo é A(x) = 1
  - O mesmo algoritmo A **rejeita** uma cadeia x se A(x) = 0.
  - A linguagem aceita por um algoritmo A é o conjunto de cadeias **L = { x ϵ {0, 1}* : A(x) = 1}**

Teoria da Complexidade Computacional (9)

- Ainda que a linguagem L seja aceita por um algoritmo A, o algoritmo não necessariamente rejeitará uma cadeia $x \notin L$ dada como entrada
  - Por exemplo, o algoritmo pode simplesmente entrar em loop infinito

- Uma linguagem L é **decidida** por um algoritmo A **se toda cadeia em L** é **aceita** por A e **toda cadeia não pertencente** a L é **rejeitada** por A

- Uma linguagem L é **aceita em tempo polinomial** por um algoritmo A se for aceita por A propriamente e se houver uma constante k tal que, para qualquer cadeia $x \in L$ de comprimento n, A aceita x em tempo $O(n^k)$.

- Uma linguagem L é **decidida em tempo polinomial** por um algoritmo A se houver uma constante k tal que, para qualquer cadeia $x \in \Sigma^*$ de comprimento n, A decide corretamente se $x \in L$ em tempo $O(n^k)$

Máquinas de Turing (1)

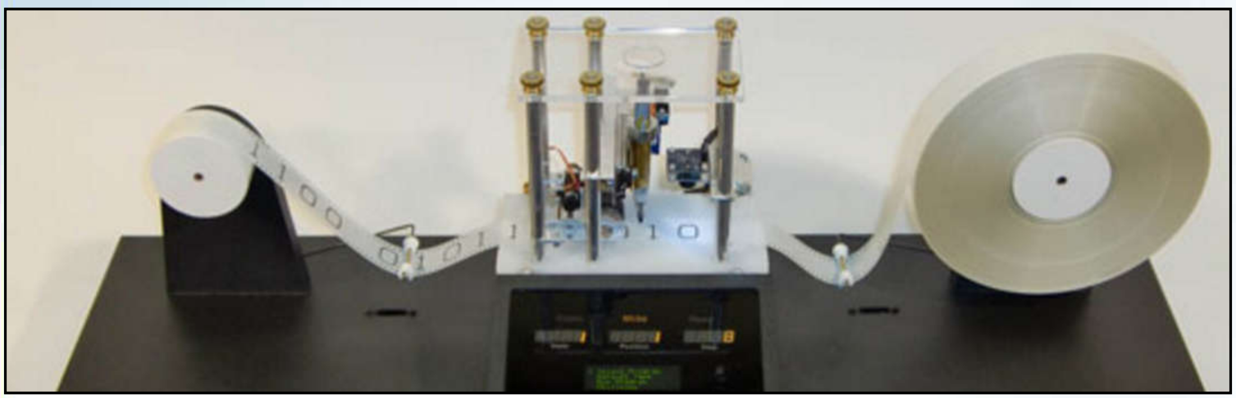
- Matemático, lógico e criptoanalista inglês
- Participação importante na II Guerra Mundial
- **Pai da Ciência da Computação**
  - Dedicou a vida à teoria da computabilidade;
  - Formalizou os conceitos de algoritmo e computabilidade
  - Aos 24 anos (1936), criou a Máquina de Turing
  - Parte de sua vida foi retratada no filme Breaking the Code (1996) e no filme The Imitation Game (2014)
- **Hoje o Prêmio Turing equivale ao Nobel da Computação**

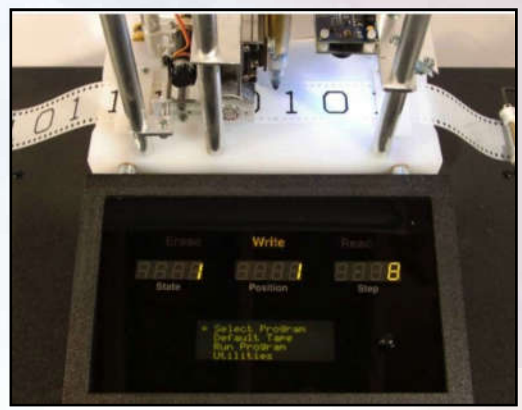
Alan Mathison Turing

Máquinas de Turing (2)

- A noção de algoritmos pode ser formalizada utilizando uma máquina universal chamada **Máquina de Turing Determinística**:
  - Funciona como computadores elementares, emulando a parte lógica
  - Foi idealizada muitos anos antes dos computadores digitais
  - O determinismo vem do fato de podermos predizer seu comportamento

- Opera sobre linguagens





Máquinas de Turing (3)

- Uma máquina de Turing é composta por (**Definição**):
  - Uma **fita** infinita nos dois sentidos, dividida em células contíguas com os dados de entrada, um símbolo por célula
  - Um **cabeçote** de leitura e escrita na fita
  - Um **registrador de estados**, que indica o estado atual da máquina
  - Uma **função de transição**, que dados o símbolo lido na fita e o estado atual, indica o que deve ser escrito, em qual direção o cabeçote deve se mover e qual será seu novo estado

Máquinas de Turing (4)

- Um programa para uma máquina de Turing especifica:
  - Um **conjunto finito Γ de símbolos da fita**, incluindo os símbolos de Σ e o símbolo especial “em branco” b ou vazio
  - Um **conjunto finito Q de estados**, incluindo o estado inicial q_0 e estados finais q_s e q_n;
  - Uma **função de transição δ** que, a partir do estado atual e o símbolo lido na fita, determina qual é o estado seguinte, qual símbolo deve ser escrito na fita e em qual direção o cabeçote deve se movimentar:

$$\delta : (Q - \{q_s, q_n\} \times \Gamma \rightarrow Q \times \Gamma \times \{-1, +1\}$$

Máquinas de Turing (5)

- **Exemplo:**
  - Suponhamos **$\delta(q, s) = (q', s', \Delta)$**
  - No estado q, ao ler o símbolo s, a máquina de Turing vai para o estado q', escreve s' no lugar de s e se movimenta para a direita (+1) ou esquerda (-1), dependendo do valor de Δ

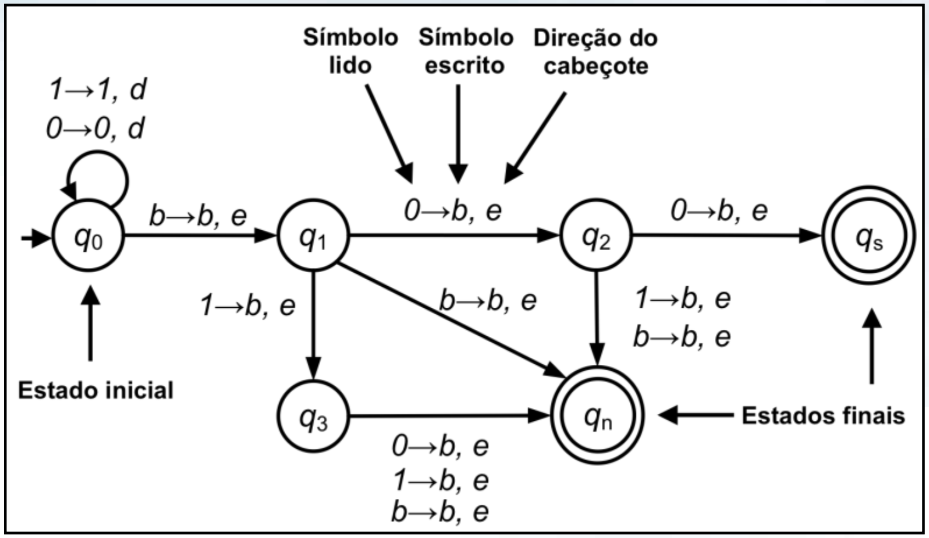
Máquinas de Turing (6)

- Dada uma cadeia de símbolos de entrada, com início na célula número 1:
  - A partir do estado inicial, a execução é passo-a-passo:
    - Se um estado final foi atingido, a computação terminou
    - Caso contrário, existe algum símbolo a ser lido na fita
    - Lido um símbolo, a função de transição informa, de acordo com o estado atual, o que deve ser escrito, em qual posição o cabeçote deve se mover e qual será seu novo estado
- Máquinas de Turing também podem ser representadas por diagramas de estados, em que todas as informações estão contidas

Máquinas de Turing (7)

- Exemplo em que $\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$ e $Q = \{q_0; q_1; q_2; q_3; q_s; q_n\}$.

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_n, b, -1)$
q_2	$(q_s, b, -1)$	$(q_n, b, -1)$	$(q_n, b, -1)$
q_3	$(q_n, b, -1)$	$(q_n, b, -1)$	$(q_n, b, -1)$



Máquinas de Turing (8)

- Dizemos que um programa M com alfabeto Σ **aceita** x pertencente a Σ* se e somente se a computação termina no estado terminal (estado q_s) quando a entrada é x
- A **linguagem** L_M reconhecida pelo programa M é definida por todas as cadeias de símbolos x pertencentes a Σ* tal que M aceita x
- M não aceita todas as cadeias em Σ*, apenas aquelas pertencentes a LM → as demais ou param no estado q_n ou não param.
- **A correspondência entre aceitar uma linguagem e resolver problemas de decisão é direta**
- Uma Máquina de Turing tem **03 aplicações gerais:**
  - Reconhecer linguagens
  - Calcular funções
  - Processar problemas de decisão

Máquinas de Turing (09)

- **Tempo Polinomial – Definição:**
  - Um **problema é solucionável em tempo polinomial determinístico** se **existir uma máquina de Turing determinística** que o **solucione em tempo limitado por um polinômio** em relação ao tamanho da entrada
  - A **máquina de Turing determinística** equivale a **um algoritmo determinístico**

Classes de Problemas (1)

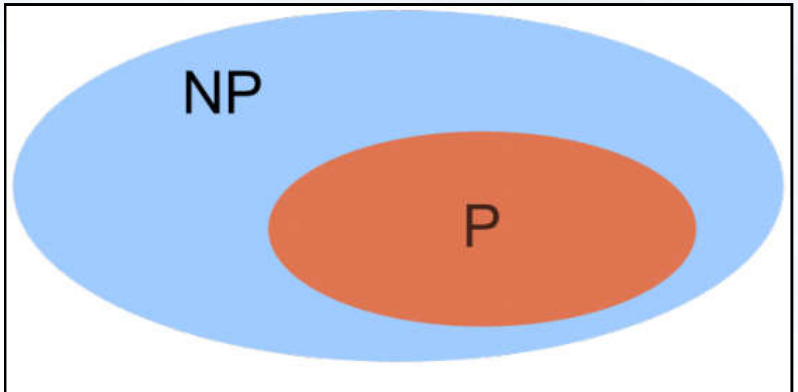
- **Classe P:**
  - Consiste nos problemas que podem **ser resolvidos deterministicamente em tempo polinomial** no tamanho da entrada, ou seja, existem **algoritmos de complexidade $O(n^k)$** para k constante que os resolvam
  - P é uma referência a **tempo determinístico polinomial**
  - **Exemplos:** pesquisa, ordenação, busca em grafos, menor caminho em grafos, fluxo máximo em grafos, detecção de árvores geradoras mínimas e classificação de arestas e vértices

Classes de Problemas (2)

- **Classe NP:**
  - Consiste nos problemas que são **verificáveis em tempo polinomial**. Dada uma solução para o problema, podemos verificar se é uma solução válida em tempo polinomial
  - NP é uma referência a **Tempo Não Determinístico Polinomial**
  - Em geral, é mais difícil resolver um problema do que verificar uma dada solução. Isto leva alguns teóricos a acreditarem que há problemas em NP que não estão em P

Classes de Problemas (3)

- **P vs. NP:**
  - É possível **verificar uma solução** para um **problema da classe P** em **tempo polinomial**, logo, **qualquer problema pertencente a P pertence a NP**
  - A questão é, P é ou não um subconjunto próprio de NP? Mais diretamente, **P=NP?**
  - Talvez a razão mais forte para que se acredite que P **seja diferente** NP seja a existência de problemas **NP-Completos**



Uma possível visão se P≠NP.

Classes de Problemas (4)

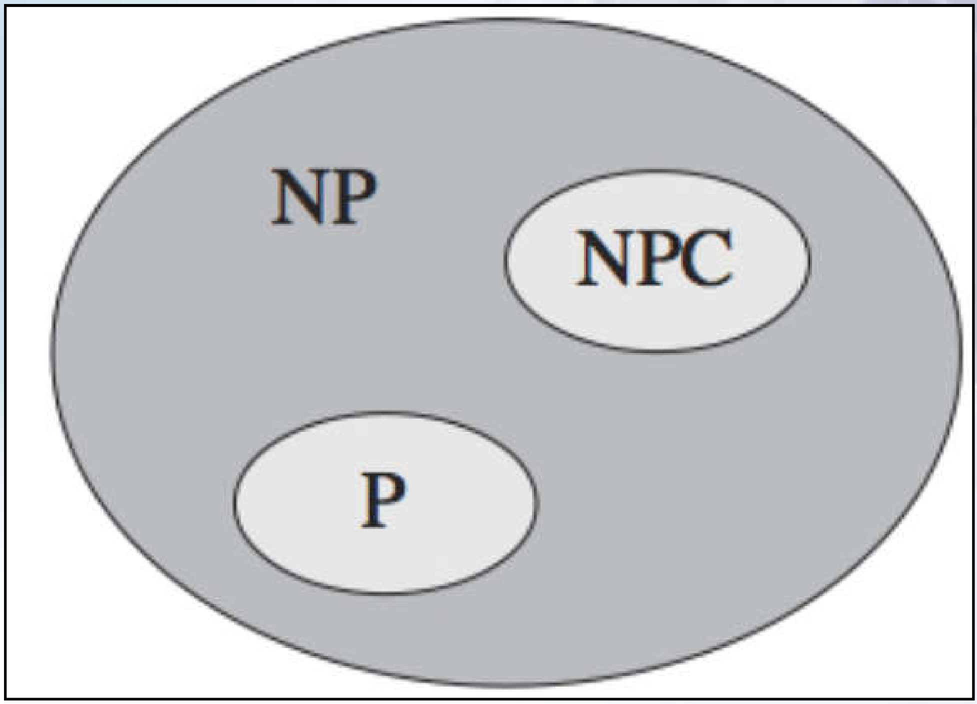
- **NP Completo - NPC (Definição Informal):**
  - Informalmente, se um problema está na classe NP-Completo, ele está em NP e é **"tão difícil"** quanto qualquer problema em NP
  - Os problemas da classe NPC possuem uma estreita relação entre si, de modo que **se um deles for resolvido em tempo polinomial**, **todos o serão, e consequentemente, P=NP**
- **Complexidade:**
  - Existe uma forte corrente que acredita que os problemas **NP-Completos** são **intratáveis**, uma vez que não existe avanço significativo na direção contrária
  - Desta forma, **P e NP seriam diferentes**, **mas não é possível concluir nada**
  - Em certo sentido, os problemas **NP-Completos são os mais difíceis em NP**
  - Uma forma de comparar a dificuldade relativa entre problemas **é a redutibilidade em tempo polinomial**

Classes de Problemas (5)

- **NP Completo (Definição Formal):**
  - Um problema de decisão Q é NP-Completo se:
    1. Q ϵ NP
    2. Q’ é polinomialmente redutível a Q para todo Q’ ϵ NP
  - Se é descoberto um algoritmo determinístico polinomial para um problema NP-Completo, ele se torna um problema P, e de acordo com as propriedades acima, todos os outros problemas em NPC também o serão
  - As pesquisas sobre P vs. NP se concentram nos problemas NP-Completos por esse motivo
- **NP-Difícil (NP-Hard):**
  - Se um problema satisfaz a propriedade 2, mas não necessariamente a propriedade 1, este pertence à classe **NP-Difícil**

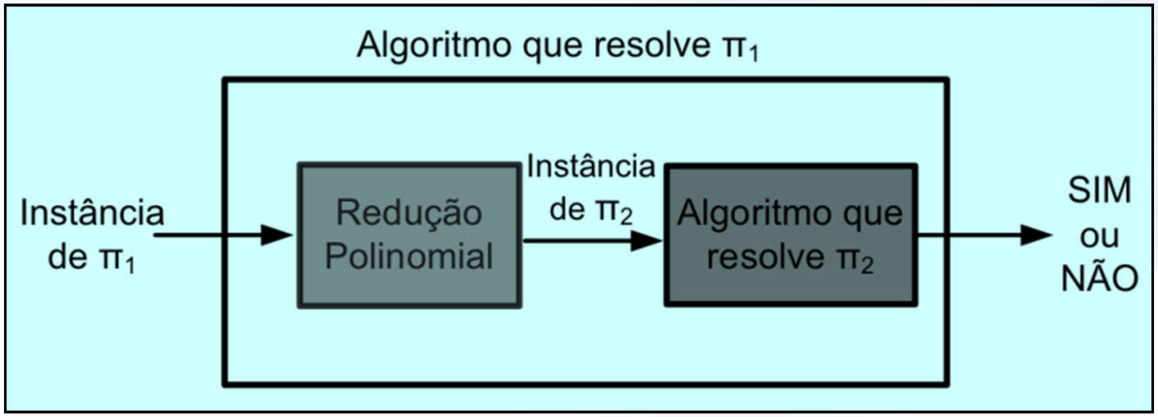
Classes de Problemas (6)

- Uma possível relação entre P, NP e NP-Completo



Redução de Problemas (1)

- **Redutibilidade em Tempo Polinomial:**
  - Um problema π_1 pode ser reduzido a um problema π_2 se qualquer instância de π_1 puder ser **"facilmente reformulada"** como uma instância de π_2: **a resposta obtida por π_2 deve ser idêntica à que seria obtida por π1**
  - O **algoritmo de redução** deve ser **polinomial**
  - Se um problema π_1 **é redutível** a um problema π_2, então π_1 **não é mais difícil** que π_2
  - Com efeito, π_2 é considerado pelo menos tão difícil quanto o π_1, dentro de um fator polinomial



Redução de Problemas (2)

- **Redutibilidade em Tempo Polinomial:**
  - Por exemplo, o problema de resolver uma equação linear se reduz ao problema de resolver uma equação quadrática:
    - As instâncias do tipo $ax + b = 0$ são transformadas em $0x^2 + ax + b = 0$
  - Quando um problema π1 é polinomialmente redutível a um problema π2 denotamos por $\pi 1 \leq_p \pi 2$

- **Redutibilidade e Linguagens:**
  - Os mesmos conceitos se aplicam à linguagens:
    - Uma linguagem L_1 pode ser transformada em uma linguagem L_2
    - A redução deve ser computada por uma Máquina de Turing polinomial
    - Cadeias de símbolos só pertencem a L_1 se pertencerem a L_2
    - Consequentemente, L_1 e L_2 pertencem à mesma classe de complexidade

Teorema de Cook (1)

- Matemático
- Cientista da Computação;
- Professor da Universidade de Toronto
- **Pai da Teoria da Complexidade Computacional**
  - Formalizou a noção de redução em tempo polinomial
  - Formalizou o conceito de NP-Completo
  - Identificou o primeiro problema NP-Completo;
- Autor do **Teorema de Cook**
- Pelo teorema, **recebeu o Prêmio Turing em 1982**

Stephen Arthur Cook

Teorema de Cook (2)

- **Definição:**
  - A definição de problemas NP-Completos é de certa forma “recursiva”.
  - Então qual é o caso base? Qual é o problema NP-Completo original?

- **SAT:**
  - O **Problema de Satisfabilidade Booleana (SAT)** foi o primeiro problema caracterizado como NP-Completo.
  - Dada uma expressão lógica com n variáveis booleanas e m conectivos lógicos NOT (¬), AND (^) e OR (v), é necessário determinar se há uma atribuição satisfatória de valores às variáveis, ou seja, que resulte em valor 1 (ou verdadeiro)
  - O **Teorema de Cook** nos diz que este problema de decisão é NP-Completo

Teorema de Cook (3)

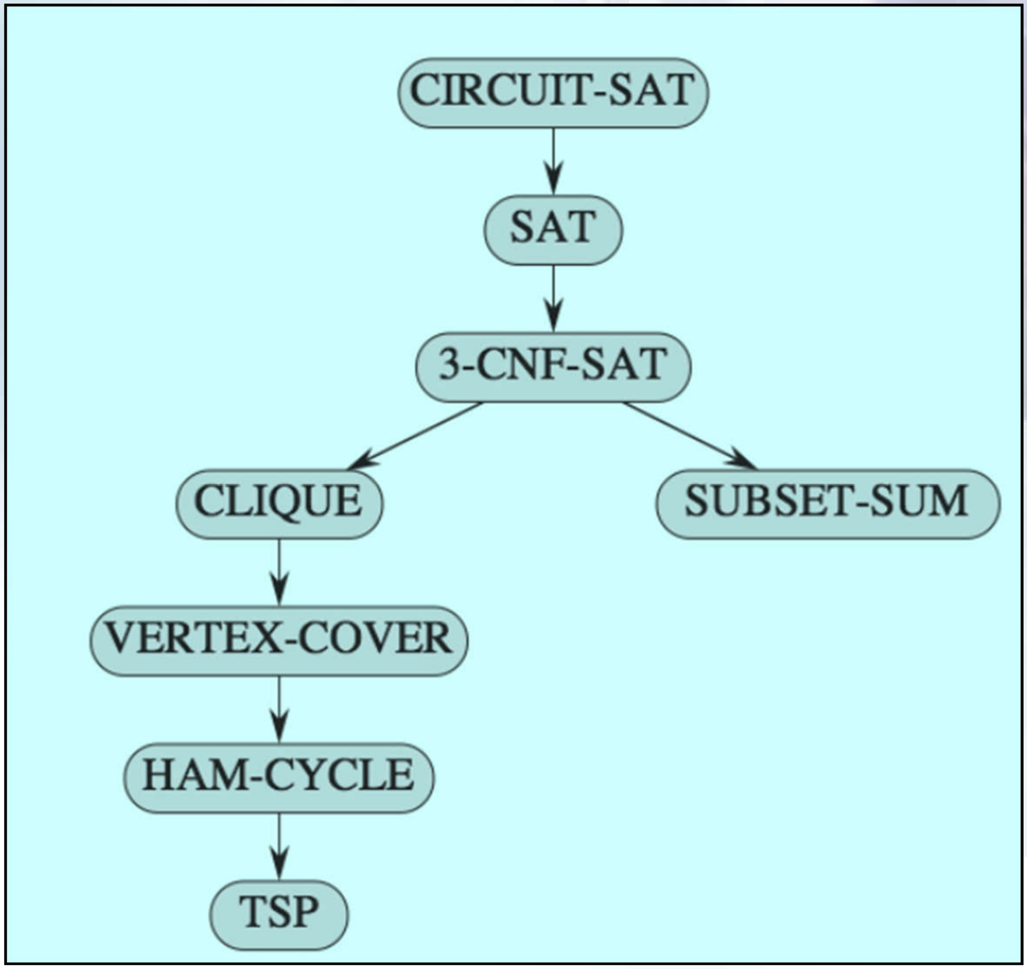
- O Teorema de Cook (1971) também é conhecido como Teorema de Cook-Levin, por também ser atribuído independentemente ao russo/americano Leonid Levin
- Levin publicou em 1973 um artigo que considerava problemas de busca, provando haver **6 problemas universais** (equivalentes aos NP-Completos), embora existam menções em anos anteriores a este trabalho
- O Teorema de Cook, descrito em um artigo de pouco mais do que 7 páginas, define o primeiro problema NP-Completo: o problema de satisfabilidade booleana

Teorema de Cook (4)

- **Definição:**
  - Resumidamente, Cook definiu uma linguagem L_{SAT} e também uma linguagem geral de problemas NP, reconhecida por uma **Máquina de Turing** não determinística polinomial genérica
  - Posteriormente, foi provado que todas as linguagens L de problemas NP se reduzem a L_{SAT}, logo, a Máquina de Turing não determinística polinomial reconhece L_{SAT}
  - Satisfazendo-se as propriedades exigidas, provou que **SAT é o primeiro problema NP-Completo**

Teorema de Cook (5)

- Esquema da Prova dos Problemas NP-Completos



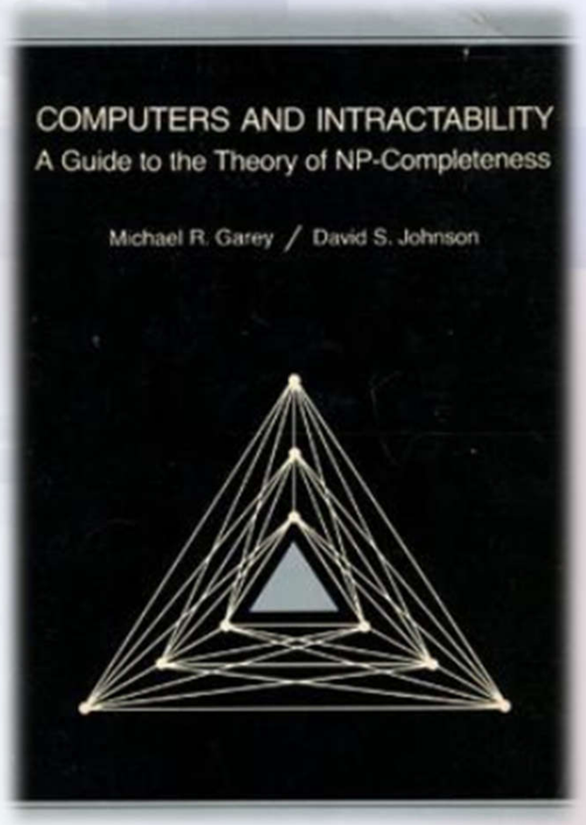
Problemas Intratáveis (1)

- Em 1971, **Richard Karp** identificou os primeiros 21 problemas da classe e contribuiu para o desenvolvimento da teoria da NP-Completude

- Posteriormente, centenas de outros problemas foram identificados por outros pesquisadores

Problemas Intratáveis (2)

- Michael R. Garey and David S. Johnson. 1979.
- Computers and Intractability: A Guide to the Theory of Np-Completeness. W. H. Freeman & Co., New York, NY, USA
- Foi o primeiro livro a tratar exclusivamente de NP-completude e intratabilidade computacional
- Apresenta um apêndice fornecendo um material exaustivo dos problemas NP-completos
- Considerado como um clássico: em um estudo de 2006, o CiteSeer listou o livro como a referência mais citada na literatura de ciência da computação



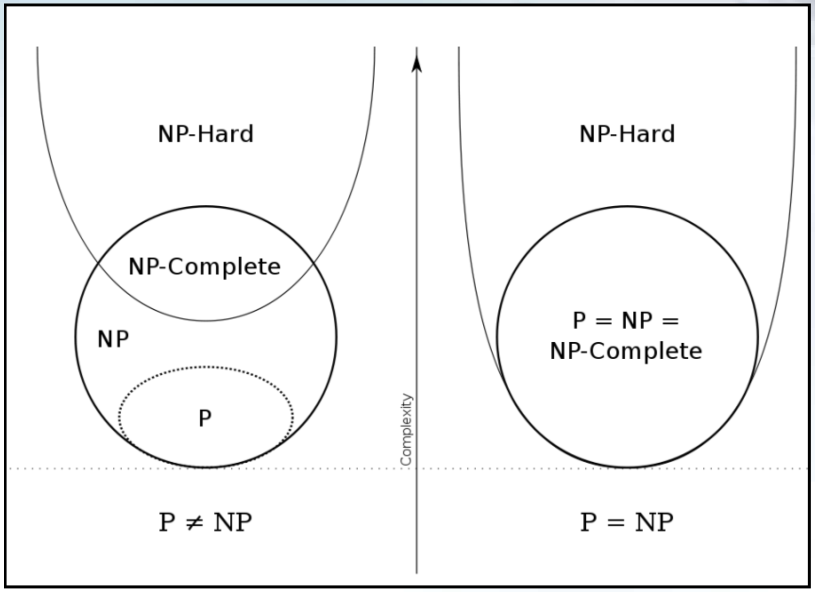
Discussões (1)

- A maioria dos problemas de interesse pertencem comprovadamente à classe NP-Completo:
  - Determinar a sequência de DNA que melhor se assemelha a um fragmento de DNA
  - Determinar procedimentos eficientes para predição de estrutura de proteínas
  - Determinar se uma afirmação matemática possui uma prova curta

Discussões (2)

- **P vs. NP:**
  - A partir das descobertas sobre os problemas da classe NP-Completo, grande parte dos cientistas da computação passou a acreditar que P **é diferente** NP
  - **Provar isto se tornou a questão mais importante da ciência da computação** e uma das mais importantes da matemática

- **Suposições:**



Discussões (3)

- **P vs. NP – Um dos Problemas do Milênio:**
  - O Clay Mathematics Institute elencou 7 problemas matemáticos e oferece um prêmio de um milhão de dólares para quem resolver um deles.
  - Provar que P é igual NP ou P é diferente de NP é um dos 7 Problemas do Milênio desde o ano 2000:
    - P vs. NP
    - A conjectura de Poincaré (resolvido por Grigori Perelman em 2006);
    - A conjectura de Hodge;
    - A hipótese de Riemann;
    - A existência de Yang-Mills e a falha na massa;
    - A existência e suavidade de Navier-Stokes;
    - A conjectura de Birch e Swinnerton-Dyer

Discussões (4)

- E Se P = NP?
  - **Várias tarefas se tornariam triviais:**
    - Transporte de pessoas e produtos mais rápido e mais barato
    - Indústrias produzindo mais rápido e mais barato
    - Traduções automáticas
    - Reconhecimento de visão
    - Compreensão de linguagens
    - Previsão do tempo, terremotos e tsunamis
    - Provas curtas para teoremas matemáticos
  - **Adeus a criptografia:**
    - A criptografia se baseia em problemas difíceis de serem resolvidos, como a fatoração de números muito grandes em números primos.
    - Portanto, é impossível de quebrar, a não ser que P=NP e fatoração seja um problema trivial...

Discussões (5)

- **P vs. NP Atualmente**
  - Existem 116 provas sobre P vs. NP, registradas pelo P vs. NP
  - Em 2016 existiu seis tentativas (igual:4, diferente:2)
  - De 26 de setembro de 2016 em diante, nenhuma atualização

Fim do Curso

Lance Fortnow, Georgia Tech

"P$\neq$NP. It will be resolved in an unpredictably long time. If I knew what kinds of techniques would be used I wouldn't tell."

Eric Allender, Rutgers University

"P $\neq$ NP will be resolved within 25 years, though this estimate is completely meaningless, of course.

Techniques: If I knew, then I wouldn't tell you."

Richard Karp

"I believe intuitively P $\neq$ NP but it is only an intuition."

Stephen Cook, Universidade de Toronto

"P $\neq$ NP will not be resolved in the next 20 years and will need new techniques".