Algoritmos e complexidade
Notas de aula

Marcus Ritt
Luciana Buriol
Edson Prestes

22 de Abril de 2008

Conteúdo

Essas notas servem como suplemento à material do livro “Complexidade de algoritmos” de Toscani/Veloso e o material didático da disciplina “Complexidade de algoritmos” da UFRGS.

Versão 2561 do 2008-04-22, compilada em 22 de Abril de 2008.

Parte I.

Análise de algoritmos

1. Introdução e conceitos básicos

A teoria da computação começou com a pergunta "Quais problemas são *efetivamente* computáveis?" e foi estudado por matemáticos como Post, Church, Kleene e Turing. A nossa intuição é que todos os computadores diferentes, por exemplo um PC ou um Mac, têm o mesmo poder computacional. Mas não é possível imaginar algum outro tipo de máquina que seja mais poderosa que essas? Cujos programas nem podem ser implementadas num PC ou Mac? Não é fácil de resolver essa pergunta, porque a resposta a ela depende das possibilidades computacionais em nosso universo, e logo do nosso conhecimento da física. Mesmo enfrentando esses problemas, matemáticos inventaram vários modelos de computação, entre eles o cálculo lambda, as funções parcialmente recursivas, a máquina de Turing e a máquina de RAM, que se tornaram todos equivalente em poder computacional, e são considerados como máquinas universais.
Nossa pergunta é mais específica: "Quais problemas são *eficientemente* computáveis?". Essa pergunta é motivada pela observação de que alguns problemas que, mesmo sendo efetivamente computáveis, são tão complicados, que a solução deles para instâncias do nosso interesse é impraticável.

Exemplo 1.1
Não existe um algoritmo que decide o seguinte: Dado um programa arbitrário (que podemos imaginar escrito em qualquer linguagem de programação como C ou Miranda) e as entradas desse programa. Ele termina? Esse problema é conhecido como "problema de parada". ◊

Visão geral

- Nosso objetivo: Estudar a análise e o projeto de algoritmos.
- Parte 1: Análise de algoritmos, i.e. o estudo teórico do desempenho e uso de recursos.
- Ela é pré-requisito para o projeto de algoritmos.
- Parte 2: As principais técnicas para projetar algoritmos.

Introdução

- Um algoritmo é um procedimento que consiste em um conjunto de regras não ambíguas as quais especificam, para cada entrada, uma seqüência finita de operações, terminando com uma saída correspondente.
- Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta correta, se forem concedidos tempo e memória suficientes para a sua execução.

Motivação

- Na teoria da computação perguntamos "Quais problemas são efetivamente computáveis?"
- No projeto de algoritmos, a pergunta é mais específica: "Quais problemas são eficientemente computáveis?"
- Para responder, temos que saber o que "eficiente" significa.
- Uma definição razoável é considerar algoritmos em tempo polinomial como eficiente (tese de Cobham-Edmonds).

Custos de algoritmos

- Também temos que definir qual tipo de custo é de interesse.
- Uma execução tem vários custos associados: Tempo de execução, uso de espaço (cache, memória, disco), consumo de energia, ...
- Existem características e medidas que são importantes em contextos diferentes Linhas de código fonte (LOC), legibilidade, manutenabilidade, corretude, custo de implementação, robustez, extensibilidade,...
- A medida mais importante e nosso foco: tempo de execução.
- A complexidade pode ser vista como uma propriedade do problema

Mesmo um problema sendo computável, não significa que existe um algoritmo que vale a pena aplicar. O problema

Expressões regulares com ^{2}

Instância Uma expressão regular e com operações $\cup$ (união), $\cdot^*$ (fecho de Kleene), $\cdot$ (concatenação) e $\cdot^2$ (quadratura) sobre o alfabeto $\Sigma = \{0, 1\}$.

Decisão $L(e) \neq \Sigma^*$?

que parece razoavelmente simples é, de fato, EXPSPACE-completo [23, Corolário 2.1] (no momento é suficiente saber que isso significa que o tempo para resolver o problema cresce ao menos exponencialmente com o tamanho da entrada).

Exemplo 1.2
Com $e = 0 \cup 1^2$ temos $L(e) = \{0, 11\}$.
Com $e = (0 \cup 1)^2 \cdot 0^*$ temos

$$L(e) = \{00, 01, 10, 11, 000, 010, 100, 110, 0000, 0100, 1000, 1100, \ldots\}.$$

$\Diamond$

Existem exemplos de outros problemas que são decidíveis, mas têm uma complexidade tão grande que praticamente todas instâncias precisam mais recursos que o universo possui (p.ex. a decisão da validade na lógica monádica fraca de segunda ordem com sucessor).

O universo do ponto da vista da ciência da computação Falando sobre os recursos, é de interesse saber quantos recursos nosso universo disponibiliza aproximadamente. A seguinte tabela contém alguns dados básicos:

Idade	$13.7 \pm 0.2 \times 10^9$ anos $\approx 43.5 \times 10^{16}$ s
Tamanho	$\geq 78 \times 10^9$ anos-luz
Densidade	$9.9 \times 10^{-30} g/cm^3$
Número de átomos	10^{80}
Número de bits	10^{120}
Número de operações lógicas elementares até hoje	10^{120}
Operações/s	$\approx 2 \times 10^{102}$

(Todos o dados correspondem ao consenso científico no momento; obviamente novos descobrimentos podem os mudar. Fontes principais: [21, 1])

Métodos para resolver um sistema de equações lineares Como resolver um sistema quadrático de equações lineares

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

ou $Ax = b$? Podemos calcular a inversa da matriz A para chegar em $x = bA^{-1}$. O *método de Cramer* nos fornece as equações

$$x_i = \frac{\det(A_i)}{\det(A)}$$

seja A_i a matriz resultante da substituição de b pela i-gésima coluna de A. (A prova dessa fato é bastante simples. Seja U_i a matriz identidade com a i-gésima coluna substituído por x: é simples verificar que $AU_i = A_i$. Com $\det(U_i) = x_i$ e $\det(A)\det(U_i) = \det(A_i)$ temos o resultado.) Portanto, se o trabalho de calcular o determinante de uma matriz de tamanho $n \times n$ é T_n, essa abordagem custa $(n+1)T_n$. Um método simples usa a fórmula de Leibniz

$$\det(A) = \sum_{\sigma \in S_n} \left(\operatorname{sgn}(\sigma) \prod_{1 \le i \le n} a_{i,\sigma(i)} \right)$$

mas ele precisa $n!$ adições (A) e $n!n$ multiplicações (M), e nossos custos são

$$(n+1)(n!A + n!nM) \ge n!(A+M) \approx \sqrt{2\pi n}(n/e)^n(A+M)$$

um número formidável! Mas talvez a fórmula de Leibniz não é o melhor jeito de calcular o determinante! Vamos tentar a fórmula de expansão de Laplace

$$\det(A) = \sum_{1 \le i \le n} (-1)^{i+j} a_{ij} \det(A_{ij})$$

(sendo A_{ij} a matriz A sem linha a i e sem a coluna j). O trabalho T_n nesse caso é dado pelo recorrência

$$T_n = n(A + M + T_{n-1}); \qquad T_1 = 1$$

cuja solução é

$$T_n = n! \left(1 + (A+M) \sum_{1 \le i < n} 1/i! \right) \text{[1]}$$

[1] $n! \sum_{1 \le i < n} 1/i! = \lfloor n!(e-1) \rfloor$.

e como $\sum_{1 \leq i < n} 1/i!$ aproxima e temos $n! \leq T_n \leq n!(1 + (A + M)e)$ e logo T_n novamente é mais que $n!$. Mas qual é o método mais eficiente para calcular o determinante? Caso for possível em tempo proporcional ao tamanho da entrada n^2, tivermos um algoritmo em tempo aproximadamente n^3.
Antes de resolver essa pergunta, vamos estudar uma abordagem diferente da pergunta original, o método de Gauss para resolver um sistema de equações lineares. Em n passos, o matriz é transformada em forma triangular e cada passo não precisa mais que n^2 operações (nesse caso inclusive divisões).

Eliminação de Gauss

Entrada Uma matriz $A = (a_{ij}) \in \mathbb{R}^{n \times n}$

Saída A em forma triangular superior.

```
1  eliminação gauss(a ∈ ℝ^{n×n})=
2  for i := 1,...,n do { eliminate column i }
3     for j := i+1,...,n do { eliminate row j }
4        for k := n,...,i do
5           a_jk := a_jk − a_ik a_ji / a_ii
6        end for
7     end for
8  end for
```

Exemplo 1.3
Para resolver

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

vamos aplicar a eliminação de Gauss à matriz aumentada

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 4 & 5 & 7 & 4 \\ 7 & 8 & 9 & 6 \end{array}\right)$$

obtendo

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & -3 & -5 & -4 \\ 0 & -6 & -12 & -8 \end{array}\right); \qquad \left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & -3 & -5 & -4 \\ 0 & 0 & -2 & 0 \end{array}\right)$$

e logo $x_3 = 0$, $x_2 = 3/4$, $x_1 = 1/2$ é uma solução. $\Diamond$

Logo temos um algoritmo que determina a solução com

$$\sum_{1 \le i \le n} 3(n-i+1)(n-i) = n^3 - n$$

operações de ponto flutuante, que é (exceto valores de n bem pequenos) consideravelmente melhor que os resultados com $n!$ operações acima[2].
Observe que esse método também fornece o determinante do matriz: ela é o produto dos elementos na diagonal! De fato, o método é um dos melhores para achar o determinante. Observe também, que ela não serve para melhorar o método de Cramer, porque a solução do problema original já vem junto[3].

Qual o melhor algoritmo?

- Para um dado problema, existem diversos algoritmos com desempenhos diferentes.
- Queremos resolver um sistema de equações lineares de tamanho n.
- O método de Cramer precisa $\approx 6n!$ operações de ponto flutuante (OPF).
- O método de Gauss precisa $\approx n^3 - n$ OPF.
- Usando um computador de 3 GHz que é capaz de executar um OPF por ciclo temos

n	Cramer	Gauss
2	4 ns	2 ns
3	12 ns	8 ns
4	48 ns	20 ns
5	240ns	40 ns
10	7.3ms	330 ns
20	152 anos	2.7 ms

[2] O resultado pode ser melhorado considerando que a_{ji}/a_{ii} não depende do k

[3] O estudo da complexidade do determinante tem muito mais aspectos interessantes. Um deles é que o método de Gauss pode produzir resultados intermediários cuja representação precisa um número exponencial de bits em relação à entrada. Portanto, o método de Gauss formal mente não tem complexidade $O(n^3)$. Resultados atuais mostram que uma complexidade de operações de bits $n^{3.2} \log ||A||^{1+o(1)}$ é possível [15].

Motivação para algoritmos eficientes

- Com um algoritmo ineficiente, um computador rápido não ajuda!
- Suponha que uma máquina resolva um problema de tamanho N em um dado tempo.
- Qual tamanho de problema uma máquina 10 vezes mais rápida resolve no mesmo tempo?

Número de operações	Máquina rápida
$\log_2 n$	N^{10}
n	$10N$
$n \log_2 n$	$10N$ (N grande)
n^2	$\sqrt{10}N \approx 3.2N$
n^3	$\sqrt[3]{10}N \approx 2.2N$
2^n	$N + \log_2 10 \approx N + 3.3$
3^n	$N + \log_3 10 \approx N + 2.1$

Exemplo 1.4
Esse exemplo mostra como calcular os dados da tabela acima. Suponha um algoritmo que precisa $f(n)$ passos de execução numa dada máquina e uma outra máquina que é c vezes mais rápida. Portanto, ele é capaz de executar c vezes mais passos que a primeira. Ao mesmo tempo, qual seria o tamanho de problema n' que a nova máquina é capaz de resolver? Temos

$$f(n') = cf(n).$$

Por exemplo para $f(n) = \log_2 n$ e $c = 10$ (exemplo acima), temos

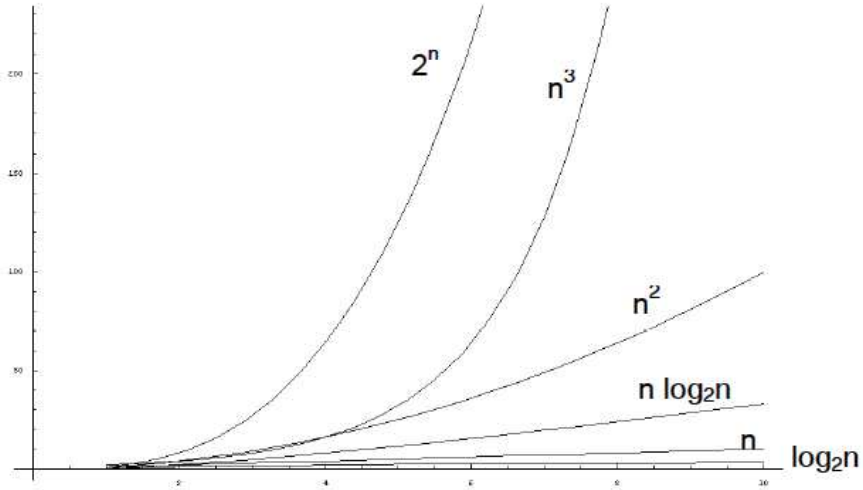
$$\log_2 n' = 10 \log_2 n \iff n' = n^{10}.$$

Em geral obtemos

$$n' = f^{-1}(cf(n))$$

(isso faz sentido para funções monotônicas, que têm inversa). $\Diamond$

Crescimento de funções



Crescimento de funções

$$1\,\text{ano} \approx 365.2425\text{d} \approx 3.2 \times 10^7\text{s}$$
$$1\,\text{século} \approx 3.2 \times 10^9\text{s}$$
$$1\,\text{milênio} \approx 3.2 \times 10^{10}\text{s}$$

Comparar eficiências

- Como comparar eficiências? Uma medida concreta do tempo depende
  - do tipo da máquina usada (arquitetura, cache, memória, ...)
  - da qualidade e das opções do compilador ou ambiente de execução
  - do tamanho do problema (da entrada)
- Portanto, foram inventadas *máquinas abstratas*.
- A *análise* da complexidade de um algoritmo consiste em determinar o número de operações básicas (atribuição, soma, comparação, ...) em relação ao tamanho da entrada.

Observe que nessa medida o tempo é "discreto".

Análise assintótica

- Em geral, o número de operações fornece um nível de detalhamento grande.
- Portanto, analisamos somente a taxa ou ordem de crescimento, substituindo funções exatas com cotas mais simples.
- Duas medidas são de interesse particular: A complexidade
  - pessimista e
  - média

Também podemos pensar em considerar a complexidade otimista (no caso melhor): mas essa medida faz pouco sentido, porque sempre é possível enganar com um algoritmo que é rápido para alguma entrada.

Exemplo

- Imagine um algoritmo com número de operações

$$an^2 + bn + c$$

- Para análise assintótica não interessam
  - os termos de baixa ordem, e
  - os coeficientes constantes.
- Logo o tempo da execução tem cota n^2, denotado com $O(n^2)$.

Observe que essas simplificações não devem ser esquecidas na escolha de um algoritmo na prática. Existem vários exemplos de algoritmos com desempenho bom assintoticamente, mas não são viáveis na prática em comparação com algoritmos "menos eficientes": A taxa de crescimento esconde fatores constantes e o tamanho mínimo de problema tal que um algoritmo é mais rápido que um outro.

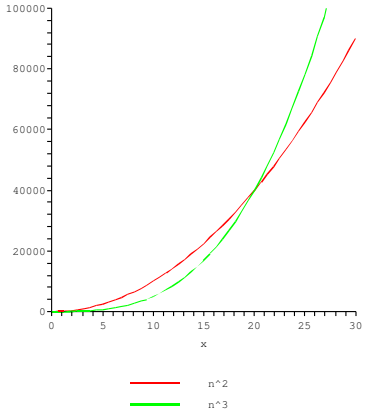
Complexidade de algoritmos

- Considere dois algoritmos A e B com tempo de execução O(n^2) e O(n^3), respectivamente. Qual deles é o mais eficiente ?
- Considere dois programas A e B com tempos de execução $100 \cdot n^2$ milisegundos,e $5 \cdot n^3$ milisegundos, respectivamente, qual é o mais eficiente?

Exemplo 1.5
Considerando dois algoritmos com tempo de execução $O(n^2)$ e $O(n^3)$ esperamos que o primeiro seja mais eficiente que o segundo. Para n grande, isso é verdadeiro, mas o tempo de execução atual pode ser $100n^2$ no primeiro e $5n^3$ no segundo caso. Logo para $n < 20$ o segundo algoritmo é mais rápido. ◊

Comparação de tempo de execução

- Assintoticamente consideramos um algoritmo com complexidade $O(n^2)$ melhor que um algoritmo com $O(n^3)$.
- De fato, para n suficientemente grande $O(n^2)$ sempre é melhor.
- Mas na prática, não podemos esquecer o tamanho do problema real.



Exemplo 1.6
Considere dois computadores C_1 e C_2 que executam 10^7 e 10^9 operações por segundo (OP/s) e dois algoritmos de ordenação A e B que necessitam $2n^2$ e $50n \log_{10} n$ operações com entrada de tamanho n, respectivamente. Qual o tempo de execução de cada combinação para ordenar 10^6 elementos?

Algoritmo	Comp. C_1	Comp. C_2
A	$\frac{2\times(10^6)^2 OP}{10^7 OP/s} = 2 \times 10^5 s$	$\frac{2\times(10^6)^2 OP}{10^9 OP/s} = 2 \times 10^3 s$
B	$\frac{50(10^6)\log 10^6 OP}{10^7 OP/s} = 30s$	$\frac{50(10^6)\log 10^6 OP}{10^9 OP/s} = 0.3s$

◊

Um panorama de tempo de execução

- Tempo constante: $O(1)$ (raro).
- Tempo sublinear ($\log(n)$, $\log(\log(n))$, etc): Rápido. Observe que o algoritmo não pode ler toda entrada.
- Tempo linear: Número de operações proporcional à entrada.
- Tempo $n \log n$: Comum em algoritmos de divisão e conquista.
- Tempo polinomial n^k: Freqüentemente de baixa ordem ($k \leq 10$), considerado eficiente.
- Tempo exponencial: 2^n, $n!$, n^n considerado intratável.

Exemplo 1.7
Exemplos de algoritmos para as complexidades acima:

- Tempo constante: Determinar se uma seqüência de números começa com 1.
- Tempo sublinear: Busca binária.
- Tempo linear: Buscar o máximo de uma seqüência.
- Tempo $n \log n$: Mergesort.
- Tempo polinomial: Multiplicação de matrizes.
- Tempo exponencial: Busca exaustiva de todos subconjuntos de um conjunto, de todas permutações de uma seqüência, etc.

$\Diamond$

Problemas super-polinomiais?

- Consideramos a classe P de problemas com solução em tempo polinomial tratável.
- NP é outra classe importante que contém muitos problemas práticos (e a classe P).
- Não se sabe se todos possuem algoritmo eficiente.

- Problemas NP-completos são os mais complexos do NP: Se um deles tem uma solução eficiente, toda classe tem.
- Vários problemas NP-completos são parecidos com problemas que têm algoritmos eficientes.

Solução eficiente conhecida	Solução eficiente improvável
Ciclo euleriano	Ciclo hamiltoniano
Caminho mais curto	Caminho mais longo
Satisfatibilidade 2-CNF	Satisfatibilidade 3-CNF

CICLO EULERIANO

Instância Um grafo não-direcionado $G = (V, E)$.

Decisão Existe um ciclo euleriano, i.e. um caminho $v_1, v_2, \ldots, v_n$ tal que $v_1 = v_n$ que usa todos arcos exatamente uma vez?

Comentário Tem uma decisão em tempo linear usando o teorema de Euler (veja por exemplo [8, Teorema 1.8.1]) que um grafo conexo contém um ciclo euleriano sse cada nó tem grau par. No caso de um grafo direcionado tem um teorema correspondente: um grafo fortemente conexo contém um ciclo euleriano sse cada nó tem o mesmo número de arcos entrantes e saintes.

CICLO HAMILTONIANO

Instância Um grafo não-direcionado $G = (V, E)$.

Decisão Existe um ciclo hamiltanio, i.e. um caminho $v_1, v_2, \ldots, v_n$ tal que $v_1 = v_n$ que usa todos nós exatamente uma única vez?

1.1. Notação assintótica

O análise de algoritmos considera principalmente recursos como tempo e espaço. Analisando o comportamento de um algoritmo em termos do tamanho da entrada significa achar uma função $c : \mathbb{N} \to \mathbb{R}^+$, que associa com todos entradas de um tamanho n um custo (médio,máximo) $c(n)$. Observe,

que é suficiente trabalhar com funções positivas (com co-domínio $\mathbb{R}^+$), porque os recursos de nosso interesse são positivos. A seguir, supomos que todas funções são dessa forma.

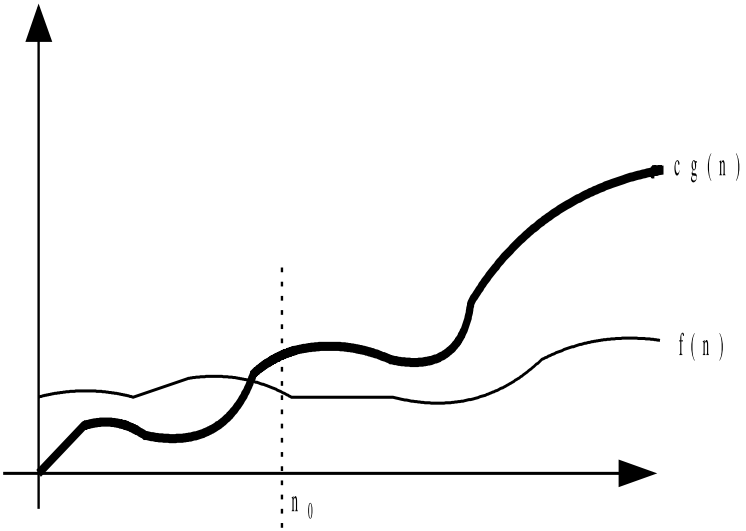
Notação assintótica: O

- Freqüentemente nosso interesse é o comportamento *assintótico* de uma função $f(n)$ para $n \to \infty$.
- Por isso, vamos introduzir *classes de crescimento*.
- O primeiro exemplo é *a classe de funções que crescem menos ou igual que* $g(n)$

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | (\exists c > 0) \exists n_0 (\forall n > n_0) : \ f(n) \leq cg(n)\}$$

A definição do O (e as outras definições em seguido) podem ser generalizadas para qualquer função com domínio $\mathbb{R}$.

Notação assintótica: O



Notação assintótica

- Com essas classes podemos escrever por exemplo

$$4n^2 + 2n + 5 \in O(n^2)$$

- Outra notação comum que usa a identidade é

$$4n^2 + 2n + 5 = O(n^2)$$

- Observe que essa notação é uma "equação única" (inglês: one-way equation);

$$O(n^2) = 4n^2 + 2n + 5$$

não é definido.

Para $f \in O(g)$ leia: "f é do ordem de g"; para $f = O(g)$ leiamos as vezes simplesmente "f é O de g". Observe numa equação como $4n^2 = O(n^2)$, as expressões $4n^2$ e n^2 denotam *funções*, não valores. Um jeito mais correto (mas menos confortável) seria escrever $\lambda n.4n^2 = O(\lambda n.n^2)$.
Caso $f \in O(g)$ com constante $c = 1$, digamos que g é uma *cota assintótica superior* de f [27, p. 15]. Em outras palavras, O define uma cota assintótica superior a menos de constantes.

O: Exemplos

$$5n^2 + n/2 \in O(n^3)$$
$$5n^2 + n/2 \in O(n^2)$$
$$\sin(n) \in O(1)$$

Exemplo 1.8
Mais exemplos

$$n^2 \in O(n^3 \log_2 n) \qquad c = 1; n_0 = 2$$
$$32n \in O(n^3) \qquad c = 32; n_0 = 1$$
$$10^n n^2 \notin O(n2^n) \qquad \text{porque } 10^n n^2 \leq cn2^n \iff 5^n n \leq c$$
$$n \log_2 n \in O(n \log_{10} n) \qquad c = 4; n_0 = 1$$

◊

O: Exemplos

Proposição 1.1
Para um polinômio $p(n) = \sum_{0 \leq i \leq m} a_i n^i$ temos

$$|p(n)| \in O(n^m) \tag{1.1}$$

Prova.

$$\begin{aligned} |p(n)| &= \left| \sum_{0 \leq i \leq m} a_i n^i \right| \\ &\leq \sum_{0 \leq i \leq m} |a_i| n^i \qquad \text{Corolário A.1} \\ &\leq \sum_{0 \leq i \leq m} |a_i| n^m = n^m \sum_{0 \leq i \leq m} |a_i| \end{aligned}$$

■

Notação assintótica: Outras classes

- Funções que crescem (estritamente) menos que $g(n)$

$$o(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | (\forall c > 0) \exists n_0 (\forall n > n_0) : \ f(n) \leq cg(n)\} \tag{1.2}$$

- Funções que crescem mais ou igual à $g(n)$

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | (\exists c > 0) \exists n_0 (\forall n > n_0) : \ f(n) \geq cg(n)\} \tag{1.3}$$

- Funções que crescem (estritamente) mais que $g(n)$

$$\omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | (\forall c > 0) \exists n_0 (\forall n > n_0) : \ f(n) \geq cg(n)\} \tag{1.4}$$

- Funções que crescem igual à $g(n)$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \tag{1.5}$$

Observe que a nossa notação somente é definida "ao redor do ∞", que é suficiente para a análise de algoritmos. Equações como $e^x = 1 + x + O(x^2)$, usadas no cálculo, possuem uma definição de O diferente.
As definições ficam equivalente, substituindo $<$ para $\leq$ e $>$ para $\geq$ (veja exercício 1.5).

Convenção 1.1
Se o contexto permite, escrevemos $f \in O(g)$ ao invés de $f(n) \in O(g(n))$, $f \leq cg$ ao invés de $f(n) \leq cg(n)$ etc.

Proposição 1.2 (Caracterização alternativa)
Caracterizações alternativas de O, o, Ω e ω são

$$f(n) \in O(g(n)) \iff \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{1.6}$$

$$f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{1.7}$$

$$f(n) \in \Omega(g(n)) \iff \liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{1.8}$$

$$f(n) \in \omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{1.9}$$

Prova. Prova de 1.6:
"⇒": Seja $f \in O(g)$. Como $s(n) = \sup_{m\geq n} f(m)/g(m)$ é não-crescente e maior ou igual que 0, é suficiente mostrar que existe um n tal que $s(n) < \infty$. Por definição do O temos $c > 0$ e n_0 tal que $\forall n > n_0\ f \leq cg$. Logo $\forall n > n_0\ \sup_{m\geq n} f(m)/g(m) \leq c$.
"⇐": Seja $\limsup_{n\to\infty} f(n)/g(n) < \infty$. Então

$$\exists c > 0 \exists n_0 \forall n > n_0 (\sup_{m\geq n} f(m)/g(m)) < c.$$

Isso implica, que para o mesmo n_0, $\forall n > n_0\ f < cg$ e logo $f \in O(g)$.
Prova de 1.7:
"⇒": Seja $f \in o(g)$, i.e. para todo $c > 0$ temos um n_0 tal que $\forall n > n_0\ f \leq cg$. Logo $\forall n > n_0\ f(n)/g(n) \leq c$, que justifique $\lim_{n\to\infty} f(n)/g(n) = 0$ (veja lema A.1).
"⇐": Seja $\lim_{n\to\infty} f(n)/g(n) = 0$, i.e. para todo $c > 0$ existe um n_0 tal que $\forall n > n_0\ f(n)/g(n) < c$ pela definição do limite. Logo $\forall n > n_0\ f \leq cg$, tal que $f \in o(g)$.
Prova de 1.8:
"⇒": Seja $f \in \Omega(g)$. Como $i(n) = \inf_{m\geq n} f(m)/g(m)$ é não-decrescente, é suficiente mostrar, que existe um n tal que $i(n) > 0$. Pela definição de Ω existem $c > 0$ e n_0 tal que $\forall n > n_0\ f \geq cg$. Logo $\forall n > n_0\ f(n)/g(n) \geq c > 0$, i.e. $i(n_0 + 1) > 0$.
"⇐": Suponha $\liminf_{n\to\infty} f(n)/g(n) = l > 0$. Vamos considerar os casos $l < \infty$ e $l = \infty$ separadamente.

Caso $l < \infty$: Escolhe, por exemplo, $c = l/2$. Pela definição do limite existe n_0 tal que $\forall n > n_0 \, |l - f/g| \leq l/2$. Logo $f \geq l/2g$ (f/g aproxima l por baixo) e $f \in \Omega(g)$.
Caso $l = \infty$, $i(n)$ não tem limite superior, i.e. $(\forall c > 0)\exists n_0 \, i(n_0) > c$. Como $i(n)$ é não-decrescente isso implica $(\forall c > 0)\exists n_0(\forall n > n_0) \, i(n) > c$. Portanto $\forall n > n_0 f > cg$ e $f \in \omega(g) \subseteq \Omega(g)$.
Prova de 1.9:

$$\begin{aligned} & f \in \omega(g) \\ \iff & (\forall c > 0)\exists n_0(\forall n > n_0) \ : f \geq cg \\ \iff & (\forall c > 0)\exists n_0(\forall n > n_0) \ : f(n)/g(n) \geq c \\ \iff & f(n)/g(n) \text{ não tem limite} \end{aligned}$$

■

Convenção 1.2
Escrevemos $f, g, \ldots$ para funções $f(n), g(n), \ldots$ caso não tem ambigüedade no contexto.

Operações

- As notações assintóticas denotam conjuntos de funções.
- Se um conjunto ocorre em uma fórmula, resulta o conjunto de todas combinações das partes.
- Exemplos: $n^{O(1)}$, $\log O(n^2)$, $n^{1+o(1)}$, $(1 - o(1)) \ln n$
- Em uma equação o lado esquerdo é (implicitamente) quantificado universal, e o lado direto existencial.
- Exemplo: $n^2 + O(n) = O(n^4)$ Para todo $f \in O(n)$, existe um $g \in O(n^4)$ tal que $n^2 + f = g$.

Exemplo 1.9
$n^{O(1)}$ denota

$$\{n^{f(n)} \mid \exists c \, f(n) \leq c\} \subseteq \{f(n) \mid \exists c \exists n_0 \, \forall n > n_0 \, f(n) \leq n^c\}$$

o conjunto das funções que crescem menos que um polinômio. ◊

Uma notação assintótica menos comum é $\tilde{O}$ que é uma abreviação para $f = O(g \log_k g)$ para algum k. $\tilde{O}$ é usado se fatores logarítmicos não importam.

Características

$$f = O(f) \tag{1.10}$$
$$cO(f) = O(f) \tag{1.11}$$
$$O(f) + O(f) = O(f) \tag{1.12}$$
$$O(O(f)) = O(f) \tag{1.13}$$
$$O(f)O(g) = O(fg) \tag{1.14}$$
$$O(fg) = fO(g) \tag{1.15}$$

Prova.
Prova de 1.10: Escolhe $c = 1$, $n_0 = 0$.
Prova de 1.11: Se $g \in cO(f)$, temos $g = cg'$ e existem $c' > 0$ e n_0 tal que $\forall n > n_0\ g' \leq c'f$. Portanto $\forall n > n_0\ g = cg' \leq cc'f$ e com cc' e n_0 temos $g \in O(f)$.
Prova de 1.12: Para $g \in O(f) + O(f)$ temos $g = h + h'$ com $c > 0$ e n_0 tal que $\forall n > n_0\ h \leq cf$ e $c' > 0$ e n_0' tal que $\forall n > n_0\ h' \leq c'f$. Logo para $n > \max(n_0, n_0')$ temos $g = h + h' \leq (c + c')f$.
Prova de 1.13: Para $g \in O(O(f))$ temos $g \leq ch$ com $h \leq c'f$ a partir de índices n_0 e n_0', e logo $g \leq cc'h$ a partir de $\max(n_0, n_0')$.
Prova de 1.14: $h = f'g'$ com $f' \leq c_f f$ e $g' \leq c_g g$ tal que $h = f'g' \leq c_f c_g fg$.
Prova de 1.15: Para $h \in O(fg)$ temos $c > 0$ e n_0 tal que $\forall n > n_0\ h \leq cfg$. Temos que mostrar, que h pode ser escrito como $h = fg'$ com $g' \in O(g)$. Seja

$$g'(n) = \begin{cases} h(n)/f(n) & \text{se } f(n) \neq 0 \\ cg(n) & \text{caso contrário} \end{cases}$$

Verifique-se que $h = fg'$ por análise de casos. Com isso, temos também $g' = h/f \leq cfg/f = cg$ nos casos $f(n) \neq 0$ e $g' = cg \leq cg$ caso contrário. ■
As mesmas características são verdadeiras para Ω (prova? veja exercício 1.1). E para o, ω e Θ?

Características

$$f = O(h) \wedge g = O(h) \Rightarrow f + g = O(h)$$
$$g = O(f) \Rightarrow f + g = \Theta(f)$$

A segunda característica se chama "princípio de absorção" [27, p. 35].

Relações de crescimento A motivação pela notação O e o seu uso no meio de fórmulas. Para os casos em que isso não for necessário, podemos introduzir relações de crescimento entre funções com uma notação mais comum. Uma definição natural é

Relações de crescimento

Definição 1.1 (Relações de crescimento)

$$f \prec g \iff f \in o(g) \tag{1.16}$$
$$f \preceq g \iff f \in O(g) \tag{1.17}$$
$$f \succ g \iff f \in \omega(g) \tag{1.18}$$
$$f \succeq g \iff f \in \Omega(g) \tag{1.19}$$
$$f \asymp g \iff f \in \Theta(g) \tag{1.20}$$

Essas relações também são conhecidas como a notação de Vinogradov. Uma variante comum dessa notação usa $\ll$ para $\preceq$ e $\gg$ para $\succeq$ (infelizmente a notação nessa área não é muito padronizada.)
Caso $f \preceq g$ digamos as vezes "f é absorvido pela g". Essas relações satisfazem as características básicas esperadas.

Características das relações de crescimento

Proposição 1.3 (Características das relações de crescimento)
Sobre o conjunto de funções $[\mathbb{N} \to \mathbb{R}^+]$

1. $f \preceq g \iff g \succeq f$,
2. $\preceq$ e $\succeq$ são ordenações parciais (reflexivas, transitivas e anti-simétricas em relação de $\asymp$),
3. $f \prec g \iff g \succ f$,
4. $\prec$ e $\succ$ são transitivas,
5. $\asymp$ é uma relação de equivalência.

Prova.

1. Temos as equivalências

$$\begin{aligned} f \preceq g &\iff f \in O(g) \\ &\iff \exists c\, \exists n_0\, \forall n > n_0\ f \leq cg \\ &\iff \exists c'\, \exists n_0\, \forall n > n_0\ g \geq c' f \qquad \text{com } c' = 1/c \\ &\iff g \in \Omega(f) \end{aligned}$$

2. A reflexividade e transitividade são fáceis de verificar. No exemplo do $\preceq$, $f \preceq f$, porque $\forall n f(n) \leq f(n)$ e $f \preceq g$, $g \preceq h$ garante que a partir de um n_0 temos $f \leq cg$ e $g \leq c'h$ e logo $f \leq (cc')h$ também. Caso $f \preceq g$ e $g \preceq f$ temos com item (a) $f \succeq g$ e logo $f \asymp g$ pela definição de Θ.

3. Temos as equivalências

$$\begin{aligned} f \prec g &\iff f \in o(g) \\ &\iff \forall c\, \exists n_0\, \forall n > n_0\ f \leq cg \\ &\iff \forall c'\, \exists n_0\, \forall n > n_0\ g \geq c' f \qquad \text{com } c' = 1/c \\ &\iff g \in \omega(f) \end{aligned}$$

4. O argumento é essencialmente o mesmo que no item (a).

5. Como Θ é definido pela intersecção de O e Ω, a sua reflexividade e transitividade é uma conseqüência da reflexividade e transitividade do O e Ω. A simetria é uma conseqüência direta do item (a).

■

Observe que esses resultados podem ser traduzidos para a notação O. Por exemplo, como $\asymp$ é uma relação de equivalência, sabemos que Θ também satisfaz

$$\begin{gathered} f \in \Theta(f) \\ f \in \Theta(g) \Rightarrow g \in \Theta(f) \\ f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h) \end{gathered}$$

A notação com relações é sugestiva e freqüentemente mais fácil de usar, mas nem todas as identidades que ela sugere são válidas, como a seguinte proposição mostra.

Identidades falsas das relações de crescimento

Proposição 1.4 (Identidades falsas das relações de crescimento)
É verdadeiro que

$$f \succ g \Rightarrow f \not\preceq g \tag{1.21}$$
$$f \prec g \Rightarrow f \not\succeq g \tag{1.22}$$

mas as seguintes afirmações *não* são verdadeiras:

$$f \not\preceq g \Rightarrow f \succ g$$
$$f \not\succeq g \Rightarrow f \prec g$$
$$f \prec g \vee f \asymp g \vee f \succ g \qquad \text{(Tricotomia)}$$

Prova. Prova de 1.21 e 1.22: Suponha $f \succ g$ e $f \preceq g$. Então existe um c tal que a partir de um n_0 temos que $f = cg$ (usa as definições). Mas então $f \not\prec g$ é uma contradição. A segunda característica pode ser provada com um argumento semelhante.
Para provar as três afirmações restantes considere o par de funções n e $e^{n \sin(n)}$. Verifique-se que nenhuma relação $\prec$, $\preceq$, $\succ$, $\succeq$ ou $\asymp$ é verdadeira. ■
Considerando essas características, a notação tem que ser usada com cautela. Uma outra abordagem é definir O etc. diferente, tal que outras relações acima são verdadeiras. Mas parece que isso não é possível, sem perder outras, veja [29].

Outras notações comuns

Definição 1.2
O *logaritmo iterado* é

$$\log^* n = \begin{cases} 0 & \text{se } n \leq 1 \\ 1 + \log^*(\log n) & \text{caso contrário} \end{cases}$$

O logaritmo iterado é uma função que cresce extremamente lento; para valores práticos de n, $\log^* n$ não ultrapassa 5.

1.2. Notas

Alan Turing provou em 1936 que o "problema de parada" não é decidível. O estudo da complexidade de algoritmos começou com o artigo seminal de Hartmanis e Stearns [12].

1.3. Exercícios

(Soluções a partir da página 221.)

Exercício 1.1
Prove as características 1.10 até 1.15 (ou características equivalentes caso alguma não se aplica) para Ω.

Exercício 1.2
Prove ou mostre um contra-exemplo. Para qualquer constante $c \in \mathbb{R}$, $c > 0$

$$f \in O(g) \iff f + c \in O(g) \tag{1.23}$$

Exercício 1.3
Prove ou mostre um contra-exemplo.

1. $\log(1 + n) = O(\log n)$
2. $\log O(n^2) = O(\log n)$
3. $\log \log n = O(\log n)$

Exercício 1.4
Considere a função definido pela recorrência

$$f_n = 2f_{n-1}; \qquad f_0 = 1.$$

Professor Veloz afirme que $f_n = O(n)$, e que isso pode ser verificado simplesmente da forma

$$f_n = 2f_{n-1} = 2O(n-1) = 2O(n) = O(n)$$

Mas sabendo que a solução dessa recorrência é $f_n = 2^n$ temos dúvidas que $2^n = O(n)$. Qual o erro do professor Veloz?

Exercício 1.5
Mostre que a definição

$$\hat{o}(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | (\forall c > 0) \exists n_0 (\forall n > n_0) : \ f(n) < cg(n)\}$$

(denotado com $\hat{o}$ para diferenciar da definição o) e equivalente com a definição 1.2 para funções $g(n)$ que são diferente de 0 a partir de um n_0.

Exercício 1.6
Mostre que o números Fibonacci

$$f_n = \begin{cases} n & \text{se } 0 \leq n \leq 1 \\ f_{n-2} + f_{n-1} & \text{se } n \geq 2 \end{cases}$$

têm ordem assintótica $f_n \in \Theta(\Phi^n)$ com $\Phi = (1 + \sqrt{5})/2$.

Exercício 1.7
Prove a seguinte variação do princípio de absorção:

$$g \in o(f) \Rightarrow f - g \in \Theta(f).$$

Exercício 1.8
Prove que

$$f \leq g \Rightarrow O(f) = O(g).$$

Exercício 1.9
Prove que $\Theta(f) = O(f)$, mas o contrário $O(f) = \Theta(f)$ não é correto.

2. Análise de complexidade

2.1. Introdução

Para analisar a eficiência de algoritmos faz pouco sentido medir os recursos gastos em computadores específicos, porque devido a diferentes conjuntos de instruções, arquiteturas e desempenho dos processadores, as medidas são difíceis de comparar. Portanto, usamos um *modelo* de uma máquina que reflita as características de computadores comuns, mas é independente de uma implementação concreta. Um modelo comum é a *máquina de RAM* com as seguintes características:

- um processador com um ou mais registros, e com apontador de instruções,
- um memória e
- um conjunto de instruções fundamentais que podem ser executadas em tempo $O(1)$ (por exemplo funções básicas sobre números inteiros e de ponto flutuante, acesso à memória e transferência de dados); essas operações refletem operações típicas de máquinas concretas.

Observe que a escolha de um modelo abstrato não é totalmente trivial. Conhecemos vários modelos de computadores, cuja poder computacional não é equivalente em termos de complexidade (que não viola a tese de Church-Turing). Mas todos os modelos encontrados (fora da computação quântica) são polinomialmente equivalentes, e portanto, a noção de eficiência fica a mesma. A tese que todos modelos computacionais são polinomialmente equivalentes as vezes está chamado *tese de Church-Turing estendida*.

O plano

Uma hierarquia de abstrações:

$$\text{Tempo de execução [s]} \longrightarrow \text{Número de operações} \longrightarrow \text{Operações no caso pessimista} \longrightarrow \text{Complexidade pessimista assintótica}$$

Custos de execuções

- Seja E o conjunto de seqüências de operações fundamentais.

- Para um algoritmo a, com entradas D seja

$$\text{exec}[a] : D \to E$$

  a função que fornece a seqüência de instruções executadas $\text{exec}[a](d)$ para cada entrada $d \in D$.

- Se atribuímos custos para cada operação básica, podemos calcular também o custo de uma execução

$$\text{custo} : E \to \mathbb{R}^+$$

- e o custo da execução do algoritmo a depende da entrada d

$$\text{desemp}[a] : D \to \mathbb{R}^+ = \text{custo} \circ \text{exec}[a]$$

Definição 2.1
O símbolo $\circ$ denota a composição de funções tal que

$$(f \circ g)(n) = f(g(n))$$

(leia: "f depois g").

Em geral, não interessam os custos específicos para cada entrada, mas o "comportamento" do algoritmo. Uma medida natural é como os custos crescem com o tamanho da entrada.

Condensação de custos

- Queremos condensar os custos para uma única medida.
- Essa medida depende somente do tamanho da entrada

$$\text{tam} : D \to \mathbb{N}$$

- O objetivo é definir uma função

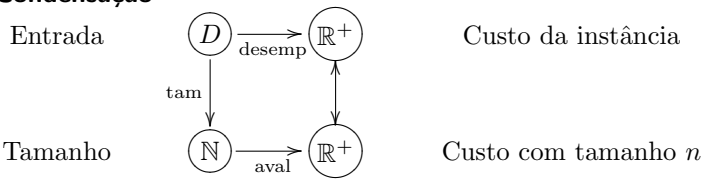
$$\text{aval}[a](n) : \mathbb{N} \to \mathbb{R}^+$$

  que define o desempenho do algoritmo em relação ao tamanho.

- Como, em geral, tem várias entradas d tal que $\text{tam}(d) = n$ temos que definir como condensar a informação de $\text{desemp}[a](d)$ delas.

Observe que as vezes $\mathbb{R}^+$ é denotado $\mathbb{R}_+$.

Condensação



Condensação

- Na prática, duas medidas condensadas são de interesse particular
- A complexidade pessimista

$$C_p^=[a](n) = \max\{\text{desemp}[a](d) \mid d \in D, \text{tam}(d) = n\}$$

- A complexidade média

$$C_m[a](n) = \sum_{\text{tam}(d)=n} P(d)\,\text{desemp}[a](d)$$

- Observe: A complexidade média é o valor esperado do desempenho de entradas com tamanho n.
- Ela é baseada na distribuição das entradas.

A complexidade média é menos usada na prática, por várias razões. Primeiramente, uma complexidade pessimista significa um desempenho garantido, independente da entrada. Em comparação, uma complexidade média de, por exemplo, $O(n^2)$, não exclui que em certos casos uma entrada com tamanho n precisa muito mais tempo. Por isso, se é importante saber quando uma execução de um algoritmo termina, preferimos a complexidade pessimista. Mesmo assim, para vários algoritmos com desempenho ruim no pior caso, estamos interessados como eles se comportam na média (complexidade média). Infelizmente, ela é difícil de determinar. Além disso, ela depende da distribuição das entradas, que freqüentemente não é conhecida ou difícil de determinar, ou é diferente em aplicações diferentes.

Definição alternativa

- A complexidade pessimista é definida como

$$C_p^=[a](n) = \max\{\text{desemp}[a](d) \mid d \in D, \text{tam}(d) = n\}$$

- Uma definição alternativa é

$$C_p^{\leq}[a](n) = \max\{\text{desemp}[a](d) \mid d \in D, \text{tam}(d) \leq n\}$$

- $C_p^{\leq}$ é monotônica e temos

$$C_p^{=}[a](n) \leq C_p^{\leq}[a](n)$$

- Caso $C_p^{=}$ seja monotônica as definições são equivalentes

$$C_p^{=}[a](n) = C_p^{\leq}[a](n)$$

$C_p^{=}[a](n) \leq C_p^{\leq}[a](n)$ é uma conseqüência da observação que

$$\{\text{desemp}[a](d) \mid d \in D, \text{tam}(d) = n\} \subseteq \{\text{desemp}[a](d) \mid d \in D, \text{tam}(d) \leq n\}$$

Analogamente, se $A \subseteq B$ tem-se que $\max A \leq \max B$.

Exemplo 2.1
Vamos aplicar essas noções num exemplo de um algoritmo simples. O objetivo é decidir se uma seqüência de números naturais contém o número 1.

BUSCA1

Entrada Uma seqüência $a_1, \ldots, a_n$ de números em $\mathbb{N}$.

Saída True, caso existe um i tal que $a_i = 1$, false, caso contrário.

```
1  for i:=1 to n do
2    if (a_i = 1) then
3      return true
4    end if
5  end for
6  return false
```

Para analisar o algoritmo, podemos escolher, por exemplo, as operações básicas $O = \{for, if, return\}$ e atribuir um custo constante de 1 para cada um delas. (Observe que como "operação básica" for são consideradas as operações de atribuição, incremento e teste da expressão booleana $i \leq n$.) Logo as execuções possíveis são $E = O^*$ (o fecho de Kleene de O) e temos a função de custos

$$\text{custo} : E \to \mathbb{R}^+ : e \mapsto |e|.$$

Por exemplo custo$((for, for, if, return)) = 4$. As entradas desse algoritmo são seqüências de números naturais, logo, $D = \mathbb{N}^*$ e como tamanho da entrada escolhemos

$$\text{tam} : D \to \mathbb{N} : (a_1, \ldots, a_n) \mapsto n.$$

A função de execução atribui a seqüência de operações executadas a qualquer entrada. Temos

$$\text{exec[Busca1]}(d) : D \to E :$$
$$(a_1, \ldots, a_n) \mapsto \begin{cases} (for, if)^i return & \text{caso existe } i = \min\{j \mid a_j = 1\} \\ (for, if)^n return & \text{caso contrário} \end{cases}$$

Com essas definições temos também a função de desempenho

$$\text{desemp[Busca1]}(n) = \text{custo} \circ \text{exec[Busca1]} :$$
$$(a_1, \ldots, a_n) \mapsto \begin{cases} 2i + 1 & \text{caso existe } i = \min\{j \mid a_j = 1\} \\ 2n + 1 & \text{caso contrário} \end{cases}$$

Agora podemos aplicar a definição da complexidade pessimista para obter

$$C_p^{\leq}[\text{Busca1}](n) = \max\{\text{desemp[Busca1]}(d) \mid \text{tam}(d) = n\} = 2n + 1 = O(n).$$

Observe que $C_p^{=}$ é monotônica, e portanto $C_p^{=} = C_p^{\leq}$.
Um caso que em geral é menos interessante podemos tratar nesse exemplo também: Qual é a complexidade otimista (complexidade no caso melhor)? Isso acontece quando 1 é o primeiro elemento da seqüência, logo, $C_o[\text{Busca1}](n) = 2 = O(1)$.

$\Diamond$

2.2. Complexidade pessimista

2.2.1. Metodologia de análise de complexidade

Uma linguagem simples

- Queremos estudar como determinar a complexidade de algoritmos metodicamente.
- Para este fim, vamos usar uma linguagem simples que tem as operações básicas de

1. Atribuição: v:=e
2. Seqüência: c1;c2
3. Condicional: se b então c1 senão c2
4. Iteração definida: para i de j até m faça c
5. Iteração indefinida: enquanto b faça c

A forma se b então c1 vamos tratar como abreviação de se b então c1 **else skip** com comando **skip** de custo 0.
Observe que a metodologia não implica que tem *um algoritmo* que, dado um algoritmo como entrada, computa a complexidade dele. Este problema não é computável (por quê?).

Convenção 2.1
A seguir vamos entender implicitamente todas operações sobre funções *pontualmente*, i.e. para alguma operação $\circ$, e funções f, g com $\mathrm{dom}(f) = \mathrm{dom}(g)$ temos

$$\forall d \in \mathrm{dom}(f) \quad (f \circ g) = f(d) \circ g(d).$$

Componentes

- A complexidade de um algoritmo pode ser analisada em termos de suas componentes (princípio de composicionalidade).
- Pode-se diferenciar dois tipos de componentes: *componentes conjuntivas* e *componentes disjuntivas*.
- Objetivo: Analisar as componentes independentemente (como sub-algoritmos) a depois compor as complexidades delas.

Composição de componentes

- Cautela: Na composição de componentes o tamanho da entrada pode mudar.
- Exemplo: Suponha que um algoritmo A produz, a partir de uma lista de tamanho n, uma lista de tamanho n^2 em tempo $\Theta(n^2)$. $n -$
- A seqüência $A; A$, mesmo sendo composta pelos dois algoritmos com $\Theta(n^2)$ *individualmente*, tem complexidade $\Theta(n^4)$.
- Portanto, vamos diferençar entre

  - algoritmos que preservam (assintoticamente) o tamanho, e
  - algoritmos em que modificam o tamanho do problema.

- Neste curso tratamos somente o primeiro caso.

Componentes conjuntivas

A seqüência

- Considere uma seqüência $c_1; c_2$.
- Qual a sua complexidade $c_p[c_1; c_2]$ em termos dos componentes $c_p[c_1]$ e $c_p[c_2]$?
- Temos

  $$\text{desemp}[c_1; c_2] = \text{desemp}[c_1] + \text{desemp}[c_2] \geq \max(\text{desemp}[c_1], \text{desemp}[c_2])$$

  e portanto (veja A.8)

  $$\max(c_p[c_1], c_p[c_2]) \leq c_p[c_1; c_2] \leq c_p[c_1] + c_p[c_2]$$

  e como $f + g \in O(\max(f, g))$ tem-se que

  $$c_p[c_1; c_2] = \Theta(c_p[c_1] + c_p[c_2]) = \Theta(\max(c_p[c_1], c_p[c_2]))$$

Prova.

$$\begin{aligned} \max(\text{desemp}[c_1](d), \text{desemp}[c_2](d)) &\leq \text{desemp}[c_1; c_2](d) \\ &= \text{desemp}[c_1](d) + \text{desemp}[c_2](d) \end{aligned}$$

logo para todas entradas d com $\text{tam}(d) = n$

$$\begin{aligned} \max_d \max(\text{desemp}[c_1](d), \text{desemp}[c_2](d)) &\leq \max_d \text{desemp}[c_1; c_2](d) \\ &= \max_d (\text{desemp}[c_1](d) + \text{desemp}[c_2](d)) \\ \iff \max(c_p[c_1], c_p[c_2]) &\leq c_p[c_1; c_2] \leq c_p[c_1] + c_p[c_2] \end{aligned}$$

■

Exemplo 2.2
Considere a seqüência $S \equiv v := \mathit{ordena}(u); w := \mathit{soma}(u)$ com complexidades $c_p[v := \mathit{ordena}(u)](n) = n^2$ e $c_p[w := \mathit{soma}(u)](n) = n$. Então $c_p[S] = \Theta(n^2 + n) = \Theta(n^2)$. ◊

Exemplo 2.3
Considere uma partição das entradas do tamanho n tal que $\{d \in D | tam(d) = n\} = D_1(n) \dot{\cup} D_2(n)$ e dois algoritmos A_1 e A_2, A_1 precisa n passos para instâncias em D_1, e n^2 para instâncias em D_2. A_2, contrariamente, precisa n^2 para instâncias em D_1, e n passos para instâncias em D_2. Com isso obtemos

$$c_p[A_1] = n^2; \qquad c_p[A_2] = n^2; \qquad c_p[A_1; A_2] = n^2 + n$$

e portanto

$$\max(c_p[A_1], c_p[A_2]) = n^2 < c_p[A_1; A_2] = n^2 + n < c_p[A_1] + c_p[A_2] = 2n^2$$

$\Diamond$

A atribuição: Exemplos

- Considere os seguintes exemplos.
- Inicialização ou transferência da variáveis inteiras tem complexidade $O(1)$
$$i := 0; \qquad j := i$$
- Determinar o máximo de uma lista de elementos v tem complexidade $O(n)$
$$m := \max(v)$$
- Inversão de uma lista u e atribuição para w tem complexidade $2n \in O(n)$
$$w := reversa(u)$$

A atribuição

- Logo, a atribuição depende dos custos para a avaliação do lado direito e da atribuição, i.e.
$$\text{desemp}[v := e] = \text{desemp}[e] + \text{desemp}[\leftarrow_e]$$
- Ela se comporta como uma seqüência dessas duas componentes, portanto
$$c_p[v := e] = \Theta(c_p[e] + c_p[\leftarrow_e])$$
- Freqüentemente $c_p[\leftarrow_e]$ é absorvida pelo $c_p[e]$ e temos
$$c_p[v := e] = \Theta(c_p[e])$$

Exemplo 2.4
Continuando o exemplo 2.2 podemos examinar a atribuição $v := ordene(w)$. Com complexidade pessimista para a ordenação da lista $c_p[ordene(w)] = O(n^2)$ e complexidade $c_p[\leftarrow_e] = O(n)$ para a transferência, temos $c_p[v := ordene(w)] = O(n^2) + O(n) = O(n^2)$. ◊

Iteração definida
Seja C =para i de j até m faça c

- O número de iterações é fixo, mas j e m dependem da entrada d.
- Seja $N(n) = \max\{m(d) - j(d) + 1 | tam(d) \leq n\}$ e $N^*(n) = \{N(n), 0\}$.
- $N^*(n)$ é o máximo de iterações para entradas de tamanho até n.
- Tendo N^*, podemos tratar a iteração definida como uma seqüência

$$\underbrace{c; c; \cdots ; c}_{N^*(n)\text{vezes}}$$

que resulta em

$$c_p[C] \leq N^*\, c_p[c]$$

Iteração indefinida
Seja C =enquanto b faça c

- Para determinar a complexidade temos que saber o número de iterações.
- Seja $H(d)$ o número da iteração (a partir de 1) em que a condição é falsa pela primeira vez
- e $h(n) = \max\{H(d) | tam(d) \leq n\}$ o número máximo de iterações com entradas até tamanho n.
- Em analogia com a iteração definida temos uma seqüência

$$\underbrace{b; c; b; c; \cdots ; b; c}_{h(n)-1\text{vezes}}; b$$

e portanto

$$c_p[C] \leq (h-1)c_p[c] + hc_p[b]$$

- Caso o teste b é absorvido pelo escopo c temos

$$c_p[C] \leq (h-1)c_p[c]$$

Observe que pode ser difícil de determinar o número de iterações $H(d)$; em geral a questão não é decidível.

Componentes disjuntivas

Componentes disjuntivas

- Suponha um algoritmo c que consiste em duas componentes disjuntivas c_1 e c_2. Logo,

$$\text{desemp}[c] \leq \max(\text{desemp}[c_1], \text{desemp}[c_2])$$

e temos

$$c_p[a] \leq \max(c_p[c_1], c_p[c_2])$$

Caso a expressão para o máximo de duas funções for difícil, podemos simplificar

$$c_p[a] \leq \max(c_p[c_1], c_p[c_2]) = O(\max(c_p[c_1], c_p[c_2])).$$

O condicional

Seja $C =$se b então c1 senão c2

- O condicional consiste em
  - uma avaliação da condição b
  - uma avaliação do comando c_1 ou c_2 (componentes disjuntivas).
- Aplicando as regras para componentes conjuntivas e disjuntivas obtemos

$$c_p[C] \leq c_p[b] + \max(c_p[c_1], c_p[c_2])$$

Para se b então c1 obtemos com $c_p[skip] = 0$

$$c_p[C] \leq c_p[b] + c_p[c_1]$$

Exemplo 2.5
Considere um algoritmo a que, dependendo do primeiro elemento de uma lista u, ordena a lista ou determina seu somatório:

Exemplo

```
1  se hd(u) = 0 então
2     v := ordena(u)
3  senão
4     s := soma(u)
```

Assumindo que o teste é possível em tempo constante, ele é absorvido pelo trabalho em ambos casos, tal que

$$c_p[a] \leq \max(c_p[v := \text{ordena}(u)], c_p[s := soma(u)])$$

e com, por exemplo, $c_p[v := \text{ordena}(u)](n) = n^2$ e $c_p[s := \text{soma}(u)](n) = n$ temos

$$c_p[a](n) \leq n^2$$

$\Diamond$

2.2.2. Exemplos

Exemplo 2.6 (Bubblesort)
Nesse exemplo vamos estudar o algoritmo Bubblesort de ordenação.

Bubblesort

BUBBLESORT

Entrada Uma seqüência $a_1, \ldots, a_n$ de números inteiros.

Saída Uma seqüência $a_{\pi(1)}, \ldots, a_{\pi(n)}$ de números inteiros tal que π é uma permutação de $[1, n]$ e para $i < j$ temos $a_{\pi(i)} \leq a_{\pi(j)}$.

```
1  for i:=1 to n
2     for j:=1 to n i
3        if a_j > a_{j+1} then
4           swap a_j ,a_{j+1}
5        end if
6     end for
7  end for
```

Bubblesort: Complexidade

- A medida comum para algoritmos de ordenação: o número de comparações (de chaves).
- Qual a complexidade pessimista?

$$\sum_{1 \le i \le n} \sum_{1 \le j \le n-i} 1 = n/2(n-1).$$

- Qual a diferença se contamos as transposições também?

$\Diamond$

Exemplo 2.7 (Ordenação por inserção direta)

Ordenação por inserção direta

Ordenação por inserção direta (inglês: straight insertion sort)

Entrada Uma seqüência $a_1, \ldots, a_n$ de números inteiros.

Saída Uma seqüência $a_{\pi(1)}, \ldots, a_{\pi(n)}$ de números inteiros tal que π é uma permutação de $[1, n]$ e para $i < j$ temos $a_{\pi(i)} \le a_{\pi(j)}$.

```
1  for i:=2 to n do
2    { invariante: a_1,...,a_{i-1} ordenado }
3    { coloca item i na posição correta }
4    c:=a_i
5    j:=i;
6    while c < a_{j-1} and j > 1 do
7      a_j:=a_{j-1}
8      j:=j-1
9    end while
10   a_j:=c
11 end for
```

(Nesse algoritmo é possível eliminar o teste $j > 1$ na linha 6 usando um elemento auxiliar $a_0 = -\infty$.)

Para a complexidade pessimista obtemos

$$c_p[SI](n) \leq \sum_{2\leq i\leq n} \sum_{1<j\leq i} O(1) = \sum_{2\leq i\leq n} = O(n^2)$$

$\Diamond$

Exemplo 2.8 (Máximo)
(Veja [27, cap. 3.3]).

Máximo

MÁXIMO

Entrada Uma seqüência de números $a_1, \ldots, a_n$ com $n > 0$.

Saída O máximo $m = \max_i a_i$.

```
1  m := a_1
2  for i := 2,...,n do
3     if a_i > m then
4        m := a_i
5     end if
6  end for
7  return m
```

Para a análise supomos que toda operação básica (atribuição, comparação, aritmética) têm um custo constante. Podemos obter uma cota superior simples de $O(n)$ observando que o laço sempre executa um número fixo de operações (ao máximo dois no corpo). Para uma análise mais detalhada vamos denotar o custo em números de operações de cada linha como l_i e supomos que toda operação básico custa 1 e a linha 2 do laço custa dois ($l_2 = 2$, para fazer um teste e um incremento), então temos

$$l_1 + (n-1)(l_2 + l_3) + kl_4 + l_7 = 3n + k - 1$$

com um número de execuções da linha 4 ainda não conhecido k. No melhor caso temos $k = 0$ e custos de $3n - 1$. No pior caso $m = n - 1$ e custos de $4n - 2$. É fácil ver que assintoticamente todos os casos, inclusive o caso médio, têm custo $\Theta(n)$. $\Diamond$

Exemplo 2.9 (Busca seqüencial)
O segundo algoritmo que queremos estudar é a busca seqüencial.

Busca seqüencial

BUSCA SEQÜENCIAL

Entrada Uma seqüência de números $a_1, \ldots, a_n$ com $n > 0$ e um chave c.

Saída A primeira posição p tal que $a_p = c$ ou $p = \infty$ caso não existe tal posição.

```
1  for i := 1, ..., n do
2    if a_i = c then
3      return i
4    end if
5  end for
6  return ∞
```

Busca seqüencial

- Fora do laço nas linhas 1–5 temos uma contribuição constante.
- Caso a seqüência não contém a chave c, temos que fazer n iterações.
- Logo temos complexidade pessimista $\Theta(n)$.

$\Diamond$

Counting-Sort

COUNTING-SORT

Entrada Um inteiro k, uma seqüência de números $a_1, \ldots, a_n$ e uma seqüência de contadores $c_1, \ldots, c_n$.

Saída Uma seqüência ordenada de números $b_1, \ldots, b_n$.

```
1  for i := 1, ..., k do
2    c_i := 0
3  end for
4  for i := 1, ..., n do
5    c_{a_i} := c_{a_i} + 1
6  end for
```

```
 7  for i := 2, ..., k do
 8      c_i := c_i + c_{i-1}
 9  end for
10  for i := n, ..., 1 do
11      b_{c_{a_i}} := a_i
12      c_{a_i} := c_{a_i} - 1
13  end for
14  return b_1, ..., b_n
```

Loteria Esportiva

LOTERIA ESPORTIVA

Entrada Um vetor de inteiros $r[1, \ldots, n]$ com o resultado e uma matriz A_{13xn} com as apostas dos n jogadores.

Saída Um vetor $p[1, \ldots, n]$ com os pontos feitos por cada apostador.

```
 1  i := 1
 2  while i ≤ n do
 3      p_i := 0
 4      for j de 1, ..., 13 do
 5          if A_{i,j} = r_j then p_i := p_i + 1
 6      end for
 7      i := i + 1
 8  end while
 9  for i de1, ..., n do
10      if p_i = 13 then print(Apostador i é ganhador!)
11  end for
12  return p
```

Exemplo 2.10 (Busca Binária)

Busca Binária

Busca Binária

Entrada Um inteiro x e uma seqüência $S = a_1, a_2, \ldots a_n$ de números ordenados.

Saída Posição i em que x se encontra na seqüência S ou -1 caso $x \notin S$.

```
1  i := 1
2  f := n
3  m := ⌊(f−i)/2⌋ + i
4  while i ≤ f do
5      if a_m = x then return m
6      if a_m < x then f := m − 1
7      else i := m + 1
8      m := ⌊(f−i)/2⌋ + i
9  end while
10 return −1
```

A busca binária é usada para encontrar um dado elemento numa seqüência ordenada de números com gaps. Ex: 3,4,7,12,14,18,27,31,32...n. Se os números não estiverem ordenados um algoritmo linear resolveria o problema, e no caso de números ordenados e sem gaps (nenhum número faltante na seqüência, um algoritmo constante pode resolver o problema. No caso da busca binária, o pior caso acontece quando o último elemento que for analisado é o procurado. Neste caso a seqüência de dados é dividida pela metade até o término da busca, ou seja, no máximo $\log_2 n = x$ vezes, ou seja $2^x = n$. Neste caso

$$\begin{aligned} C_p[a] = &= \sum_{i=1}^{\log_2 n} c \\ &= O(\log_2 n) \end{aligned}$$

◊

Exemplo 2.11 (Busca em Largura)

Busca em Largura

Busca em Largura

Entrada Um nó origem s e um grafo direcionado estruturado como uma seqüência das listas de adjacências de seus nós.

Saída Posição i vetor de distâncias (número de arcos) de cada nó ao origem.

```
1  for cada vértice u ∈ V − {s} do
2      c_u :=BRANCO
3      d_u = ∞
4  end for
5  c_s :=CINZA
6  d_s := 0
7  Q := ∅
8  Enqueue(Q, s)
9  while Q ≠ ∅
10   u := Dequeue(Q)
11   for cada v ∈ Adj(u)
12       if c_v =BRANCO
13         then c_v =CINZA
14         d_v = d_u + 1
15         Enqueue(Q, v)
16       end if
17   end for
18   c_u =PRETO
19 end while
```

Este algoritmo, bem como sua análise, estão disponíveis no livro do Cormen [7]. O laço **while** (linhas 9-19) será executado no máximo $|V|$ vezes, ou seja, para cada nó do grafo. Já o laço **for** (linhas 11-17) executará d_u vezes, sendo d_u o grau de saída do nó u. Desta forma, os dois laços executarão $|E|$ vezes (este tipo de análise se chama de análise agregada). Como o primeiro laço **for** do algoritmo tem complexidade $O(|V|)$, então a complexidade total do algoritmo será $O(|V| + |M|$. ◊

Exercícios sobre Ordens assintóticas

- (Exercício Adicional 2.9) Suponha que f e g são funções polinomiais e $\mathbb{N}$ em $\mathbb{N}$: $f(n) \in \Theta(n^r)$ e $g(n) \in \Theta(n^s)$. O que se pode afirmar sobre a função composta $g(f(n))$?

- (Exercício adicional 2.3.e) Classifique as funções $f(n) = 5 \cdot 2^n + 3$ e $g(n) = 3 \cdot n^2 + 5 \cdot n$ como $f \in O(g)$, $f \in \Theta(g)$ ou $f \in \Omega(g)$.

- Verifique se $2^n \cdot n \in \Theta(2^n)$ e se $2^{n+1} \in \Theta(2^n)$.

Exemplo 2.12 (Multiplicação de matrizes)
O algoritmo padrão da computar o produto $C = AB$ de matrizes (de tamanho $m \times n$, $n \times o$) segue diretamente a definição

$$c_{ik} = \sum_{1 \le j \le n} a_{ij} b_{jk} \qquad 1 \le i \le m; 1 \le k \le o$$

e resulta na implementação

MULTIPLICAÇÃO DE MATRIZES

Entrada Duas matrizes $A = (a_{ij}) \in \mathbb{R}^{m \times n}$, $B = (b_{jk}) \in \mathbb{R}^{n \times o}$.

Saída O produto $C = (c_{ik}) = AB \in \mathbb{R}^{m \times o}$.

```
1  for i := 1,...,m do
2    for k := 1,...,o do
3      c_ik := 0
4      for j := 1,...,n do
5        c_ik := c_ik + a_ij b_jk
6      end for
7    end for
8  end for
```

No total, precisamos $mno(M+A)$ operações, com M denotando multiplicações e A adições. É costume estudar a complexidade no caso $n = m = o$ e somente considerar as multiplicações, tal que temos uma entrada de tamanho $\Theta(n^2)$ e $\Theta(n^3)$ operações[1].
Esse algoritmo não é o melhor possível: Temos o algoritmo de Strassen que precisa somente $n^{\log_2 7} \approx n^{2.807}$ multiplicações (o algoritmo está detalhado no capítulo 6.3) e o algoritmo de Coppersmith-Winograd com $n^{2.376}$ multiplicações [6]. Ambas são pouco usadas na prática porque o desempenho real é melhor somente para n grande (no caso de Strassen $n \approx 700$; no caso de Coppersmith-Winograd o algoritmo não é praticável). A conjetura atual é que existe um algoritmo (ótimo) de $O(n^2)$.

◊

[1] Também é de costume contar as operações de ponto flutuante diretamente e não em relação ao tamanho da entrada. Senão a complexidade seria $2n/3^{3/2} = O(n^{3/2})$.

2.3. Complexidade média

Nesse capítulo, vamos estudar algumas técnicas de análise da complexidade média.

Motivação

- A complexidade pessimista é pessimista demais?
- Imaginável: poucos (ou irrelevantes) instâncias aumentam ela artificialmente.
- Isso motiva a estudar a complexidade média.
- Para tamanho n, vamos considerar
  - O espaço amostral $D_n = \{d \in D | \operatorname{tam}(d) = n\}$
  - Uma distribuição de probabilidade Pr sobre D_n
  - A variável aleatória desemp$[a]$
  - O custo médio

$$C_m[a](n) = E[\text{desemp}[a]] = \sum_{d \in D_n} P(d)\,\text{desemp}[a](d)$$

Tratabilidade?

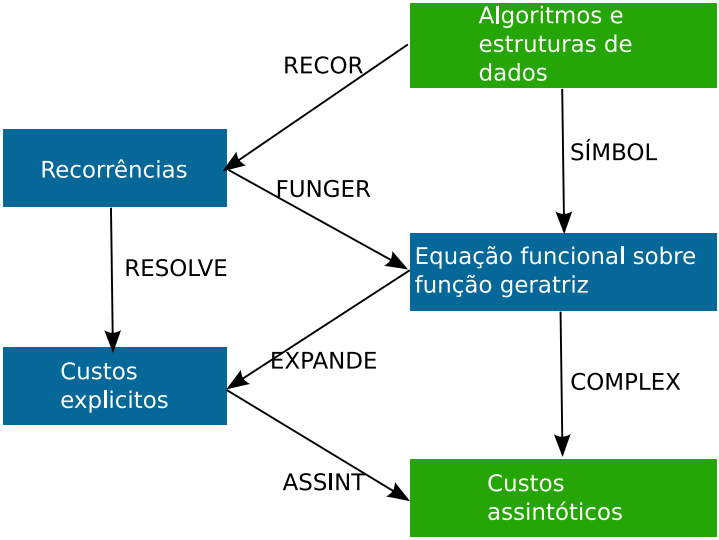
- Esperança: Problemas intratáveis viram tratável?
- Exemplos de tempo esperado:
  - Caminho Hamiltoniano: linear!
  - Parada não-determinístico em k passos: fica NP-completo.

(Resultados citados: [11, 9] (Caminho Hamiltanio), [31] (Parada em k passos).)

Criptografia

- Criptografia somente é possível se existem "funções única" (inglês: one-way functions).
- Uma função sem saída $f : \{0,1\}^* \to \{0,1\}^*$ é tal que
  - dado x, computar $f(x)$ é fácil (eficiente)
  - dada $f(x)$ achar um x' tal que $f(x') = f(x)$ é difícil

Método



(Vitter, Flajolet, [30])

Exemplo 2.13 (Busca seqüencial)
(Continuando exemplo 2.9.)

Busca seqüencial

- Caso a chave fica na i-ésima posição, temos que fazer i iterações.
- Supondo uma distribuição uniforme da posição da chave na seqüência, temos

$$\frac{1}{n+1}\left(n + \sum_{1 \leq i \leq n} i\right)$$

iterações e uma complexidade média de $\Theta(n)$.

$\Diamond$

Exemplo 2.14
(Continuando o exemplo 2.1.)
Seja n um tamanho fixo. Para BUSCA1 temos o espaço amostral $D_n = \{(a_1, \ldots, a_n) | a_1 \geq 1, \ldots, a_n \geq 1\}$. Para a análise da complexidade média, qual seria uma distribuição adequada das entradas? Observe que isso não

é totalmente trivial, porque temos um conjunto infinito de números naturais. Qual seria uma distribuição realística? Parece natural que na prática números maiores são menos prováveis. Logo escolhemos um processo tal que cada elemento da seqüência é gerado independentemente com a probabilidade $\Pr[a_i = n] = 2^{-n}$ (que é possível porque $\sum_{1 \le i} 2^{-i} = 1$).
Com isso temos

$$\Pr[(a_1, \ldots, a_n)] = \prod_{1 \le i \le n} 2^{-a_i}$$

e, $\Pr[a_i = 1] = 1/2$ e $\Pr[a_i \neq 1] = 1/2$. Considere a variáveis aleatórias desemp$[a]$ e

$$p = \begin{cases} \mu & \text{se o primeira 1 e na posição } \mu \\ n & \text{caso contrário} \end{cases}$$

Temos desemp$[a] = 2p + 1$ (veja os resultados do exemplo 2.1). Para i ser a primeira posição com elemento 1 as posições $1, \ldots, i-1$ devem ser diferente de 1, e a_i deve ser 1. Logo para $1 \le i < n$

$$\Pr[p = i] = \Pr[a_1 \neq 1] \cdots \Pr[a_{i-1} \neq 1] \Pr[a_i = 1] = 2^{-i}.$$

O caso $p = n$ tem duas causas: ou a posição do primeiro 1 é n ou a seqüência não contém 1. Ambas tem probabilidade 2^{-n} e logo $\Pr[p = n] = 2^{1-n}$.
A complexidade média calcula-se como

$$\begin{aligned} c_p[a](n) &= E[\text{desemp}[a]] = E[2p + 1] = 2E[p] + 1 \\ E[p] &= \sum_{i \ge 0} \Pr[p = i] i = 2^{-n} n + \sum_{1 \le i \le n} 2^{-n} n \\ &= 2^{-n} n + 2 - 2^{-n}(n + 2) = 2 - 2^{1-n} \qquad \text{(A.34)} \\ c_p[a](n) &= 5 - 2^{2-n} = O(1) \end{aligned}$$

A seguinte tabela mostra os custos médios para $1 \le n \le 9$

n	1	2	3	4	5	6	7	8	9
C_m	3	4	4.5	4.75	4.875	4.938	4.969	4.984	4.992

◊

Exemplo 2.15 (Ordenação por inserção direta)
(Continuando exemplo 2.7.)

Ordenação por inserção direta

- Qual o número médio de comparações?

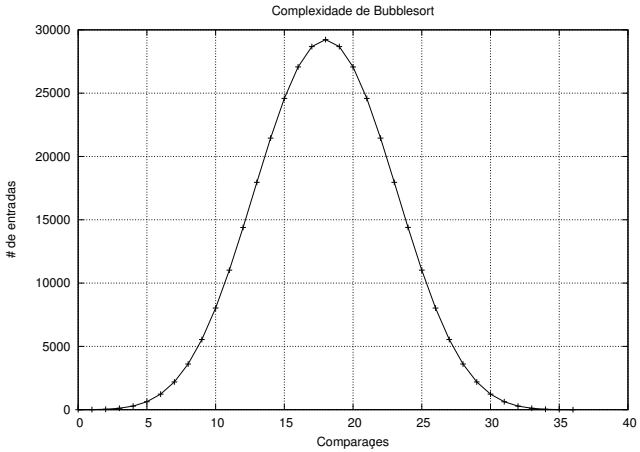
- Observação: Com as entradas distribuídas uniformemente, a posição da chave i na seqüência já ordenada também é.
- Logo chave i precisa
$$\sum_{1\leq j\leq i} j/i = \frac{i+1}{2}$$
comparações em média.
- Logo o número esperado de comparações é
$$\sum_{2\leq i\leq n} \frac{i+1}{2} = \frac{1}{2}\sum_{3\leq i\leq n+1} i = \frac{1}{2}\left(\frac{(n+1)(n+2)}{2} - 3\right) = \Theta(n^2)$$

◊

Exemplo 2.16 (Bubblesort)
(Continuando exemplo 2.6.)
O número de comparações de Bubblesort é independente da entrada. Logo, com essa operação básica temos uma complexidade pessimista e média de $\Theta(n)$.

Bubblesort: Transposições

- Qual o número de transposições em média?



Qual a complexidade contando o número de transposições? A figura em cima mostra no exemplo das instâncias com tamanho 9, que o trabalho varia entre 0 transposições (seqüência ordenada) e 36 transposições (caso pior) e na maioria dos casos, precisamos 18 transposições. A análise no caso que todas entradas são permutações de $[1, n]$ distribuídas uniformemente resulta em $n/4(n-1)$ transposições na média, a metade do caso pior.

Inversões

- Uma *inversão* em uma permutação é um par que não é em ordem, i.e.

$$i < j \quad \text{tal que} \quad a_i > a_j.$$

- Exemplos

Permutação	#Inversões
123	0
132	1
213	1
231	2
312	2
321	3

- Freqüentemente o número de inversões facilite a análise de algoritmos de ordenação.

- Em particular para algoritmos com transposições de elementos adjacentes: $\#\text{Transposições} = \#\text{ Inversões}$

Número médio de transposições

- Considere o conjunto de todas permutações S_n sobre $[1, n]$.

- Denota com $\text{inv}(\pi)$ o número de inversões de uma permutação.

- Para cada permutação π existe uma permutação π^- correspondente com ordem inversa:

$$35124; \qquad 42153$$

- Cada inversão em π não é inversão em π^- e vice versa:

$$\text{inv}(\pi) + \text{inv}(\pi^-) = n(n-1)/2.$$

Número médio de transposições

- O número médio de inversões é

$$\begin{aligned}
1/n! \sum_{\pi \in S_n} \mathrm{inv}(\pi) &= 1/(2n!) \left(\sum_{\pi \in S_n} \mathrm{inv}(\pi) + \sum_{\pi \in S_n} \mathrm{inv}(\pi) \right) \\
&= 1/(2n!) \left(\sum_{\pi \in S_n} \mathrm{inv}(\pi) + \sum_{\pi \in S_n} \mathrm{inv}(\pi^-) \right) \\
&= 1/(2n!) \left(\sum_{\pi \in S_n} \mathrm{inv}(\pi) + \mathrm{inv}(\pi^-) \right) \\
&= 1/(2n!) \left(\sum_{\pi \in S_n} n(n-1)/2 \right) \\
&= n(n-1)/4
\end{aligned}$$

- Logo, a complexidade média (de transposições) é $\Theta(n^2)$.

$\Diamond$

Exemplo 2.17 (Máximo)
(Continuando exemplo 2.8.)
Queremos analisar o número médio de atribuições na determinação do máximo, i.e. o número de atualizações do máximo.

Máximo

- Qual o número esperado de atualizações no algoritmo MÁXIMO?
- Para uma permutação π considere a *tabela de inversões* $b_1, \ldots, b_n$.
- b_i é o número de elementos na esquerda de i que são maiores que i.
- Exemplo: Para 53142

b_1	b_2	b_3	b_4	b_5
2	3	1	1	1

- Os b_i obedecem $0 \leq b_i \leq n - i$.

Tabelas de inversões

- Observação: Cada tabela de inversões corresponde com uma permutação e vice versa.
- Exemplo: A permutação correspondente com

b_1	b_2	b_3	b_4	b_5
3	1	2	1	0

- Vantagem para a análise: Podemos escolher os b_i independentemente.
- Observação, na busca do máximo i é máximo local se todos números no seu esquerdo são menores, i.e. se todos números que são maiores são no seu direito, i.e. se $b_i = 0$.

Número esperado de atualizações

- Seja X_i a variável aleatória $X_i = [i \text{ é máximo local}]$.
- Temos $\Pr[X_i = 1] = \Pr[b_i = 0] = 1/(n - i + 1)$.
- O número de máximos locais é $X = \sum_{1 \le i \le n} X_i$.
- Portanto, o número esperado de máximos locais é

$$\begin{aligned} E[X] &= E\left[\sum_{1 \le i \le n} X_i\right] = \sum_{1 \le i \le n} E[X_i] \\ &= \sum_{1 \le i \le n} \Pr[X_i] = \sum_{1 \le i \le n} \frac{1}{n - i + 1} = \sum_{1 \le i \le n} \frac{1}{i} = H_n \end{aligned}$$

- Contando atualizações: tem uma a menos que os máximos locais $H_n - 1$.

$\Diamond$

Exemplo 2.18 (Quicksort)
Nessa seção vamos analisar é Quicksort, um algoritmo de ordenação que foi inventado pelo C.A.R. Hoare em 1960 [14].

Quicksort

- Exemplo: o método Quicksort de ordenação por comparação.
- Quicksort usa divisão e conquista.
- Idéia:
  - Escolhe um elemento (chamado pivô).
  - Divide: Particione o vetor em elementos menores que o pivô e maiores que o pivô.
  - Conquiste: Ordene as duas partições recursivamente.

Particionar

PARTITION

Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída Um índice $m \in [l, r]$ e a com elementos ordenados tal que $a_i \leq a_m$ para $i \in [l, m[$ e $a_m \leq a_i$ para $i \in]m, r]$.

```
1  escolhe um pivô a_p
2  troca a_p e a_r
3  i := l − 1 { último índice menor que pivô }
4  for j:=l to r − 1 do
5     if a_j ≤ a_r then
6        i:=i+1
7        troca a_i e a_j
8     end if
9  end for
10 troca a_{i+1} e a_r
11 return i + 1
```

Escolher o pivô

- PARTITION combina os primeiros dois passos do Quicksort..
- Operações relevantes: *Número de comparações* entre chaves!

- O desempenho de PARTITION depende da escolha do pivô.
- Dois exemplos
  - Escolhe o primeiro elemento.
  - Escolhe o maior dos primeiros dois elementos.
- Vamos usar a segunda opção.

Complexidade de particionar

- O tamanho da entrada é $n = r - l + 1$
- Dependendo da escolha do pivô: precisa nenhuma ou uma comparação.
- O laço nas linhas 4–9 tem $n - 1$ iterações.
- O trabalho no corpo do laço é $1 = \Theta(1)$ (uma comparação)
- Portanto temos a complexidade pessimista

$$c_p[\text{Partition}] = n - 1 = n = \Theta(n)$$
$$c_p[\text{Partition}] = n - 1 + 1 = n = \Theta(n).$$

Quicksort

QUICKSORT

Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída a com os elementos em ordem não-decrescente, i.e. para $i < j$ temos $a_i \leq a_j$.

```
1  if l < r then
2    m := Partition(l,r,a);
3    Quicksort(l,m-1,a);
4    Quicksort(m+1,r,a);
5  end if
```

$$\begin{aligned}\text{desemp}[QS](a_l,\ldots,a_r) &= \text{desemp}[P](a_l,\ldots,a_r)\\ &\quad + \text{desemp}[QS](a_l,\ldots,a_{m-1}) + \text{desemp}[QS](a_{m+1},\ldots,a_r)\end{aligned}$$

$$\text{desemp}[QS](a_l,\ldots,a_r) = 0 \qquad \text{se } l \geq r$$

Complexidade pessimista

- Qual a complexidade pessimista?
- Para entrada $d = (a_1,\ldots,a_n)$, sejam $d_l = (a_l,\ldots,a_{m-1})$ e $d_r = (a_{m+1},\ldots,a_r)$

$$\begin{aligned}c_p[QS](n) &= \max_{d\in D_n} \text{desemp}[P](d) + \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)\\ &= n + \max_{d\in D_n} \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)\\ &= n + \max_{1\leq i\leq n} c_p[QS](i-1) + c_p[QS](n-i)\end{aligned}$$

- $c_p[QS](0) = c_p[QS](1) = 0$

Esse análise é válido para escolha do maior entre os dois primeiros elementos como pivô. Também vamos justificar o último passo na análise acima com mais detalhes. Seja $D_n = \bigcup_i D_n^i$ uma partição das entradas com tamanho n tal que para $d \in D_n^i$ temos $|d_l| = i-1$ (e conseqüentemente $|d_r| = n-i$). Então

$$\begin{aligned}&\max_{d\in D_n} \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)\\ =\ &\max_{1\leq i\leq n} \max_{d\in D_n^i} \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r) \qquad \text{separando } D_n\\ =\ &\max_{1\leq i\leq n} \max_{d\in D_n^i} c_p[QS](i-1) + c_p[QS](n-i)\end{aligned}$$

e o último passo é justificado, porque a partição de uma permutação aleatória gera duas partições aleatórias, e existe uma entrada d em que as duas subpartições assumem o máximo. Para determinar o máximo da última expressão, podemos observar que ele deve ocorrer no índice $i = 1$ ou $i = \lfloor n/2 \rfloor$ (porque $f(i) = c_p[QS](i-1) + c_p[QS](n-i)$ é simétrico com eixo de simetria $i = n/2$).

Complexidade pessimista

- O máximo ocorre para $i = 1$ ou $i = n/2$
- Caso $i = 1$

$$\begin{aligned} c_p[QS](n) &= n + c_p[QS](0) + c_p[QS](n-1) = n + c_p[QS](n-1) \\ &= \cdots = \sum_{1 \le i \le n} (i-1) = \Theta(n^2) \end{aligned}$$

- Caso $i = n/2$

$$\begin{aligned} c_p[QS](n) &= n + 2c_p[QS](n/2) = n - 1 + 2((n-1)/2 + c_p(n/4)) \\ &= \cdots = \Theta(n \log_2 n) \end{aligned}$$

- Logo, no caso pior, Quicksort precisa $\Theta(n^2)$ comparações.
- No caso bem balanceado: $\Theta(n \log_2 n)$ comparações.

Complexidade média

- Seja X a variável aleatória que denota a posição (inglês: rank) do pivô a_m na seqüência.
- Vamos supor que todos elementos a_i são diferentes (e, sem perda da generalidade, uma permutação de $[1, n]$).

$$\begin{aligned} c_m[QS] &= \sum_{d \in D_n} \Pr[d] \operatorname{desemp}[QS](d) \\ &= \sum_{d \in D_n} \Pr[d](\operatorname{desemp}[P](d) + \operatorname{desemp}[QS](d_l) + \operatorname{desemp}[QS](d_r)) \\ &= n + \sum_{d \in D_n} \Pr[d](\operatorname{desemp}[QS](d_l) + \operatorname{desemp}[QS](d_r)) \\ &= n + \sum_{1 \le i \le n} \Pr[X = i](c_m[QS](i-1) + c_m[QS](n-i)) \end{aligned}$$

Novamente, o último passo é o mais difícil de justificar. A mesma partição que aplicamos acima leva a

$$\begin{aligned}
&\sum_{d\in D_n} \Pr[d](\text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)) \\
&= \sum_{1\le i\le n} \sum_{d\in D_n^i} \Pr[d](\text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)) \\
&= \sum_{1\le i\le n} \frac{1}{|D|} \sum_{d\in D_n^i} (\text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)) \\
&= \sum_{1\le i\le n} \frac{|D_n^i|}{|D|} (c_m[QS](i-1) + c_m[QS](n-i)) \\
&= \sum_{1\le i\le n} \Pr[X=i](c_m[QS](i-1) + c_m[QS](n-i))
\end{aligned}$$

é o penúltimo passo é correto, porque a média do desempenho as permutações d_l e d_r é a mesma que sobre as permutações com $i-1$ e $n-i$, respectivamente: toda permutação ocorre com a mesma probabilidade e o mesmo número de vezes (para mais detalhes veja [18, p. 119]).
Se denotamos o desempenho com $T_n = c_p[QS](n)$, obtemos a recorrência

$$T_n = n + \sum_{1\le i\le n} \Pr[X=i]\,(T_{i-1} + T_{n-i})$$

com base $T_n = 0$ para $n \le 1$. A probabilidade de escolher o i-gésimo elemento como pivô depende da estratégia da escolha. Vamos estudar dois casos.

1. Escolhe o primeiro elemento como pivô. Temos $\Pr[X=i] = 1/n$. Como $\Pr[X=i]$ não depende do i a equação acima vira

   $$T_n = n - 1 + 2/n \sum_{0\le i<n} T_i$$

   (com uma comparação a menos no particionamento).

2. Escolhe o maior dos dois primeiros elementos[2]. Temos $\Pr[X=i] = 2(i-1)/(n(n-1))$.

[2]Supomos para análise que todos elementos são diferentes. Um algoritmo prático tem que considerar o caso que um ou mais elementos são iguais (veja [27, p. 72])

$$
\begin{aligned}
T_n &= n + 2/(n(n-1)) \sum_{1 \le i \le n} (i-1)\,(T_{i-1} + T_{n-i}) \\
&= n + 2/(n(n-1)) \sum_{0 \le i < n} i\,(T_i + T_{n-i-1}) \\
&= n + 2/(n(n-1)) \sum_{0 \le i < n} iT_i + \sum_{0 \le i < n} iT_{n-i-1} \\
&= n + 2/(n(n-1)) \sum_{0 \le i < n} iT_i + \sum_{0 \le i < n} (n-i-1)T_i \\
&= n + 2/n \sum_{0 \le i < n} T_i
\end{aligned}
$$

Recorrência

- A solução final depende da escolha do pivô.
- Dois exemplos
  - Escolhe o primeiro elemento: $\Pr[X = i] = 1/n$.
  - Escolhe o maior dos primeiros dois elementos diferentes: $\Pr[X = i] = 2(i-1)/(n(n-1))$.
- Denota $T_n = c_m[QS](n)$
- Ambas soluções chegam (quase) na mesma equação recorrente

$$T_n = n + 2/n \sum_{0 \le i < n} T_i$$

Exemplo 2.19
Vamos determinar a probabilidade de escolher o pivô $\Pr[X = i]$ no caso $n = 3$ explicitamente:

Permutação	Pivô
123	2
132	3
213	2
231	3
312	3
321	3

Logo temos as probabilidades

Pivô i	1	2	3
$\Pr[X = i]$	0	1/3	2/3

$\Diamond$

Resolver a equação

- A solução da recorrência

$$T_n = n + 1 + 2/n \sum_{0 \le i < n} T_i$$

é

$$T_n = \Theta(n \ln n)$$

- Logo, em ambos casos temos a complexidade média de $\Theta(n \ln n)$.

$$T_n = n + 1 + 2/n \sum_{0 \le i < n} T_i \qquad \text{para } n > 0$$

multiplicando por n obtemos

$$nT_n = n^2 + n + 2 \sum_{0 \le i < n} T_i$$

a mesmo equação para $n - 1$ é

$$(n-1)T_{n-1} = (n-1)^2 + n - 1 + 2 \sum_{0 \le i < n-1} T_i \qquad \text{para } n > 1$$

subtraindo a segunda da primeira obtemos

$$\begin{aligned} nT_n - (n-1)T_{n-1} &= 2n + 2T_{n-1} \qquad \text{para } n > 0, \text{ verificando } n = 1 \\ nT_n &= (n+1)T_{n-1} + 2n \end{aligned}$$

multiplicando por $2/(n(n+1))$

$$\frac{2}{n+1} T_n = \frac{2}{n} T_{n-1} + \frac{4}{n+1}$$

substituindo $A_n = 2T_n/(n+1)$

$$A_n = A_{n-1} + \frac{2}{n+1} = \sum_{1 \le i \le n} \frac{4}{i+1}$$

e portanto

$$\begin{aligned} T_n &= 2(n+1) \sum_{1 \le i \le n} \frac{1}{i+1} \\ &= 2(n+1)(H_n - \frac{n}{n+1}) \\ &= \Theta(n \ln n) \end{aligned}$$

$\Diamond$

2.4. Exercícios

(Soluções a partir da página 223.)

Exercício 2.1
Qual a complexidade pessimista dos seguintes algoritmos?

ALG1

Entrada Um tamanho de problema n.

```
1  for i := 1...n do
2     for j := 1...2^i
3        operações constantes
4        j := j + 1
5     end for
6  end for
```

ALG2

Entrada Um tamanho de problema n.

```
1  for i := 1...n do
2    for j := 1...2^i
3      operações com complexidade O(j^2)
4      j := j + 1
5    end for
6  end for
```

ALG3

Entrada Um tamanho de problema n.

```
1  for i := 1...n do
2    for j := i...n
3      operações com complexidade O(2^i)
4    end for
5  end for
```

ALG4

Entrada Um tamanho de problema n.

```
1  for i := 1...n do
2    j := 1
3    while j ≤ i do
4      operações com complexidade O(2^j)
5      j := j+1
6    end for
7  end for
```

ALG5

Entrada Um tamanho de problema n.

```
1  for i := 1...n do
```

```
2     j := i
3     while j ≤ n do
4       operações com complexidade O(2^j)
5       j := j+1
6     end for
7   end for
```

Exercício 2.2
Tentando resolver a recorrência

$$T_n = n - 1 + 2/n \sum_{0 \le i < n} T_i$$

que ocorre na análise do Quicksort (veja exemplo 2.18), o aluno J. Rapidez chegou no seguinte resultado: Supomos que $T_n = O(n)$ obtemos

$$\begin{aligned} T_n &= n - 1 + 2/n \sum_{0 \le i < n} O(i) \\ &= n - 1 + 2/n O(n^2) = n - 1 + O(n) = O(n) \end{aligned}$$

e logo, a complexidade média do Quicksort é $O(n)$. Qual o problema?

Exercício 2.3
Escreve um algoritmo que determina o segundo maior elemento de uma seqüência $a_1, \ldots, a_n$. Qual a complexidade pessimista dele considerando uma comparação como operação básica?

Exercício 2.4
Escreve um algoritmo que, dado uma seqüência $a_1, \ldots, a_n$ com $a_i \in \mathbb{N}$ determina um conjunto de índices $C \subseteq [1, n]$ tal que

$$\left| \sum_{i \in C} a_i - \sum_{i \notin C} a_i \right|$$

é mínimo. Qual a complexidade pessimista dele?

Exercício 2.5
Qual o número médio de atualizações no algoritmo

```
1  s := 0
2  for i = 1,...,n do
3     if i > ⌊n/2⌋ then
4        s:=s+i
5     end if
6  end for
```

Exercício 2.6
COUNT6

Entrada Uma seqüência $a_1, \ldots, a_n$ com $a_i \in [1, 6]$.

Saída O número de elementos tal que $a_i = 6$.

```
1  k := 0
2  for i = 1,...,n do
3     if a_i = 6 then
4        k := k + 1
5     end if
6  end for
```

Qual o número médio de atualizações $k := k + 1$, supondo que todo valor em cada posição da seqüência tem a mesma probabilidade? Qual o número médio com a distribuição $P[1] = 1/2$, $P[2] = P[3] = P[4] = P[5] = P[6] = 1/10$?

Exercício 2.7
Suponha um conjunto de chaves numa árvore binária completa de k níveis e suponha uma busca binária tal que cada chave da árvore está buscada com a mesma probabilidade (em particular não vamos considerar o caso que uma chave buscada não pertence à árvore.). Tanto nós quanto folhas contém chaves. Qual o número médio de comparações numa busca?

Exercício 2.8
Usando a técnica para resolver a recorrência (veja p. 62)

$$T_n = n + 1 + 2/n \sum_{0 \le i < n} T_i$$

resolve as recorrências

$$T_n = n + 2/n \sum_{0 \le i < n} T_i$$

$$T_n = n - 1 + 2/n \sum_{0 \le i < n} T_i$$

explicitamente.

Parte II.

Projeto de algoritmos

3. Introdução

Resolver problemas

- *Modelar* o problema
  - Simplificar e abstrair
  - Comparar com problemas conhecidos
- *Inventar* um novo algoritmo
  - Ganhar experiência com exemplos
  - Aplicar ou variar técnicas conhecidas (mais comum)

Resolver problemas

- *Provar* a corretude do algoritmo
  - Testar só não vale
  - Pode ser informal
- *Analisar* a complexidade
- *Aplicar e validar*
  - Implementar, testar e verificar
  - Adaptar ao problema real
  - Avaliar o desempenho

4. Algoritmos gulosos

Radix omnium malorum est cupiditas.

(Seneca)

4.1. Introdução

(Veja [27, cap. 5.1.3].)

Algoritmos gulosos

- Algoritmos gulosos se aplicam a problemas de otimização.
- Idéia principal: Decide localmente.
- Um algoritmo guloso constrói uma solução de um problema
  - Começa com uma solução inicial.
  - Melhora essa solução com uma decisão *local* (gulosamente!).
  - Nunca revisa uma decisão.
- Por causa da localidade: Algoritmos gulosos freqüentemente são apropriados para processamento online.

Trocar moedas

TROCA MÍNIMA

Instância Valores (de moedas ou notas) $v_1 > v_2 > \cdots > v_n = 1$, uma soma s.

Solução Números $c_1, \ldots, c_n$ tal que $s = \sum_{1 \leq i \leq n} c_i v_i$

Objetivo Minimizar o número de unidades $\sum_{1 \leq i \leq n} c_i$.

A abordagem gulosa

```
1  for i:=1,...,n do
2     c_i := ⌊s/v_i⌋
3     s := s − c_i v_i
4  end for
```

Exemplo

Exemplo 4.1
Com $v_1 = 500, v_2 = 100, v_3 = 25, v_4 = 10, v_5 = 1$ e $s = 3.14$, obtemos $c_1 = 0, c_2 = 3, c_3 = 0, c_4 = 1, c_5 = 4$.
Com $v_1 = 300, v_2 = 157, v_3 = 1$, obtemos $v_1 = 1, v_2 = 0, v_3 = 14$.
No segundo exemplo, existe uma solução melhor: $v_1 = 0, v_2 = 2, v_3 = 0$.
No primeiro exemplo, parece que a abordagem gulosa acha a melhor solução. Qual a diferença? ◊

Uma condição simples é que todos valores maiores são múltiplos inteiros dos menores; essa condição não é necessária, porque o algoritmo guloso também acha soluções para outros sistemas de moedas, por exemplo no primeiro sistema do exemplo acima.

Lema 4.1
A solução do algoritmo guloso é a única que satisfaz

$$\sum_{i \in [m,n]} c_i v_i < v_{m-1}$$

para $m \in [2, n]$. (Ela é chamada a *solução canônica*.)

Proposição 4.1
Se $v_{i+1} | v_i$ para $1 \leq i < n$ a solução gulosa é mínima.

Prova. Sejam os divisores $v_i = f_i v_{i+1}$ com $f_i \geq 2$ para $1 \leq i < n$ e define $f_n = v_n = 1$. Logo cada valor tem a representação $v_i = f_i f_{i+1} f_{i+2} \cdots f_n$.
Seja $c_1, \ldots, c_n$ uma solução mínima. A contribuição de cada valor satisfaz $c_i v_i < v_{i-1}$ senão seria possível de substituir f_{i-1} unidades de v_i para uma de v_{i-1}, uma contradição com a minimalidade da solução (observe que isso somente é possível porque os f_i são números inteiros; senão o resto depois da substituição pode ser fracional e tem quer ser distribuído pelos valores menores

que pode causar um aumento de unidades em total). Logo $c_i \leq f_{i-1} - 1$ e temos

$$\begin{aligned}\sum_{i\in[m,n]} c_i v_i &\leq \sum_{i\in[m,n]} (f_{i-1} - 1)v_i \\ &= \sum_{i\in[m,n]} f_{i-1}v_i - \sum_{i\in[m,n]} v_i \\ &= \sum_{i\in[m,n]} v_{i-1} - \sum_{i\in[m,n]} v_i \\ &= v_{m-1} - v_n = v_{m-1} - 1 < v_{m-1}\end{aligned}$$

Agora aplique lema 4.1. ■

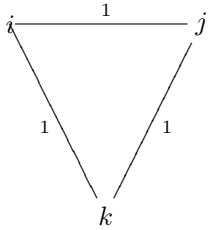
Otimalidade da abordagem gulosa

- A pergunta pode ser generalizada: Em quais circunstâncias um algoritmo guloso produz uma solução ótima?
- Se existe um solução gulosa: freqüentemente ela tem uma implementação simples e é eficiente.
- Infelizmente, para um grande número de problemas não tem algoritmo guloso ótimo.
- Uma condição (que se aplica também para programação dinâmica) é a *subestrutura ótima*.
- A teoria de *matroides* e *greedoides* estuda as condições de otimalidade de algoritmos gulosos.

Definição 4.1 (Subestrutura ótima)
Um problema de otimização tem *subestrutura ótima* se uma solução ótima (mínima ou máxima) do problema consiste em soluções ótimas das subproblemas.

Exemplo 4.2
Considere caminhos (simples) em grafos. O caminho mais curto $v_1 v_2 \ldots v_n$ entre dois vértices v_1 e v_n tem subestrutura ótima, porque um subcaminho também é mais curto (senão seria possível de obter um caminho ainda mais curto).

Do outro lado, o caminho mais longo entre dois vértices $v_1 \dots v_n$ não tem subestrutura ótima: o subcaminho $v_2 \dots v_n$, por exemplo, não precisa ser o caminho mais longo. Por exemplo no grafo



o caminho mais longo entre i e j é ikj, mas o subcaminho kj não é o subcaminho mais longo entre k e j. ◊

Para aplicar a definição 4.1 temos que conhecer (i) o conjunto de subproblemas de um problema e (ii) provar, que uma solução ótima contém uma (sub-)solução que é ótima para um subproblema. Se sabemos como estender uma solução de um subproblema para uma solução de todo problema, a subestrutura ótima fornece um algoritmo genérico da forma

Solução genérica de problemas com subestrutura ótima

Entrada Uma instância I de um problema.

Saída Uma solução ótima S^* de I.

```
1  resolve(I):=
2    S*:=nil                          { melhor solução }
3    for todos subproblemas I' de I do
4      S' :=resolve(I')
5      estende S' para uma solução S de I
6      if S' é a melhor solução then
7        S* := S
8      end if
9    end for
```

Informalmente, um algoritmo guloso é caracterizado por uma subestrutura ótima e a característica adicional, que podemos escolher o subproblema que leva a solução ótima através de uma regra simples. Portanto, o algoritmo guloso evite resolver todos subproblemas.

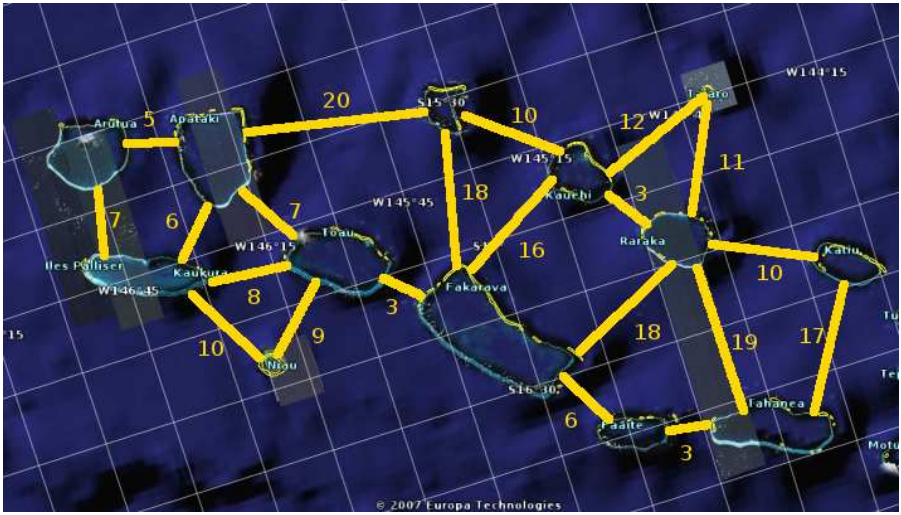
Uma subestrutura ótima é uma condição necessária para um algoritmo guloso ou de programação dinâmica ser ótima, mas ela não é suficiente.

4.2. Algoritmos em grafos

4.2.1. Árvores espalhadas mínimas

Motivação

Pontes para Polinésia Francesa!



Árvores espalhadas mínimas (AEM)

Árvore espalhada mínima (AEM)

Instância Grafo conexo não-direcionado $G = (V, E)$, pesos $c : E \to \mathbb{R}^+$.

Solução Um subgrafo $H = (V_H, E_H)$ conexo.

Objetivo Minimiza os custos $\sum_{e \in E_H} c(e)$.

- Um subgrafo conexo com custo mínimo deve ser uma árvore (por quê?).
- Grafo não conexo: Busca uma árvore em todo componente (floresta mínima).

Aplicações

- Redes elétricas
- Sistemas de estradas
- Pipelines
- Caixeiro viajante
- Linhas telefônicas alugadas

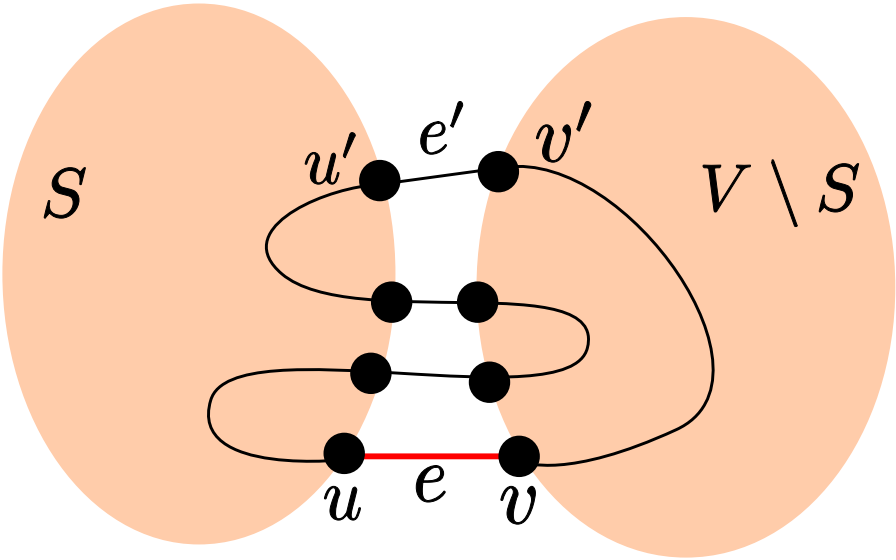
Resolver AEM

- O número de árvores espalhadas pode ser exponencial.
- Como achar uma solução ótima?
- Observação importante

Lema 4.2 (Corte)
Considere um *corte* $V = S \cup V \setminus S$ (com $S \neq \emptyset$, $V \setminus S \neq \emptyset$). O arco mínimo entre S e $V \setminus S$ faz parte de qualquer AEM.

Prova. Na prova vamos supor, que todos pesos são diferentes.
Suponha um corte S tal que o arco mínimo $e = \{u, v\}$ entre S e $V \setminus S$ não faz parte de um AEM T. Em T existe um caminho de u para v que contém ao menos um arco e' que cruza o corte. Nossa afirmação: Podemos substituir e' com e, em contradição com a minimalidade do T.
Prova da afirmação: Se substituirmos e' por e obtemos um grafo T'. Como e é mínimo, ele custa menos. O novo grafo é conexo, porque para cada par de nós ou temos um caminho já em T que não faz uso de e' ou podemos obter, a partir de um caminho em T que usa e' um novo caminho que usa um desvio sobre e. Isso sempre é possível, porque há um caminho entre u e v sem e, com dois sub-caminhos de u para u' e de v' para v, ambos sem usar e'. O novo grafo também é uma árvore, porque ele não contém um ciclo. O único ciclo possível é o caminho entre u e v em T com o arco e, porque T é uma árvore. ■

Prova



Resolver AEM

- A característica do corte possibilita dois algoritmos simples:
  1. Começa com algum nó e repetidamente adicione o nó ainda não alcançável com o arco de custo mínimo de algum nó já alcançável: *algoritmo de Prim*.
  2. Começa sem arcos, e repetidamente adicione o arco com custo mínimo que não produz um ciclo: *algoritmo de Kruskal*.

AEM: Algoritmo de Prim

AEM-Prim

Entrada Um grafo conexo não-orientado $G = (V, E_G)$ com pesos $c : V_G \to \mathbb{R}^+$

Saída Uma árvore $T = (V, E_T)$ com custo $\sum_{e \in E_T} c(e)$ mínimo.

```
1  V' := {v} para um v ∈ V
2  E_T := ∅
3  while V' ≠ V do
4     escolhe e = {u, v} com custo mínimo
5        entre V' e V \ V' (com u ∈ V')
6     V' := V' ∪ {v}
7     E_T := E_T ∪ {e}
8  end while
```

AEM: Algoritmo de Kruskal

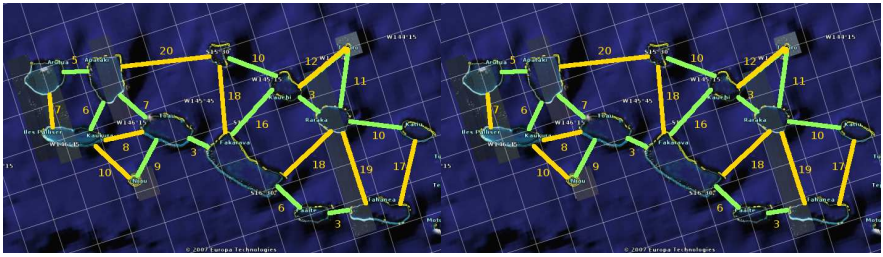
AEM-KRUSKAL

Entrada Um grafo conexo não-orientado $G = (V, E_G)$ com pesos $c : V_G \to \mathbb{R}^+$

Saída Uma árvore $T = (V, E_T)$ com custo $\sum_{e \in E_T} c(e)$ mínimo.

```
1  E_T := ∅
2  while (V, E_T) não é conexo do
3     escolha e com custo mínimo que não produz cíclo
4     E_T := E_T ∪ {e}
5  end while
```

Exemplo 4.3
Resultado dos algoritmos de Prim e Kruskal para Polinésia Francesa:



O mesmo! ◊

Implementação do algoritmo de Prim

- Problema: Como manter a informação sobre a distância mínima de forma eficiente?
- Mantenha uma distância do conjunto V' para cada nó em $V \setminus V'$.
- Nós que não são acessíveis em um passo têm distância ∞.
- Caso um novo nó seja selecionado: atualiza as distâncias.

Implementação do algoritmo de Prim

- Estrutura de dados adequada:
  - Fila de prioridade Q de pares (e, v) (chave e elemento).
  - Operação $Q.\min()$ remove e retorna (e, c) com c mínimo.
  - Operação Q.atualiza(e, c') modifica a chave de e para v' caso v' é menor que a chave atual.
- Ambas operações podem ser implementadas com custo $O(\log n)$.

AEM: Algoritmo de Prim

AEM-Prim

Entrada Um grafo conexo não-orientado $G = (V, E_G)$ com pesos $c : V_G \to \mathbb{R}^+$

Saída Uma árvore $T = (V, E_T)$ com custo $\sum_{e \in E_T} c(e)$ mínimo.

```
1  E_T := ∅
2  Q := {((v,u), c(v,u)) | u ∈ N(v)}  { nós alcancáveis }
3  Q := Q ∪ {((u,u), ∞) | u ∈ V \ N(v) \ {v}}  { nós restantes }
4  while Q ≠ ∅ do
5     ((u,v), c) := Q.min()  { (u,v) é arco mínimo }
6     for w ∈ N(v) do
7           Q.atualiza((v,w), c(v,w))
8     end for
9     E_T := E_T ∪ {{u,v}}
10 end while
```

Algoritmo de Prim

- Complexidade do algoritmo com o refinamento acima:
- O laço 4–9 precisa $n - 1$ iterações.
- O laço 6–8 precisa no total menos que m iterações.
- $c_p[\text{AEM-PRIM}] = O(n \log n + m \log n) = O(m \log n)$

Uma implementação do algoritmo de Kruskal em tempo $O(m \log n)$ também é possível. Para esta implementação é necessário de manter conjuntos que representam nós conectados de maneira eficiente. Isso leva a uma estrutura de dados conhecida como *Union-Find* que tem as operações

- $C :=$ cria(e): cria um conjunto com único elemento e.
- união(C_1, C_2): junta os conjuntos C_1 e C_2.
- $C :=$ busca(e): retorna o conjunto do elemento e.

Essas operações podem ser implementados em tempo $O(1)$, $O(1)$ e $O(\log n)$ (para n elementos) respectivamente.

4.2.2. Caminhos mais curtos

Caminhos mais curtos

- Problema freqüente: Encontra caminhos mais curtos em um grafo.
- Variações: Caminho mais curto
  - entre dois nós.
  - entre um nó e todos outros (inglês: single-source shortest paths, SSSP).
  - entre todas pares (inglês: all pairs shortest paths, APSP).

Caminhos mais curtos

CAMINHOS MAIS CURTOS

Instância Um grafo direcionado $G = (\{v_1, \ldots, v_n\}, E)$ com função de custos $c : E \to \mathbb{R}^+$ e um nó inicial $s \in V$.

Solução Uma atribuição $d : V \to \mathbb{R}^+$ da distância do caminho mais curto $d(t)$ de s para t.

Aproximação: Uma idéia

1. Começa com uma estimativa viável: $d(s) = 0$ e $d(t) = \infty$ para todo nó em $V \setminus \{s\}$.
2. Depois escolhe uma aresta $e = (u, v) \in E$ tal que $d(u) + c(e) \leq d(v)$ e atualiza $d(v) = d(u) + c(e)$.
3. Repete até não ter mais arestas desse tipo.

Esse algoritmo é correto? Qual a complexidade dele?

4.3. Algoritmos de seqüenciamento

Seqüenciamento de intervalos

Considere o seguinte problema

Seqüenciamento de intervalos

Instância Um conjunto de intervalos $S = \{[c_i, f_i], 1 \leq i \leq n\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Solução Um conjunto *compatível* $C \subseteq S$ de intervalos, i.e. cada par $i_1, i_2 \in C$ temos $i_1 \cap i_2 = \emptyset$.

Objetivo Maximiza a cardinalidade $|C|$.

(inglês: interval scheduling)

Como resolver?

- Qual seria uma boa estratégia gulosa para resolver o problema?
- Sempre selecionada o intervalo que
  - que começa mais cedo?

- que termina mais cedo?
- que começa mais tarde?
- que termina mais tarde?
- mais curto?
- tem menos conflitos?

Implementação

SEQÜENCIAMENTO DE INTERVALOS

Entrada Um conjunto de intervalos $S = \{[c_i, f_i] \mid 1 \leq i \leq n\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Saída Um conjunto máximo de intervalos compatíveis C.

```
1  C := ∅
2  while S ≠ ∅ do
3     Seja [c, f] ∈ S com f mínimo
4     C := C ∪ [c, f]
5     S := S \ {i ∈ S | i ∩ [c, f] ≠ ∅}
6  end while
7  return C
```

Seja $C = ([c_1, f_1], \ldots, [c_n, f_n])$ o resultado do algoritmo SEQÜENCIAMENTO DE INTERVALOS e $O = ([c'_1, f'_1], \ldots, [c'_m, f'_m])$ seqüenciamento máximo, ambos em ordem crescente.

Proposição 4.2
Para todo $1 \leq i \leq n$ temos $f_i \leq f'_i$.

Prova. Como O é ótimo, temos $n \leq m$. Prova por indução. Base: Como o algoritmo guloso escolhe o intervalo cujo terminação é mínima, temos $f_1 \leq f'_1$. Passo: Seja $f_i \leq f'_i$ com $i < n$. O algoritmo guloso vai escolher entre os intervalos que começam depois f_i o com terminação mínima. O próximo intervalo $[c'_{i+1}, m'_{i+1}]$ do seqüenciamento ótimo está entre eles, porque ele começa depois f'_i e $f'_i \geq f_i$. Portanto, o próximo intervalo escolhido pelo algoritmo guloso termina antes de f'_{i+1}, i.e. $f_{i+1} \leq f'_{i+1}$. ■

Proposição 4.3
O seqüenciamento do algoritmo guloso é ótimo.

Prova. Suponha que o algoritmo guloso retorna menos intervalos C que um seqüenciamento ótimo O. Pela proposição 4.2, o último (n-ésimo) intervalo do C termina antes do último intervalo de O. Como O tem mais intervalos, existe mais um intervalo que poderia ser adicionado ao conjunto C pelo algoritmo guloso, uma contradição com o fato, que o algoritmo somente termina se não sobram intervalos compatíveis. ■

Complexidade

- Uma implementação detalhada pode
  - Ordenar os intervalos pelo fim deles em tempo $O(n \log n)$
  - Começando com intervalo 1 sempre escolher o intervalo atual e depois ignorar todos intervalos que começam antes que o intervalo atual termina.
  - Isso pode ser implementado em uma varredura de tempo $O(n)$.
  - Portanto o complexidade pessimista é $O(n \log n)$.

Conjunto independente máximo

Considere o problema

> Conjunto independente máximo, CIM
>
> **Instância** Um grafo não-direcionado $G = (V, E)$.
>
> **Solução** Um conjunto *independente* $M \subseteq V$, i.e. todo $m_1, m_2 \in V$ temos $\{m_1, m_2\} \notin E$.
>
> **Objetivo** Maximiza a cardinalidade $|M|$.

(inglês: maximum independent set, MIS)

Grafos de intervalo

- Uma instância S de seqüenciamento de intervalos define um grafo não-direcionado $G = (V, E)$ com

$$V = S; \qquad E = \{\{i_1, i_2\} \mid i_1, i_2 \in S, i_1 \cap i_2 \neq \emptyset\}$$

- Grafos que podem ser obtidos pelo intervalos são *grafos de intervalo*.

- Um conjunto compatível de intervalos corresponde com um conjunto independente nesse grafo.
- Portanto, resolvemos CIM para grafos de intervalo!
- Sem restrições, CIM é NP-completo.

Variação do problema

Considere uma variação de Seqüenciamento de intervalos:

> Particionamento de intervalos
>
> **Instância** Um conjunto de intervalos $S = \{[c_i, f_i], 1 \leq i \leq n\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.
>
> **Solução** Uma atribuição de rótulos para intervalos tal que cada conjunto de intervalos com a mesma rótula é compatível.
>
> **Objetivo** Minimiza o número de rótulos diferentes.

Observação

- Uma superposição de k intervalos implica uma cota inferior de k rótulos.
- Seja d o maior número de intervalos super-posicionados (a *profundidade* do problema).
- É possível atingir o mínimo d?

Algoritmo

> Particionamento de intervalos
>
> **Instância** Um conjunto de intervalos $S = \{[c_i, f_i], 1 \leq i \leq n\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.
>
> **Solução** Uma atribuição de rótulos para os intervalos tal que cada conjunto de intervalos com a mesma rótula é compatível.

Objetivo Minimiza o número de rótulos diferentes.

```
1  Ordene S em ordem de começo crescente.
2  for i := 1,...,n do
3    Exclui rótulos de intervalos
4      precedentes conflitantes
5    Atribui ao intervalo i o número
6      inteiro mínimo ≥ 1 que sobra
```

Corretude

- Com profundidade d o algoritmo precisa ao menos d rótulos.
- De fato ele precisa exatamente d rótulos. Por qûe?
- Qual a complexidade dele?

Observações: (i) Suponha que o algoritmo precise mais que d rótulos. Então existe um intervalo tal que todos números em $[1, d]$ estão em uso pelo intervalos conflitantes, uma contradição com o fato que a profundidade é d. (ii) Depois da ordenação em $O(n \log n)$ a varredura pode ser implementada de forma que precisa $O(n)$ passos. Portanto a complexidade é $O(n \log n)$.

Coloração de grafos

Considere o problema

Coloração mínima

Instância Um grafo não-direcionado $G = (V, E)$.

Solução Uma coloração de G, i.e. uma atribuição de cores $c : V \to C$ tal que $c(v_1) \neq c(v_2)$ para $\{v_1, v_2\} \in E$.

Objetivo Minimiza o número de cores $|C|$.

- Particionamento de intervalos resolve o problema Coloração mínima para grafos de intervalo.
- Coloração mínima para grafos sem restrições é NP-completo.

4.4. Tópicos

Compressão de dados

- Seqüência genética (NM_005273.2, Homo sapiens guanine nucleotide binding protein)

  $$GATCCCTCCGCTCTGGGGAGGCAGCGCTGGCGGCGG\ldots$$

  com $1666bp$.

- Como comprimir?
  - Com código fixo:

    $$A = 00; \qquad G = 01;$$
    $$T = 10; \qquad C = 11.$$

    Resultado: $2b/bp$ e $3332b$ total.
  - Melhor abordagem: Considere as freqüências, use códigos de diferente comprimento

A	G	T	C
.18	.30	.16	.36

Códigos: Exemplos

- Tentativa 1

  $$T = 0; \qquad A = 1;$$
  $$G = 01; \qquad C = 10$$

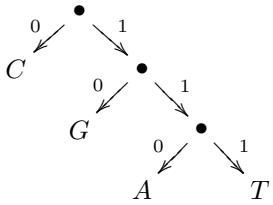
- Desvantagem: Ambíguo! $01 = TC$ ou $01 = G$?
- Tentativa 2

  $$C = 0; \qquad G = 10;$$
  $$A = 110; \qquad T = 111$$

- Custo: $0.36 \times 1 + 0.30 \times 2 + 0.18 \times 3 + 0.16 \times 3 = 1.98b/bp$, $3299b$ total.

Códigos livre de prefixos

- Os exemplos mostram
  - Dependendo das freqüências, um código com comprimento variável pode custar menos.
  - Para evitar ambigüedades, nenhum prefixo de um código pode ser outro código: ele é *livre de prefixos* (inglês: prefix-free).
- Observação: Esses códigos correspondem a árvores binárias.



Cada código livre de prefixo pode ser representado usando uma árvore binária: Começando com a raiz, o sub-árvore da esquerda representa os códigos que começam com 0 e a sub-árvore da direita representa os códigos que começam com 1. Esse processo continua em cada sub-árvore considerando os demais bits. Caso todos bits de um código foram considerados, a árvore termina nessa posição com uma folha para esse código.
Essas considerações levam diretamente a um algoritmo. Na seguinte implementação, vamos representar árvores binárias como estrutura da dados abstrata que satisfaz

BinTree ::= Nil — Node(BinTree,Bintree)

uma folha sendo um nó sem filhos. Vamos usar a abreviação

Leaf ::= Node(Nil,Nil).

PrefixTree

Entrada Um conjunto de códigos C livre de prefixos.

Saída Uma árvore binária, representando os códigos.

```
1  if |C ⊨ 0 then
2    return Nil                    { não tem árvore }
3  end if
4  if |C ⊨ {ϵ} then
5    return Leaf                 { único código vazio }
```

```
6  end if
7  Escreve C = 0C_1 ∪ 1C_2
8  return Node(PrefixTree(C_1), PrefixTree(C_2))
```

Contrariamente, temos também

Proposição 4.4
O conjunto das folhas de cada árvore binária corresponde com um código livre de prefixo.

Prova. Dado uma árvore binária com as folhas representando códigos, nenhum código pode ser prefixo de outro: senão ocorreria como nó interno. ■

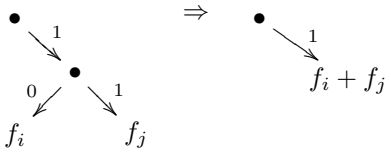
Qual o melhor código?

- A teoria de informação (Shannon) fornece um limite.
- A quantidade de informação que um símbolo que ocorre com freqüência f transmite é
$$-\log_2 f$$
e o número médio de bits (para um número grande de símbolos)
$$H = -\sum f_i \log_2 f_i.$$
- H é um limite inferior para qualquer código.
- Nem sempre é possível atingir esse limite. Com
$$A = 1/3, B = 2/3; \qquad H \approx 0.92b$$
mas o código ótimo precisa $1b$ por símbolo.
- Nosso exemplo: $H \approx 1.92$.

Como achar o melhor código?

- Observação 1: Uma solução ótima é uma árvore completa.

- Observação 2: Em uma solução ótima, os dois símbolos com menor freqüência ocorrem como irmãos no nível mais alto. Logo: Podemos substituir eles com um nó cujo freqüência é a soma dos dois.



Algoritmo

HUFFMAN

Entrada Um alfabeto de símbolos S com uma freqüência f_s para cada $s \in S$.

Saída Uma árvore binária que representa o melhor código livre de prefixos para S.

```
1  Q := {Leaf(f_s) | s ∈ S} { fila de prioridade }
2  while |Q| > 0 do
3     b_1 := Q.min() com b_1 = Node(f_i, b_i1, b_i2)
4     b_2 := Q.min() com b_2 = Node(f_j, b_j1, b_j2)
5     Q := Q.add(Node(f_i + f_j, b_1, b_2))
6  end while
```

Exemplo 4.4

Saccharomyces cerevisiae
Considere a seqüência genética do Saccharomyces cerevisiae (inglês: baker's yeast)

$$MSITNGTSRSVSAMGHPAVERYTPGHIVCVGTHKVEVV\ldots$$

com $2900352bp$. O alfabeto nessa caso são os 20 aminoácidos

$$S = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}.$$

que ocorrem com as freqüências

A	C	D	E	F	G	H	I	K	L
0.055	0.013	0.058	0.065	0.045	0.050	0.022	0.066	0.073	0.096
M	N	P	Q	R	S	T	V	W	Y
0.021	0.061	0.043	0.039	0.045	0.090	0.059	0.056	0.010	0.034

Resultados

O algoritmo HUFFMAN resulta em

A	C	D	E	F	G	H	I	K	L
0010	100110	1110	0101	0000	1000	100111	1101	0011	100
M	N	P	Q	R	S	T	V	W	Y
000111	1011	11011	01011	10111	1111	0001	1010	000110	10110

que precisa $4.201b/bp$ (compare com $\log_2 20 \approx 4.32$). ◊

4.5. Notas

Magazine, Nemhauser e Trotter [22] dão critérios necessários e suficientes para uma solução gulosa do problema de troca ser ótima.

4.6. Exercícios

(Soluções a partir da página 227.)

Exercício 4.1 (Análise de series)
Suponha uma série de eventos, por exemplo, as transações feitos na bolsa de forma

compra Dell, vende HP, compra Google, ...

Uma certa ação pode acontecer mais que uma vez nessa seqüência. O problema: Dado uma outra seqüência, decida o mais rápido possível se ela é uma subseqüência da primeira.
Achar um algoritmo eficiente (de complexidade $O(m + n)$ com seqüência de tamanho n e m), prova a corretude a analise a complexidade dele.
(Fonte: [17]).

Exercício 4.2 (Comunicação)
Imagine uma estrada comprida (pensa em uma linha) com casas ao longo dela. Suponha que todas casas querem acesso ao comunicação com celular. O problema: Posiciona o número mínimo de bases de comunicação ao longo da estrada, com a restriçaõ que cada casa tem quer ser ao máximo 4 quilômetros distante de uma base.
Inventa um algoritmo eficiente, prova a corretude e analise a complexidade dele.
(Fonte: [17]).

5. Programação dinâmica

5.1. Introdução

Temos um par de coelhos recém-nascidos. Um par recém-nascido se torne fértil depois um mês. Depois ele gera um outro par cada mês seguinte. Logo, os primeiros descendentes nascem em dois meses. Supondo que os coelhos nunca morrem, quantos pares tem depois de n meses?

n	0	1	2	3	4	$\cdots$
#	1	1	2	3	5	$\cdots$

Como os pares somente produzem filhos depois de dois meses temos

$$F_n = \underbrace{F_{n-1}}_{\text{população antiga}} + \underbrace{F_{n-2}}_{descendentes}$$

com a recorrência completa

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{caso } n \geq 2 \\ 1 & \text{caso } n \in \{0, 1\} \end{cases}$$

Os números definidos por essa recorrência são conhecidos como so *números Fibonacci*.

Uma implementação recursiva simples para calcular-os é

```
1  fib(n) :=
2    if n ≤ 1 then
3      return 1
4    else
5      return fib(n − 1)+fib(n − 2)
6    end if
7  end
```

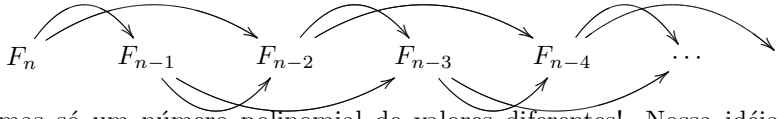
Qual a complexidade dessa implementação? Temos a recorrência de tempo

$$T(n) = \begin{cases} T(n-1) + T(n-2) + \Theta(1) & \text{caso } n \geq 2 \\ \Theta(1) & \text{caso contrário} \end{cases}$$

É simples de ver (indução!) que $T(n) \geq F_n$, e sabemos que $F_n = \Omega(2^n)$ (outra indução), portanto a complexidade é exponencial. (A fórmula exata é

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \left\lfloor \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

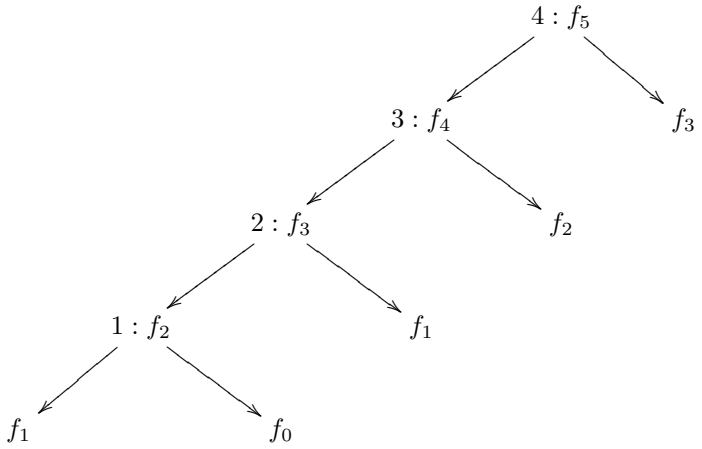
e foi publicado por Binet em 1843.)
Qual o problema dessa solução? De um lado temos um número exponencial de caminhos das chamadas recursivas



mas só um número polinomial de valores diferentes! Nossa idéia: usa um *cache*!

```
1  f_0 := 1
2  f_1 := 1
3  f_i := ⊥  para  i ≥ 2
4
5  fib(n)  :=
6     if  f_n = ⊥  then
7        f_n :=  fib(n − 1)+fib(n − 2)
8     end if
9     return  f_n
10 end
```

Exemplo de uma execução:



	f_5	f_4	f_3	f_2	f_1	f_0
Inicial	$\perp$	$\perp$	$\perp$	$\perp$	1	1
1	$\perp$	$\perp$	$\perp$	2	1	1
2	$\perp$	$\perp$	3	2	1	1
3	$\perp$	5	3	2	1	1
4	8	5	3	2	1	1

O trabalho agora é $O(n)$, i.e. linear! Essa abordagem se chama *memoização*: usamos um cache para evitar recalcular resultados intermediários freqüentes. Esse implementação e *top-down* e corresponde exatamente com a recorrência acima. Uma implementação (ligeiramente) mais eficiente que preenche o "cache" da forma *bottom-up* é

```
1  fib(n) :=
2    f_0 := 1
3    f_1 := 1
4    for i ∈ [2, n] do
5      f_i := f_{i-1} + f_{i-2}
6    end for
7    return f_n
8  end
```

Finalmente, podemos otimizar a computação no caso dos números Fibonacci ainda mais, evitando o cache totalmente

```
1  fib(n) :=
2    f := 1
3    g := 1
4    for i ∈ [2, n] do
5      { invariante: f = F_{i-2} ∧ g = F_{i-1} }
6      g:=f+g
7      f:=g  f
8      { invariante: f = F_{i-1} ∧ g = F_i }
9    end for
```

A idéia de armazenar valores intermediárias usados freqüentemente numa computação recursiva é uma das idéias principais da *programação dinâmica*. A seqüência de implementações nosso exemplo dos números Fibonacci é típico para algoritmos de programação dinâmica, inclusive o último para reduzir a complexidade de espaço. Na prática, as implementações ascendentes (ingl. *bottom-up*) são preferidas.

Programação Dinâmica

1. Para aplicar PD o problema deve apresentar substrutura ótima (uma solução ótima para um problema contém soluções ótimas de seus subproblemas) e superposição de subproblemas (reutiliza soluções de subproblemas).

2. Pode ser aplicada a problemas NP-completos e polinomiais.

3. Em alguns casos o algoritmo direto tem complexidade exponencial, enquanto que o algoritmo desenvolvido por PD é polinomial.

4. Às vezes a complexidade continua exponencial, mas de ordem mais baixa.

5. É útil quando não não é fácil chegar a uma seqüência ótima de decisões sem testar todas as seqüências possíveis para então escolher a melhor.

6. Reduz o número total de seqüências viáveis, descartando aquelas que sabidamente não podem resultar em seqüências ótimas.

Idéias básicas da PD

1. Objetiva construir uma resposta ótima através da combinação das respostas obtidas para subproblemas

2. Inicialmente a entrada é decomposta em partes mínimas e resolvidas de forma ascendente (bottom-up)

3. A cada passo os resultados parciais são combinados resultando respostas para subproblemas maiores, até obter a resposta para o problema original

4. A decomposição é feita uma única vez, e os casos menores são tratados antes dos maiores

5. Este método é chamado ascendente, ao contrário dos métodos recursivos que são métodos descendentes.

Passos do desenvolvimento de um algoritmo de PD

1. Caracterizar a estrutura de uma solução ótima
2. Definir recursivamente o valor de uma solução ótima
3. Calcular o valor de uma solução ótima em um processo ascendente
4. Construir uma solução ótima a partir de informações calculadas

5.2. Comparação de seqüências

5.2.1. Subseqüência Comum Mais Longa

Subseqüência Comum Mais Longa

- Dada uma seqüência $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Z = \langle z_1, z_2, \ldots, z_k \rangle$, X é uma subseqüência de Z se existe uma seqüência estritamente crescente $\langle i_1, i_2, \ldots, i_k \rangle$ de índices de X tais que, para todo $j = 1, 2, \ldots, k$, temos $x_{i_j} = z_j$.
- Exemplo: $Z = \langle B, C, D, B \rangle$ é uma subseqüência de $X = \langle A, B, C, D, A, B \rangle$
- Dadas duas seqüências X e Y, dizemos que uma seqüência Z é uma subseqüência comum de X e Y se Z é uma subseqüência de X e Y ao mesmo tempo.

Subseqüência Comum Mais Longa

- $Z = \langle B, C, A, B \rangle$ é uma subseqüência comum mais longa de $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$
- Definição do Problema da SCML

SCML

Instância Duas seqüencias $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$.

Solução Uma subseqüência comum Z de X, Y.

Objetivo Maximiza o comprimento de Z.

- Exemplo de Aplicação: comparar dois DNAs:

$$X = ACCGGTCGAGTG$$
$$Y = GTCGTTCGGAATGCCGTTGCTCTGTAAA$$

- Os caracteres devem aparecer na mesma ordem, mas não necessariamente ser consecutivos.

Teorema: Subestrutura Ótima de uma SCML
Sejam as seqüências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, e seja $Z = \langle z_1, z_2, \ldots, z_k \rangle$ qualquer SCML de X e Y

- Se $x_m = y_n$, então $Z_k = x_m = y_n$ e Z_{k-1} é um SCML de X_{m-1} e Y_{n-1}
- Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma SCML de X_{m-1} e Y
- Se $x_m \neq y_n$, então $z_k \neq y_m$ implica que Z é uma SCML de X e Y_{n-1}

Isso leva ao definição recursiva

$$SCML(X,Y) = \begin{cases} SCML(X',Y') + 1 & \text{se } X = X'c,\ Y = Y'c \\ \max\{SCML(X,Y'), SCML(X',Y)\} & \text{se } X = X'c_1,\ Y = Y'c_2 \text{ e } c_1 \neq c_2 \\ 0 & \text{se } X = \epsilon \text{ ou } Y = \epsilon \end{cases}$$

Qual a complexidade de implementação recursiva (naiva)? No pior caso executamos

$$T(n,m) = T(n-1,m) + T(n,m-1) + \Theta(1)$$

operações. Isso com certeza é mais que o número de caminhos de (n,m) até $(0,0)$, que é maior que $\binom{m+n}{n}$, i.e. exponential no pior caso.
Com memoização ou armazenamento de valores intermediários, podemos reduzir o tempo e espaço para $O(nm)$:

SCML

SCML

Entrada Dois strings X e Y e seus respectivos tamanhos m e n medidos em número de caracteres.

Saída O tamanho da maior subseqüência comum entre X e Y.

```
 1  m := comprimento(X)
 2  n := comprimento(Y)
 3  for i := 1 to m do c[i,0] := 0;
 4  for i := 1 to m do c[0,j] := 0;
 5  for i := 1 to m do
 6      for j := 1 to n do
 7          if x_i = y_i then
 8              c[i,j] := c[i-1,j-1] + 1
 9          else
10              c[i,j] := max(c[i,j-1], c[i-1,j]
11          end if
12      end for
13  return c[m,n]
```

Caso só o comprimento da maior subseqüência em comum importa, podemos reduzir o espaço usado. Os valores de cada linha ou coluna dependem só dos valores da linha ou coluna anterior. Supondo, que o comprimento de uma linha e menor que o comprimento de uma coluna, podemos manter duas linhas e calcular os valores linha por linha. Caso as colunas são menores, procedemos das mesmo forma coluna por coluna. Com isso, podemos determinar o comprimento da maior subseqüência em comum em tempo $O(nm)$ e espaço $O(\min\{n, m\})$.
Caso queremos recuperar a própria subseqüência, temos que manter essa informação adicionalmente:

SCML

SCML

Entrada Dois strings X e Y e seus respectivos tamanhos m e n medidos em número de caracteres.

Saída O tamanho da maior subseqüência comum entre X e Y e o vetor b para recuperar a maior SCML.

```
1  m := comprimento[X]
2  n := comprimento[Y]
3  for i := 1 to m do c[i,0] := 0;
```

```
4  for i := 1 to m do c[0,j] := 0;
5  for i := 1 to m do
6      for j := 1 to n do
7          if x_i = y_i then
8              c[i,j] := c[i-1,j-1] + 1
9              b[i,j] := ↖
10         else if c[i-1,j] ≥ c[i,j-1] then
11                 c[i,j] := c[i-1,j]
12                 b[i,j] := ↑
13             else
14                 c[i,j] := c[i,j-1]
15                 b[i,j] := ←
16 return c e b
```

Nesse caso, não tem método simples para reduzir o espaço de $O(nm)$ (veja os comentários sobre o algoritmo de Hirschberg abaixo). Mantendo duas linhas ou colunas de c, gasta menos recursos, mas para recupar a subseqüência comum, temos que manter $O(nm)$ elementos em b.
O algoritmo de Hirschberg [13]), via Divisão e Conquista, resolve o problema da subseqüência comum mais longa em tempo $O(mn)$, mas com complexidade de espaço linear $O(m+n)$. O algoritmo recursivamente divide a tabela em quatro quadrantes e ignora os quadrantes superior-direito e inferior-esquerdo, visto que a solução não passa por eles. Após, o algoritmo é chamado recursivamente nos quadrantes superior-esquerdo e inferior-direito. Em cada chamada recursiva é criada uma lista com as operações executadas, e tal lista é concatenada ao final das duas chamadas recursivas. A recuperação da seqüências de operações pode ser feita percorrendo-se linearmente esta lista.

Print-SCML

PRINT-SCML

Entrada Matriz $b \in \{\leftarrow, \nwarrow, \uparrow\}^{m \times n}$.

Saída A maior subseqüência Z comum entre X e Y obtida a partir de b.

```
1  if i = 0 or j = 0 then return
2  if b[i,j] =↖ then
```

```
3     Print-SCML(b, X, i 1, j 1)
4     print x_i
5   else if b[i,j] =↑ then
6     Print-SCML(b, X, i 1, j)
7   else
8     Print-SCML(b, X, i, j 1)
```

5.2.2. Similaridade entre strings

Considere o problema de determinar o número mínimo de operações que transformam um string s em um string t, se as operações permitidas são a inserção de um caracter, a deleção de um caracter ou a substituição de um caracter para um outro. O problema pode ser visto como um alinhamento de dois strings da forma

```
sonhar
vo--ar
```

em que cada coluna com um caracter diferente (inclusive a "falta" de um caracter `-`) tem custo 1 (uma coluna $(a, -)$ corresponde à uma deleção no primeiro ou uma inserção no segundo string, etc.). Esse problema tem subestrutura ótima: Uma solução ótima contém uma solução ótima do subproblema sem a última coluna, senão podemos obter uma solução de menor custo. Existem quatro casos possíveis para a última coluna:

$$\begin{bmatrix} a \\ - \end{bmatrix}; \begin{bmatrix} - \\ a \end{bmatrix}; \begin{bmatrix} a \\ a \end{bmatrix}; \begin{bmatrix} a \\ b \end{bmatrix}$$

com caracteres a, b diferentes. O caso (a, a) somente se aplica, se os últimos caracteres são iguais, o caso (a, b) somente, se eles são diferentes. Portanto, considerando todos casos possíveis fornece uma solução ótima:

$$d(s,t) = \begin{cases} \max(|s|, |t|) & \text{se } |s \models 0 \text{ ou } |t \models 0 \\ \min(d(s',t)+1, d(s,t')+1, d(s',t')+[c_1 \neq c_2]) & \text{se } s = s'c_1 \text{ e } t = t'c_2 \end{cases}$$

Essa distância está conhecida como *distância de Levenshtein* [20]. Uma implementação direta é

Distância entre textos

DISTÂNCIA

Entrada Dois strings s, t e seus respectivos tamanhos n e m medidos em número de caracteres.

Saída A distância mínima entre s e t.

```
1  distância(s,t,n,m):=
2    if (n=0) return m
3    if (m=0) return n
4    if (s_n = t_m) then
5      sol_0 = distância(s,t,n 1,m 1)
6    else
7      sol_0 = distância(s,t,n 1,m 1)+1
8    end if
9    sol_1 = distância(s,t,n,m 1) + 1
10   sol_2 = distância(s,t,n 1,m) + 1
11   return min(sol_0, sol_1, sol_2)
```

Essa implementação tem complexidade exponencial. Com programação dinâmica, armazenando os valores intermediários de d em uma matriz m, obtemos

Distância entre textos

PD-DISTÂNCIA

Entrada Dois strings s e t, e n e m, seus respectivos tamanhos medidos em número de caracteres.

Saída A distância mínima entre s e t.

Comentário O algritmo usa uma matriz $M = (m_{i,j}) \in \mathbb{N}^{(n+1)\times(m+1)}$ que armazena as distâncias mínimas $m_{i,j}$ entre os prefixos $s[1 \ldots i]$ e $t[1 \ldots j]$.

```
1  PD-distância(s,t,n,m):=
2    for i := 1,...,n do m_{i,0} := i
3    for i := 1,...,m do m_{0,i} := i
```

```
4     for i := 1, ..., n do
5       for j := 1, ..., m do
6         if (s_i = t_j) then
7           sol_0 := m_{i-1,j-1}
8         else
9           sol_0 := m_{i-1,j-1} + 1
10        end if
11        sol_1 := m_{i,j-1} + 1
12        sol_2 := m_{i-1,j} + 1
13        m_{i,j} := min(sol_0, sol_1, sol_2);
14      end for
15    return m_{i,j}
```

Distância entre textos

Valores armazenados na matriz M para o cálculo da distância entre ALTO e LOIROS

	·	L	O	I	R	O	S
·	0	1	2	3	4	5	6
A	1	1	2	3	4	5	6
L	2	1	2	3	4	5	6
T	3	2	2	3	4	5	6
O	4	3	3	3	4	4	5

```
-ALTO-
LOIROS
```

Distância entre textos

PD-DISTÂNCIA

Entrada Dois strings s e t, e n e m, seus respectivos tamanhos medidos em número de caracteres.

Saída A distância mínima entre s e t e uma matriz $P = (p_{i,j})$ que armazena a seqüência de operações.

Comentário O algoritmo usa uma matriz $M = (m_{i,j}) \in \mathbb{N}^{(n+1)\times(m+1)}$ que armazena as distâncias mínimas $m_{i,j}$ entre os prefixos $s[1 \ldots i]$ e $t[1 \ldots j]$.

```
1  PD distância(s,t,n,m):=
2    for i := 1,...,n do m_{i,0} = i; p_{i,0} := −1
3    for i := 1,...,m do m_{0,i} = i; p_{0,i} := −1
4    for i := 1,...,n do
5      for j := 1,...,m do
6        if (s_i = t_j) then
7          sol_0 = m_{i−1,j−1}
8        else
9          sol_0 = m_{i−1,j−1} + 1
10       end if
11       sol_1 := m_{i,j−1} + 1
12       sol_2 := m_{i−1,j} + 1
13       m_{i,j} := min(sol_0, sol_1, sol_2);
14       p_{i,j} := min{i|sol_i = m_{ij}}
15     end for
16   return m_{i,j}
```

Reconstrução da Seqüência de Operações

PD-operações

Entrada Uma matriz $P = (p_{ij})$ de tamanho $n \times m$ com marcação de operações, strings s, t, posições i e j.

Saída Uma seqüência a_1, a_2, a_x de operações executadas.

```
1  PD operações(P,s,t,i,j):=
2    case
3      p_{i,j} = −1:
4        return
5      p_{i,j} = 0:
6        PD−operações(s,t,i−1,j−1)
7        print ('M')
8      p_{i,j} = 1:
```

```
 9          PD-operações(s,t,i,j-1)
10          print ('I')
11        p_{i,j} = 2:
12          PD-operações(s,t,i-1,j)
13          print ('D')
14      end case
```

O algoritmo possui complexidade de tempo e espaço $O(mn)$, sendo que o espaço são duas matrizes P e M. O espaço pode ser reduzido para $O(\min\{n, m\})$ usando uma adaptação do algoritmo de Hirschberg.

5.3. Mochila máxima

Mochila máxima (ingl. maximum Knapsack)

Instância Um n itens com valores v_i e peso w_i, $1 \leq i \leq n$ e um limite de peso da mochila W.

Solução Um subconjunto $S \subseteq [1, n]$ que cabe na mochila, i.e. $\sum_{i \in S} w_i \leq W$.

Objetivo Maximizar o valor total $\sum_{i \in S} v_i$ dos itens selecionados.

Idéia: Ou item i faz parte da solução ótima com itens $i \ldots n$ ou não.

- Caso sim: temos um valor v_i acima da solução ótima para itens $i + 1, \ldots, n$ com capacidade restante $W - w_i$.
- Caso não: temos um valor correspondente ao solução ótima para itens $i]1, \ldots, n$ com capacidade W.

Seja $M(i, w)$ o valor da solução máxima para itens em $[i, n]$ e capacidade W. A idéia acima define uma recorrência

$$M(i,w) = \begin{cases} 0 & \text{se } i > n \text{ ou } w = 0 \\ M(i+1,w) & \text{se } w_i > w \text{ não cabe} \\ \max\{M(i+1,w), M(i+1,w-w_i)+v_i\} & \text{se } w_i \leq w \end{cases}$$

A solução desejada é $M(n, W)$. Para determinar a seleção de itens:

Mochila máxima (Knapsack)

- Seja $S^*(k, v)$ a solução de tamanho menor entre todas soluções que
  - usam somente itens $S \subseteq [1, k]$ e
  - tem valor exatamente v.
- Temos

$$\begin{aligned} S^*(k, 0) &= \emptyset \\ S^*(1, v_1) &= \{1\} \\ S^*(1, v) &= \text{undef} \qquad \text{para } v \neq v_1 \end{aligned}$$

Mochila máxima (Knapsack)

- S^* obedece a recorrência

$$S^*(k, v) = \min_{\text{tamanho}} \begin{cases} S^*(k-1, v-v_k) \cup \{k\} & \text{se } v_k \leq v \text{ e } S^*(k-1, v-v_k) \text{ definido} \\ S^*(k-1, v) \end{cases}$$

- Menor tamanho entre os dois

$$\sum_{i \in S^*(k-1, v-v_k)} t_i + t_k \leq \sum_{i \in S^*(k-1, v)} t_i.$$

- Melhor valor: Escolhe $S^*(n, v)$ com o valor máximo de v definido.
- Tempo e espaço: $O(n \sum_i v_i)$.

5.4. Multiplicação de Cadeias de Matrizes

Qual é a melhor ordem para multiplicar n matrizes $M = M_1 \times \cdots \times M_n$? Como o produto de matrizes é associativo, temos várias possibilidades de chegar em M. Por exemplo, com quatro matrizes temos as cinco possibilidades

Possíveis multiplicações

Dadas (M_1, M_2, M_3, M_4) pode-se obter $M_1 \times M_2 \times M_3 \times M_4$ de 5 modos distintos, mas resultando no mesmo produto

$$M_1(M_2(M_3M_4))$$
$$M_1((M_2M_3)M_4)$$
$$(M_1M_2)(M_3M_4)$$
$$(M_1(M_2M_3))M_4$$
$$((M_1M_2)M_3)M_4$$

- Podemos multiplicar duas matrizes somente se $Ncol(A) = Nlin(B)$
- Sejam duas matrizes com dimensões $p \cdot q$ e $q \cdot r$ respectivamente. O número de multiplicações resultantes é $p \cdot q \cdot r$.

Dependendo do tamanho dos matrizes, um desses produtos tem o menor número de adições é multiplicações. O produto de duas matrizes $p \times q$ e $q \times r$ precisa prq multiplicações e $pr(q-1)$ adições. No exemplo acima, caso temos matrizes do tamanho 3×1, 1×4, 4×1 e 1×5 as ordens diferentes resultam em

Número de multiplicações para cada seqüência

$$20 + 20 + 15 = 55$$
$$4 + 5 + 15 = 24$$
$$12 + 20 + 60 = 92$$
$$4 + 5 + 15 = 24$$
$$12 + 12 + 15 = 39$$

operações, respectivamente. Logo, antes de multiplicar as matrizes vale a pena determinar a ordem ótima (caso o tempo para determinar ela não é proibitivo). Dada uma ordem, podemos computar o número de adições e multiplicações em tempo linear. Mas quantas ordens tem? O produto final consiste em duas matrizes que são os resultados dos produtos de i e $n-i$ matrizes; o ponto de separação i pode ser depois qualquer matriz $1 \leq i < n$. Por isso o número de possibilidades C_n satisfaz a recorrência

$$C_n = \sum_{1 \leq i < n} C_i C_{n-i}$$

para $n \geq 1$ e as condições $C_1 = 1$ e $C_2 = 1$. A solução dessa recorrência é $C_n = \binom{2n}{n}\frac{1}{2(2n-1)} = O(4^n/n^{3/2})$ e temos $C_n \geq 2^{n-2}$, logo têm um número exponencial de ordens de multiplicação possíveis[1].

Solução por Recorrência

O número de possibilidades T_n satisfaz a recorrência

$$T(n) = \sum_{1 \leq i < n-1} T(i) \cdot T(n-i)$$

para $n \geq 1$ e as condições $T(1) = 1$.
A solução dessa recorrência é $T(n) = \binom{2n}{n}\frac{1}{2(2n-1)} = O(4^n/n^{3/2})$ e temos $C_n \geq 2^{n-2}$
Então não vale a pena avaliar o melhor ordem de multiplicação, enfrentando um número exponencial de possibilidades? Não, existe uma solução com programação dinâmica, baseada na mesma observação que levou à nossa recorrência.

$$m_{ik} = \begin{cases} \min_{i \leq j < k} m_{ij} + m_{(j+1)k} + b_{i-1}b_j b_k & \text{caso } i < k \\ 0 & \text{caso } i = kx \end{cases}$$

Multiplicação de Cadeias de Matrizes

- Dada uma cadeia $(A_1, A_2, \ldots, A_n)$ de n matrizes, coloque o produto $A_1 A_2 \ldots A_n$ entre parênteses de forma a minimizar o número de multiplicações.

Algoritmo Multi-Mat-1

Retorna o número mínimo de multiplicações necessárias para multiplicar a cadeia de matrizes passada como parâmetro.

MULTI-MAT-1

Entrada Cadeia de matrizes $(A_1, A_2, \ldots, A_n)$ e suas respectivas dimensões b_i, $0 \leq i \leq n$. A matrix A_i tem dimensão $b_{i-1} \times b_i$.

Saída Número mínimo de multiplicações.

[1] Podemos obter uma solução usando funções geratrizes. $(C_{n-1})_{n \geq 1}$ são os números Catalan, que têm um número enorme de aplicações na combinatórica.

```
1  for i:=1 to n do m_{i,j} := 0
2  for u:=1 to n-1 do {diagonais superiores}
3    for i:=1 to n-u do {posição na diagonal}
4      j:=i+u          {u = j - i}
5      m_{i,j} := ∞
6      for k:=i to j-1 do
7        c:= m_{i,k} + m_{k+1,j} + b_{i-1} · b_k · b_j
8        if c < m_{i,j} then m_{i,j}:=c
9      end for
10   end for
11 end for
12 return m_{1,n}
```

Considerações para a Análise

- O tamanho da entrada se refere ao número de matrizes a serem multiplicadas
- As operações aritméticas sobre os naturais são consideradas operações fundamentais

Análise de Complexidade do Algoritmo

$$C_p = \sum_{u=1}^{n-1}\sum_{i=1}^{n-u}(1+\sum_{k-i}^{j-1}4) = \sum_{u=1}^{n-1}\sum_{i=1}^{n-u}(1+4u)$$
$$= \sum_{u=1}^{n-1}(n-u)\dot{(}1+4u) = \sum_{u=1}^{n-1} n+4nu-u-4u^2 = O(n^3)$$

Análise de Complexidade do Algoritmo

$$C_p[\text{Inicialização}] = O(n)$$
$$C_p[\text{Iteração}] = O(n^3)$$
$$C_p[\text{Finalização}] = O(1)$$
$$C_p[\text{Algoritmo}] = O(n) + O(n^3) + O(1) = O(n^3)$$

Algoritmo Multi-Mat-2

Retorna o número mínimo de multiplicações e a parentização respectiva para multiplicar a cadeia de matrizes passada como parâmetro.

MULTI-MAT-1

Entrada Cadeia de matrizes $(A_1, A_2, \ldots, A_n)$ e suas respectivas dimensões armazenadas no vetor b.

Saída Número mínimo de multiplicações.

```
1  for i:=1 to n do m_{i,j} := 0 {inicializa diagonal principal}
2  for d:=1 to n-1 do { para todas diagonais superiores}
3    for i:=1 to n-u do { para cada posição na diagonal}
4      j:=i+u          {u=j  i}
5      m_{i,j} := ∞
6      for k:=i to j do
7        c:= m_{i,k} + m_{k+1,j} + b_{i-1} · b_k · b_j
8        if c < m_{i,j} then
9             m_{i,j}:=c
10            P_{i,j} = k
11     end for
12   end for
13 end for
14 return m_{1,n}, p
```

Algoritmo Print-Parentização

PRINT-PARENTIZAÇÃO

Entrada Matriz P, índices i e j .

Saída Impressão da parentização entre os índices i e j.

```
1  if i = j then
2     print "A_i"
3  else
4     print "("
5     Print-Parentização(P, i, P_{i,j})
6     Print-Parentização(P, P_{i,j} + 1, j)
7     print ")"
8  end if
```

5.5. Tópicos

5.5.1. Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall calcula o caminho mínimo entre todos os pares de vértices de um grafo.

Algoritmo de Floyd-Warshall

- Conteúdo disponível na seção 25.2 do Cormen, e Exemplo 5.2.4 (Laira&Veloso, 2^a edição).
- Calcula o caminho mínimo entre cada par de vértices de um grafo.
- Considera que o grafo não tenha ciclos negativos, embora possa conter arcos de custo negativo.

Subestrutura ótima

Subcaminhos de caminhos mais curtos são caminhos mais curtos.

- Lema 24.1 (Cormen): Dado um grafo orientado ponderado $G = (V, E)$, com função peso $w : E \rightarrow R$, seja $p = (v_1, v_2, \ldots, v_k)$ um caminho mais curto do vértice v_1 até o vértice v_k e, para quaisquer i e j tais que $1 \leq i \leq j \leq k$, seja $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ o subcaminho p desde o vértice v_i até o vértice v_j. Então, p_{ij} é um caminho mais curto de v_i até v_j.

Algoritmo de Floyd-Warshall

ALGORITMO DE FLOYD-WARSHALL

Entrada Um grafo $G = (V, E)$ e uma matriz quadrada com cada célula contendo o peso do arco (i,j), ou ∞ caso o arco não existir

Saída Uma matriz quadrada com cada célula contendo a distância mínima entre i e j.

```
1  n := linhas(W)
2  for k := 1 to n
3      for i := 1 to n
```

```
4          for j := 1 to n
5              d_ij^k := min(d_ij^{k-1}, d_ik^{k-1} + d_kj^{k-1})
6  return(D^n)
```

Exemplo

5.5.2. Caixeiro viajante

O problema de caixeiro viajante é um exemplo em que a programação dinâmica ajuda reduzir um trabalho exponencial. Esperadamente, o algoritmo final ainda é exponencial (o problema é NP-completo), mas significadamente menor.

PROBLEMA DO CAIXEIRO VIAJANTE

Instância Um grafo G=(V,E) com pesos d (distâncias) atribuídos aos links. $V = [1, n]$ sem perda de generalidade.

Solução Uma rota que visita todos vértices exatamente uma vez, i.e. uma permutação de $[1, n]$.

Objetivo Minimizar o custo da rota $\sum_{1 \leq i < n} d_{\{i,i+1\}} + d_{\{n,1\}}$.

O algoritmo é baseado na seguinte idéia (proposta por Bellman em 1962 [4]). Seja $v_1, v_2, \ldots$ uma solução ótima. Sem perda de generalidade, podemos supor que $v_1 = 1$. Essa solução tem como sub-solução ótima o caminho $v_2, v_3, \ldots$ que passa por todos vértices exceto v_1 e volta. Da mesmo forma a última sub-solução tem o caminho $v_3, v_4, \ldots$ que passa por todos vértices exceto v_1, v_2 e volta, como sub-solução. Essa soluções têm sub-estrutura ótima, porque qualquer outro caminho menor pode ser substituído para o caminho atual.

Logo, podemos definir $T(i, V)$ como menor rota começando no vértice i e passando por todos vértices em V exatamente uma vez e volta para vértice 1. A solução desejada então é $T(1, [2, n])$. Para determinar o valor de $T(i, V)$ temos que minizar sobre todas continuações possíveis. Isso leva à recorrência

$$T(i,V) = \begin{cases} \min_{v \in V} d_{iv} + T(v, V \setminus \{v\}) & V \neq \emptyset \\ d_{i1} & \text{caso } V = \emptyset \end{cases}$$

Se ordenamos todos os sub-conjuntos dos vértices $[1,n]$ em ordem de $\subseteq$, obtemos uma matrix de dependências

	V_1	V_2	$\cdots$	V_{2^n}
1				
2				
$\vdots$				
n				

em que qualquer elemento depende somente de elementos em colunas mais para esquerda. Uma implementação pode representar uma subconjunto de $[1,n]$ como número entre 0 e $2^n - 1$. Nessa caso, a ordem natural já respeita a ordem $\subseteq$ entre os conjuntos, e podemos substituir um teste $v \in V_j$ com $2^v \& j = 2^v$ e a operação $V_j \setminus \{v\}$ com $j - 2^v$.

```
1  for i ∈ [1,n] do T_{i,0} := d_{i1} { base }
2  for j ∈ [1,2^n − 1] do
3    for i ∈ [1,n] do
4      T_{i,j} := min_{2^k & j = 2^k} d_{ik} + T_{i,j−2^k} { tempo O(n) ! }
5    end for
6  end for
```

A complexidade de tempo desse algoritmo é $n^2 2^n$ porque a minimzação na linha 4 precisa $O(n)$ operações. A complexidade do espação é $O(n2^n)$. Essa é atualmente o melhor algoritmo exato conhecido para o problema do caixeiro viajante (veja também xkcd.com/399).

5.5.3. Árvore de busca binária ótima

Motivação para o Problema

- Suponha que temos um conjunto de chaves com probabilidades de busca conhecidas.
- Caso a busca é repetida muitas vezes, vela a pena construir uma estrutura de dados que minimiza o tempo médio para encontrar uma chave.
- Uma estrutura de busca eficiente é um árvore binária.

Portanto, vamos investigar como construir o árvore binária ótimo. Para um conjunto de chaves com distribuição de busca conhecida, queremos minimizar o número médio de comparações (nossa medida de custos).

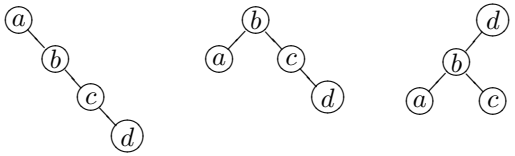
Exemplo 5.1
Considere a seqüência ordenada $a < b < c < d$ e as probabilidades

Exemplo

Elemento	a	b	c	d
Pr	0.2	0.1	0.6	0.1

qual seria uma árvore ótima? Alguns exemplos

Árvore correspondente



com um número médio de comparações $0.2\times1+0.1\times2+0.6\times3+0.1\times4 = 2.6$, $0.2\times2+0.1\times1+0.6\times2+0.1\times3 = 2.0$, $0.2\times3+0.1\times2+0.6\times3+0.1\times1 = 2.7$, respectivamente. ◊

Árvore de Busca Binária Ótima
Em geral, temos que considerar não somente as probabilidades que procure-se um dado chave, mas também as probabilidades que um chave procurada não pertence à árvore. Logo supomos que temos

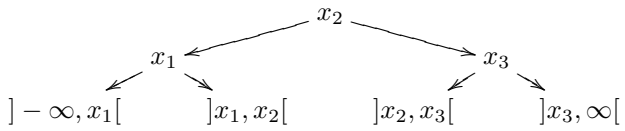
1. uma seqüência ordenada $a_1 < a_2 < \cdots < a_n$ de n chaves e
2. probabilidades

$$\Pr[c < a_1], \Pr[c = a_1], \Pr[a_1 < c < a_2], \ldots$$
$$\ldots, \Pr[a_{n-1} < c < a_n], \Pr[c = a_n], \Pr[a_n < c]$$

que a chave procurado c é uma das chaves da seqüência ou cai num intervalo entre elas.

A partir dessas informações queremos minimizar a complexidade média da busca. Em uma dada árvore, podemos observar que o número de comparações para achar uma chave existente é igual a profundidade dela na árvore (começando com profundidade 1 na raiz). Caso a chave não pertence à árvore,

podemos imaginar chaves artificiais que representam os intervalos entre as chaves, e o número de comparações necessárias é um menos que a profundidade de uma chave artificial. Um exemplo de um árvore com chaves artificiais (representado pelos intervalos correspondentes) é



Para facilitar o tratamento, vamos introduzir chaves adicionais $a_0 = -\infty$ e $a_{n+1} = \infty$. Com isso, obtemos a complexidade média de busca

$$c_M = \sum_{1\le i\le n} \Pr[c = a_i]\mathrm{prof}(a_i) + \sum_{0\le i\le n} \Pr[a_i < c < a_{i+1}](\mathrm{prof}(]a_i, a_{i+1}[) - 1);$$

ela depende da árvore concreta.

Como achar a árvore ótima? A observação crucial é a seguinte: *Uma das chaves deve ser a raiz e a duas sub-árvores da esquerda e da direita devem ser árvores ótimas pelas sub-seqüências correspondentes.*

Para expressar essa observação numa equação, vamos denotar com $c_M(e, d)$ a complexidade ponderada de uma busca numa sub-árvore ótima sobre os elementos $a_e, \ldots, a_d$. Para a complexidade da árvore inteira, definido acima, temos $c_M = c_M(1, n)$. Da mesmo forma, obtemos

$$c_M(e,d) = \sum_{e\le i\le d} \Pr[c = a_i]\mathrm{prof}(a_i) + \sum_{e-1\le i\le d} \Pr[a_i < c < a_{i+1}](\mathrm{prof}(]a_i, a_{i+1}[) - 1)$$

Árvore de Busca Binária Ótima

Supondo que a_r é a raiz desse sub-árvore, essa complexidade pode ser escrito como

$$\begin{aligned}
c_M(e,d) &= \Pr[c = a_r] \\
&\quad + \sum_{e\le i<r} \Pr[c = a_i]\mathrm{prof}(a_i) + \sum_{e-1\le i<r} \Pr[a_i < c < a_{i+1}](\mathrm{prof}(]a_i, a_{i+1}[) - 1) \\
&\quad + \sum_{r<i\le d} \Pr[c = a_i]\mathrm{prof}(a_i) + \sum_{r\le i\le d} \Pr[a_i < c < a_{i+1}](\mathrm{prof}(]a_i, a_{i+1}[) - 1) \\
&= \left(\sum_{e-1\le i\le d} \Pr[a_i < c < a_{i+1}] + \sum_{e\le i\le d} \Pr[c = a_i] \right) \\
&\quad + c_M(e, r-1) + c_M(r+1, d) \\
&= \Pr[a_{e-1} < c < a_{d+1}] + c_M(e, r-1) + c_M(r+1, d)
\end{aligned}$$

Árvore de Busca Binária Ótima

(O penúltimo passo é justificado porque, passando para uma sub-árvore a profundidade e um a menos.) Com essa equação podemos construir uma recorrência para a complexidade média ótima: Escolhe sempre a raiz que minimiza essa soma. Como base temos complexidade 0 se $d > e$:

$$c_M(e,d) = \begin{cases} \min_{e \le r \le d} \Pr[a_{e-1} < c < a_{d+1}] + c_M(e, r-1) + c_M(r+1, d) & \text{caso } e \le d \\ 0 & \text{caso } e > d \end{cases} \tag{5.1}$$

Árvore de Busca Binária Ótima

Ao invés de calcular o valor c_M recursivamente, vamos usar a programação (tabelação) dinâmica com três tabelas:

- c_{ij}: complexidade média da árvore ótima para as chaves a_i até a_j
- r_{ij}: raiz da árvore ótima para as chaves a_i até a_j
- p_{ij}: $\Pr[a_{i-1} < c < a_{j+1}]$

Árvore de Busca Binária Ótima

ABB-ÓTIMA

Entrada Probabilidades $p_i = \Pr[c = a_i]$ e $q_i = \Pr[a_i < c < a_{i+1}]$.

Saída Vetores c e r como descrita acima.

```
1   for i:=1 to n+1 do
2     p_{i(i-1)}:=q_{i-1}
3     c_{i(i-1)}:=0
4   end for
5
6   for d:=0 to n-1 do { para todas diagonais }
7     for i:=1 to n-d do { da chave i }
8       j:=d+i          { até chave j }
9       p_{ij}:=p_{i(j-1)} + p_j + q_j
10      c_{ij}:=∞
11      for r:=i to j do
12        c:= p_{ij} + c_{i(r-1)} + c_{(r+1)j}
13        if c < c_{ij} then
```

```
14              c_ij:=c
15              r_ij:=r
16          end for
17      end for
18  end for
```

i/j	1	2	3	$\cdots$	n
1					
2					
·					
·					
n					

Finalmente, queremos analisar a complexidade desse algoritmo. São três laços, cada com não mais que n iterações e com trabalho constante no corpo. Logo a complexidade pessimista é $O(n^3)$.

6. Divisão e conquista

6.1. Introdução

Método de Divisão e Conquista

- Dividir o problema original em um determinado número de subproblemas independentes
- Conquistar os subproblemas, resolvendo-os recursivamente até obter o caso base.
- Combinar as soluções dadas aos subproblemas, a fim de formar a solução do problema original.

Recorrências

- O tempo de execução dos algoritmos recursivos pode ser descrito por uma recorrência.
- Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores.

Divisão e conquista

DC

Entrada Uma instância I de tamanho n.

```
1 if n = 1 then
2   return Solução direta
3 else
4   Divide I em sub instâncias I_1,...,I_k, k > 0
5     com tamanhos n_i < n.
6   Resolve recursivamente: I_1,...,I_k.
7   Resolve I usando sub soluções DC(I_1),...,DC(I_k).
8 end if
```

Recursão natural

- Seja $d(n)$ o tempo para a divisão.
- Seja $s(n)$ o tempo para computar a solução final.
- Podemos somar: $f(n) = d(n) + s(n)$ e obtemos

$$T(n) = \begin{cases} \Theta(1) & \text{para } n < n_0 \\ \sum_{1 \leq i \leq k} T(n_i) + f(n) & \text{caso contrário.} \end{cases}$$

Recursão natural: caso balanceado

$$T(n) = \begin{cases} \Theta(1) & \text{para } n < n_0 \\ kT(\lceil n/m \rceil) + f(n) & \text{caso contrário.} \end{cases}$$

Mergesort

MERGESORT

Entrada Índices p, r e um vetor A com elementos $A_p, \ldots, A_r$

Saída A com elementos em ordem não-decrescente, i.e. para $i < j$ temos $A_i \leq A_j$.

```
1  if n = 1 then
2    return Solução direta
3  else
4    Divide I em sub instâncias I_1,...,I_k, k > 0
5      com tamanhos n_i < n.
6    Resolve recursivamente: I_1,...,I_k.
7    Resolve I usando sub soluções DC(I_1),...,DC(I_k).
8  end if
```

Recorrências simplificadas

Formalmente, a equação de recorrência do Mergesort é

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Em vez de

$$T(n) = 2T(n/2) + \Theta(n)$$

Para simplificar a equação, sem afetar a análise de complexidade correspondente, em geral:

- supõe-se argumentos inteiros para funções (omitindo pisos e tetos)
- omite-se a condição limite da recorrência

Recorrências: caso do Mergesort
A equação de recorrência do Mergesort é

$$T(n) = 2T(n/2) + \Theta(n)$$

Sendo que:

- T(n) representa o tempo da chamada recursiva da função para um problema de tamanho n.
- $2T(\frac{n}{2})$ indica que, a cada iteração, duas chamadas recursivas (2T) serão executadas para entradas de tamanho $\frac{n}{2}$.
- Os resultados das duas chamadas recursivas serão combinados (*merged*) com um algoritmo com complexidade de pior caso $\Theta(n)$.

6.2. Resolver recorrências

Métodos para resolver recorrências
Existem vários métodos para resolver recorrências:

- Método da substituição
- Método de árvore de recursão
- Método mestre

6.2.1. Método da substituição

Método da substituição

- O método da substituição envolve duas etapas:
  1. pressupõe-se um limite hipotético.
  2. usa-se indução matemática para provar que a suposição está correta.
- Aplica-se este método em casos que é fácil pressupor a forma de resposta.
- Pode ser usado para estabelecer limites superiores ou inferiores.

Mergesort usando o método da substituição

Supõe-se que a recorrência

$$T(n) = 2T(\left\lfloor \frac{n}{2} \right\rfloor) + n$$

tem limite superior igual a $n \log n$, ou seja, $T(n) = O(n \log n)$. Devemos provar que $T(n) \leq c \cdot n \log n$ para uma escolha apropriada da constante $c > 0$.

$$\begin{aligned} T(n) &\leq 2(c \left\lfloor \frac{n}{2} \right\rfloor \log(\left\lfloor \frac{n}{2} \right\rfloor) + n) \\ &\leq cn \log \frac{n}{2} + n = cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

para $c \geq 1$.
A expressão na equação

$$c \cdot n \log n \underbrace{-c \cdot n + n}_{\text{residuo}}$$

se chama *resíduo*. O objetivo na prova é mostrar, que o resíduo é negativo.

Como fazer um bom palpite

- Usar o resultado de recorrências semelhantes. Por exemplo, considerando a recorrência $T(n) = 2T(\left\lfloor \frac{n}{2} \right\rfloor + 17) + n$, tem-se que $T(n) \in O(n \log n)$.

- Provar limites superiores (ou inferiores) e reduzir o intervalo de incerteza. Por exemplo, para equação do Mergesort podemos provar que $T(n) = \Omega(n)$ e $T(n) = O(n^2)$. Podemos gradualmente diminuir o limite superior e elevar o inferior, até obter $T(n) = \Theta(n \log n)$.

- Usa-se o resultado do método de árvore de recursão como limite hipotético para o método da substituição.

Exemplo 6.1
Vamos procurar o máximo de uma seqüência por divisão é conquista:

MÁXIMO

Entrada Uma seqüência a e dois índices l, r tal que $a_l, \ldots, a_{r-1}$ é definido.

Saída $\max_{l \leq i < r} a_i$

```
1  m_1 := M(a, l, ⌊(l + r)/2⌋
2  m_2 := M(a, ⌊(l + r)/2, r⌋
```

Isso leva a recorrência

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

$\Diamond$

Algumas sutilezas nas resoluções

- Considere a recorrência $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$. Prove que $T(n) = O(n)$

- Considere a recorrência $T(n) = 2T(\lfloor n/2 \rfloor) + n$. É possível provar que $T(n) = O(n)$?

- Considere a recorrência $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$. Prove que $T(n) = O(\log n \log \log n)$

Proposta de exercícios: 4.1-1, 4.1-2, 4.1-5 e 4.1-6 do Cormen (pág. 54 da edição traduzida).

Substituição direta

- Supomos que a recorrência tem a forma

$$T(n) = \begin{cases} f(1) & \text{caso } n = 1 \\ T(n-1) + f(n) & \text{caso contrário} \end{cases}$$

- Para resolver ele podemos substituir

$$T(n) = f(n) + T(n-1) = f(n) + f(n-1) + T(n-1) + f(n-2) \\ = \cdots = \sum_{1 \le i \le n} f(i)$$

(Os $\cdots$ substituem uma prova por indução.)

É simples de generalizar isso para

$$\begin{aligned} T(n) &= T(n/c) + f(n) \\ &= f(n) + f(n/c) + f(n/c^2) + \cdots \\ &= \sum_{0 \le i \le log_c n} f(n/c^i) \end{aligned}$$

Exemplo 6.2
Na aula sobre a complexidade média do algoritmo Quicksort (veja página 59), encontramos uma recorrência da forma

$$T(n) = n + T(n-1)$$

cuja solução é $\sum_{1 \le i \le n} i = n(n-1)/2$. ◊

Substituição direta

- Da mesma forma podemos resolver recorrências como

$$T(n) = \begin{cases} f(1) & \text{caso } n = 1 \\ T(n/2) + f(n) & \text{caso contrário} \end{cases}$$

- substituindo

$$T(n) = f(n) + T(n/2) = f(n) + f(n/2) + T(n/4) \\ = \cdots = \sum_{0 \le i \le \log_2 n} f(n/2^i)$$

Exemplo 6.3
Ainda na aula sobre a complexidade média do algoritmo Quicksort (veja página 59), encontramos outra recorrência da forma

$$T(n) = n + 2T(n/2).$$

Para colocar ela na forma acima, vamos dividir primeiro por n para obter

$$T(n)/n = 1 + T(n/2)/(n/2)$$

e depois substituir $A(n) = T(n)/n$, que leva à recorrência

$$A(n) = 1 + A(n/2)$$

cuja solução é $\sum_{0 \le i \le \log_2 n} 1 = \log_2 n$. Portanto temos a solução $T(n) = n \log_2 n$.
Observe que a análise não considera constantes: qualquer função $cn \log_2 n$ também satisfaz a recorrência. A solução exata é determinada pela base; uma alternativa é concluir que $T(n) = \Theta(n \log_2 n)$. ◊

Também podemos generalizar isso. Para

$$f(n) = 2f(n/2)$$

é simples de ver que

$$f(n) = 2f(n/2) = 2^2 f(n/4) = \cdots = 2^{\log_2 n} n = n$$

(observe que toda função $f(n) = cn$ satisfaz a recorrência).
Generalizando obtemos

$$f(n) = cf(n/2) = \cdots = c^{\log_2 n} = n^{log_2 c}$$
$$f(n) = c_1 f(n/c_2) = \cdots = c_1^{\log_{c_2} n} = n^{log_{c_2} c_1}$$

O caso mais complicado é o caso combinado

$$T(n) = c_1 T(n/c_2) + f(n).$$

O nosso objetivo é nos livrar da constante c_1. Se dividimos por $c_1^{\log_{c_2} n}$ obtemos

$$\frac{T(n)}{c_1^{\log_{c_2} n}} = \frac{T(n/c_2)}{c_1^{\log_{c_2} n/c_2}} + \frac{f(n)}{c_1^{\log_{c_2} n}}$$

e substituindo $A(n) = T(n)/c_1^{\log_{c_2} n}$ temos

$$A(n) = A(n/c_2) + \frac{f(n)}{c_1^{\log_{c_2} n}}$$

uma forma que sabemos resolver:

$$A(n) = \sum_{0 \le i \le \log_{c_2} n} \frac{f(n/c_2^i)}{c_1^{\log_{c_2} n / c_2^i}} = \frac{1}{c_1^{\log_{c_2} n}} \sum_{0 \le i \le \log_{c_2} n} f(n/c_2^i) c_1^i$$

Após de resolver essa recorrência, podemos re-substituir para obter a solução de $T(n)$.

Exemplo 6.4
Multiplicação de números inteiros A multiplicação de números binários (veja exercício 6.3) gera a recorrência

$$T(n) = 3T(n/2) + cn$$

Dividindo por $3^{\log_2 n}$ obtemos

$$\frac{T(n)}{3^{\log_2 n}} = \frac{T(n/2)}{3^{\log_2 n/2}} + \frac{cn}{3^{\log_2 n}}$$

e substituindo $A(n) = T(n)/3^{\log_2 n}$ temos

$$\begin{aligned} A(n) &= A(n/2) + \frac{cn}{3^{\log_2 n}} \\ &= c \sum_{0 \le i \le \log_2 n} \frac{n/2^i}{3^{\log_2 n/2^i}} \\ &= \frac{cn}{3^{\log_2 n}} \sum_{0 \le i \le \log_2 n} \left(\frac{3}{2}\right)^i \\ &= \frac{cn}{3^{\log_2 n}} \frac{(3/2)^{\log_2 n + 1}}{1/2} \\ &= 3c \end{aligned}$$

e portanto

$$T(n) = A(n) 3^{\log_2 n} = 3cn^{\log_2 3} = \Theta(n^{1.58})$$

◊

6.2.2. Método da árvore de recursão

O método da árvore de recursão

Uma árvore de recursão apresenta uma forma bem intuitiva para a análise de complexidade de algoritmos recursivos.

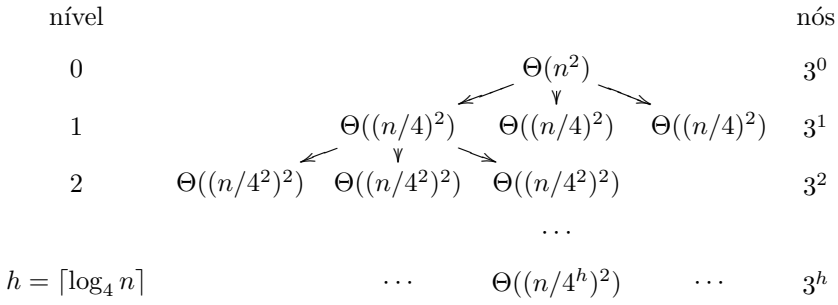
- Numa árvore de recursão cada nó representa o custo de um único subproblema da respectiva chamada recursiva
- Somam-se os custos de todos os nós de um mesmo nível, para obter o custo daquele nível
- Somam-se os custos de todos os níveis para obter o custo da árvore

Exemplo

Dada a recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

- Em que nível da árvore o tamanho do problema é 1? No nível $i = \log_4 n = \frac{\log_2 n}{2}$.
- Quantos níveis tem a árvore? A árvore tem $\log_4 n + 1$ níveis (0, 1, 2, 3, ..., $\log_4 n$).
- Quantos nós têm cada nível? 3^i.
- Qual o tamanho do problema em cada nível? $\frac{n}{4^i}$.
- Qual o custo de cada nível i da árvore? $3^i c(\frac{n}{4^i})^2$.
- Quantos nós tem o último nível? $\Theta(n^{\log_4 3})$.
- Qual o custo da árvore? $\sum_{i=0}^{\log_4(n)} n^2 \cdot (3/16)^i = O(n^2)$.

Exemplo



Prova por substituição usando o resultado da árvore de recorrência

- O limite de $O(n^2)$ deve ser um limite restrito, pois a primeira chamada recursiva é $\Theta(n^2)$.
- Prove por indução que $T(n) = \Theta(n^2)$

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\leq d \cdot n^2 \end{aligned}$$

para $(\frac{3d}{16} + c) \leq d$, ou seja, para valores de d tais que $d \geq \frac{16}{13} c$

Exemplo 6.5
Outro exemplo é a recorrência $T(n) = 3T(n/2) + cn$ da multiplicação de números binários. Temos $\log_2 n$ níveis na árvore, o nível i com 3^i nós, tamanho do problema $n/2^i$, trabalho $cn/2^i$ por nó e portanto $(3/2)^i n$ trabalho total por nível. O número de folhas é $3^{\log_2 n}$ e portanto temos

$$\begin{aligned} T(n) &= \sum_{0 \leq i < \log_2 n} (3/2)^i n + \Theta(3^{\log_2 n}) \\ &= n \left(\frac{(3/2)^{\log_2 n} - 1}{3/2 - 1} \right) + \Theta(3^{\log_2 n}) \\ &= 2(n^{\log_2 3} - 1) + \Theta(n^{\log_2 3}) \\ &= \Theta(n^{\log_2 3}) \end{aligned}$$

Observe que a recorrência $T(n) = 3T(n/2) + c$ tem a mesma solução. ◊

Resumindo o método

1. Desenha a árvore de recursão
2. Determina
   - o número de níveis
   - o número de nós e o custo por nível

- o número de folhas

3. Soma os custos dos níveis e o custo das folhas
4. (Eventualmente) Verifica por substituição

Árvore de recorrência: ramos desiguais
Calcule a complexidade de um algoritmo com a seguinte equação de recorrência

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

Proposta de exercícios: 4.2-1, 4.2-2 e 4.2-3 do Cormen [7].

6.2.3. Método Mestre

Método Mestre
Para aplicar o método mestre deve ter a recorrência na seguinte forma:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

onde:

- $a \geq 1$
- $b > 1$
- $f(n)$ é uma função assintoticamente positiva.

Se a recorrência estiver no formato acima, então $T(n)$ é limitada assintoticamente como:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$

Considerações

- Nos casos 1 e 3 $f(n)$ deve ser polinomialmente menor, resp. maior que $n^{\log_b a}$, ou seja, $f(n)$ difere assintoticamente por um fator n^ϵ para um $\epsilon > 0$.
- Os três casos não abrangem todas as possibilidades

Proposta de exercícios: 6.1 e 6.2.

Algoritmo Potenciação

POTENCIAÇÃO-TRIVIAL (PT)

Entrada Uma base $a \in \mathbb{R}$ e um exponente $n \in \mathbb{N}$.

Saída A potência a^n.

```
1  if n = 0
2    return 1
3  else
4    return PT(a, n − 1) × a
5  end if
```

Complexidade da potenciação

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + 1 & \text{se } n > 0 \end{cases}$$

A complexidade dessa recorrência é linear, ou seja, $T(n) \in O(n)$

Algoritmo Potenciação para $n = 2^i$

POTENCIAÇÃO-NPOTÊNCIA2 (P2)

Entrada Uma base $a \in \mathbb{R}$ e um exponente $n \in \mathbb{N}$.

Saída A potência a^n.

```
1  if n = 1 then
2    return a
3  else
4      x := P2(a , n ÷ 2)
5      return x × x
6  end if
```

Complexidade da potenciação-Npotência2

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c & \text{se } n > 0 \end{cases}$$

A complexidade dessa recorrência é logarítmica, ou seja, $T(n) \in O(\log n)$

Busca Binária

BUSCA-BINÁRIA(I,F,X,S)

Entrada Um inteiro x, índices i e f e uma seqüência $S = a_1, a_2, \ldots a_n$ de números ordenados.

Saída Posição i em que x se encontra na seqüência S ou ∞ caso $x \notin S$.

```
1  if i = f then
2    if a_i = x return i
3    else return ∞
4  end if
5  m := ⌊(f−i)/2⌋ + i
6  if x < a_m then
7    return Busca Binária(i, m−1)
8  else
9    return Busca Binária(m+1, f)
10 end if
```

Complexidade da Busca-Binária

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + c & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é logarítmica, ou seja, $T(n) \in O(\log n)$

Quicksort

QUICKSORT

Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída a com os elementos em ordem não-decrescente, i.e. para $i < j$ temos $a_i \leq a_j$.

```
1  if l < r then
2    m := Partition(l,r,a);
3    Quicksort(l,m-1,a);
4    Quicksort(m+1,r,a);
5  end if
```

Complexidade do Quicksort no pior caso

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(n-1) + \Theta(n) & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é quadrática, ou seja, $T(n) \in O(n^2)$

Complexidade do Quicksort no melhor caso

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é $T(n) \in O(n \log n)$

Complexidade do Quicksort com Particionamento Balanceado

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\frac{9n}{10}) + T(\frac{n}{10}) + \Theta(n) & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é $T(n) \in O(n \log n)$

Prova do Teorema Master

- Consideremos o caso simplificado em que $n = 1, b, b^2, ...$, ou seja, n é uma potência exata de dois e assim não precisamos nos preocupar com tetos e pisos.

Prova do Teorema Master
Lema 4.2: Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Defina $T(n)$ sobre potências exatas de b pela recorrência:

$$\begin{cases} T(n) = \Theta(1) & \text{se } n = 1 \\ T(n) = aT(\frac{n}{b}) + f(n) & \text{se } n = b^i \end{cases}$$

Onde i é um inteiro positivo. Então

$$T(n) = \Theta(n^{log_b a}) + \sum_{j=0}^{\log_b(n-1)} a^j \cdot f(n/b^j)$$

Prova do Teorema Master
Lema 4.4: Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Uma função $g(n)$ definida sobre potências exatas de b por

$$g(n) = \sum_{j=0}^{\log_b(n-1)} a^j \cdot f(n/b^j)$$

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0 \in \mathbb{R}^+$, então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \cdot lgn)$
3. Se $a \cdot f(n/b) \leq c \cdot f(n)$ para $c < 1$ e para todo $n \geq b$, então $g(n) = \Theta(f(n))$

Prova do Teorema Master
Lema 4.4: Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Defina $T(n)$ sobre potências exatas de b pela recorrência

$$\begin{cases} T(n) = \Theta(1) & \text{se } n = 1 \\ T(n) = aT(\frac{n}{b}) + f(n) & \text{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então $T(n)$ pode ser limitado assintoticamente para potências exatas de b como a seguir:

Prova do Método Mestre

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0 \in \mathbb{R}^+$, então $T(n) = \Theta(n^{\log_b a})$

2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \cdot lgn)$

3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0 \in \mathbb{R}^+$, e se $a \cdot f(n/b) \leq c \cdot f(n)$ para $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$

6.3. Tópicos

O algoritmo de Strassen

No capítulo 2.2.2 analisamos um algoritmo para multiplicar matrizes quadradas com complexidade de $O(n^3)$ multiplicações, sendo n o tamanho (número de linhas e colunas) das matrizes. Também mencionamos que existem algoritmos mais eficientes. A idéia do algoritmo de Strassen é: subdivide os matrizes num produto $A \times B = C$ em quatro sub-matrizes com o metade do comprimento (e, portanto, um quarto de elementos):

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \times \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}.\right)$$

Com essa subdivisão, o produto AB obtem-se pelas equações

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

e precisa-se de oito multiplicações de matrizes de tamanho $n/2$ ao invés de uma multiplicação de matrizes de tamanho n. A recorrência correspondente (somente considerando multiplicações)

$$T(n) = 8T(n/2) + O(1)$$

infelizmente tem solução $T(n) = O(n^3)$ e portanto não melhora o algoritmo

simples. Strassen inventou as equações

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}(B_{21} + B_{22}) \\
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

(cuja verificação é simples). Essas equações contém somente *sete* multiplicações de matrizes de tamanho $n/2$, que leva à recorrência

$$T(n) = 7T(n/2) + O(1)$$

para o número de multiplicações, cuja solução é $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

6.4. Exercícios

(Soluções a partir da página 228.)

Exercício 6.1
Resolva os seguintes recorrências

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(2n/3) + 1$
3. $T(n) = 3T(n/4) + n \log n$
4. $T(n) = 2T(n/2) + n \log n$
5. $T(n) = 4T(n/2) + n^2 \lg n$
6. $T(n) = T(n-1) + \log n$
7. $T(n) = 2T(n/2) + \frac{n}{\log n}$

8. $T(n) = 3T(n/2) + n \log n$

Exercício 6.2
Aplique o teorema mestre nas seguintes recorrências:

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(2n/3) + 1$
3. $T(n) = 3T(n/4) + n \log n$
4. $T(n) = 2T(n/2) + n \log n$

Exercício 6.3
Descreva o funcionamento dos problemas abaixo, extraia as recorrências dos algoritmos, analise a complexidade pelos três métodos vistos em aula.

1. Produto de número binários (Exemplo 5.3.8 da Toscani&Veloso).
2. Algoritmo de Strassen para multiplicação de matrizes (Cormen 28.2 e capítulo 6.3).
3. Encontrar o k-ésimo elemento de uma lista não ordenada (Cormen 9.3).

7. Backtracking

Algoritmos

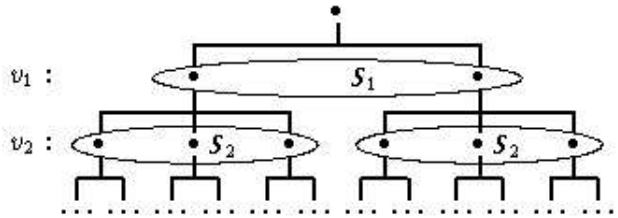
- Supondo um computador de 4.8GHz (4.8 bilhões de instruções por s) e uma instância de tamanho $n = 50$ de um problema qualquer.
- Suponha que o problema pode ser resolvido por três algoritmos com as complexidades n^2, 2^n e 3^n.
  1. Para complexidade $Alg_1 = O(n^2) \iff 0,0000025$ segundos
  2. Para complexidade $Alg_2 = O(2^n) \approx 13$ dias
  3. Para complexidade $Alg_2 = O(3^n) \approx 23080600$ séculos!

Backtracking

- Algoritmo de força bruta "refinado".
- Explora sistematicamente todas possíveis configurações do espaço de busca.
- Assume que uma solução é representada por vetores (v_1, v_2, v_3, ..., v_n) de elementos.
- Percorre na forma de busca em profundidade o domínio dos vetores, até que as soluções ótimas sejam encontradas.

Backtracking

- A exploração do espaço de soluções pode ser representada por uma árvore de busca em profundidade.
- A árvore raramente é armazenada em memória, mas apenas um caminho com comprimento igual à altura da árvore.

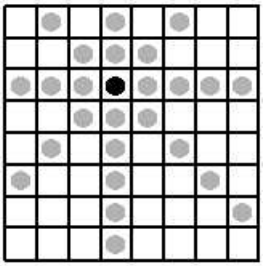


O problema das n-rainhas

PROBLEMA DAS n-RAINHAS

Instância Um tablado de xadrez de dimensão n*n, e n rainhas.

Solução Todas as formas de posicionar as n rainhas no tablado sem que duas rainhas estejam na mesma coluna, linha ou diagonal.



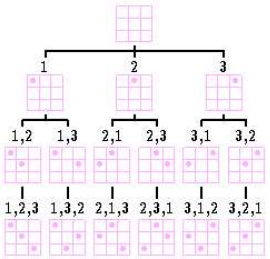
O problema das n-rainhas (simplificado: sem restrição da diagonal)

O que representam as folhas da árvore de busca para este problema?

O problema das n-rainhas

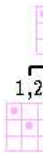
- A melhor solução conhecida para este problema é via Backtracking.
- Existem n^n formas de posicionar n rainhas no tablado
- Restringindo uma rainha por coluna problema passa de n^n para $n!$
- Pode-se facilmente verificar que $n! \leq n^n$
- Pela aproximação de Stirling

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(1/n)) \tag{7.1}$$



O problema das n-rainhas

Se considerarmos também a restrição de diagonal podemos reduzir ainda mais



o espaço de busca (neste caso, nenhuma solução é factível)

Backtracking

- Testa soluções sistematicamente até que a solução esperada seja encontrada
- Durante a busca, se a inserção de um novo elemento "não funciona", o algoritmo retorna para a alternativa anterior (*backtracks*) e tenta um novo ramo
- Quando não há mais elementos para testar, a busca termina
- É apresentado como um algoritmo recursivo
- O algoritmo mantém somente uma solução por vez

Backtracking

- Durante o processo de busca, alguns ramos podem ser evitados de ser explorados
    1. O ramo pode ser infactível de acordo com restrições do problema
    2. Se garantidamente o ramo não vai gerar uma solução ótima

O problema do caixeiro viajante

Encontrar uma rota de menor distância tal que, partindo de uma cidade inicial, visita todas as outras cidades uma única vez, retornando à cidade de partida ao final.

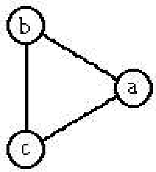
PROBLEMA DO CAIXEIRO VIAJANTE

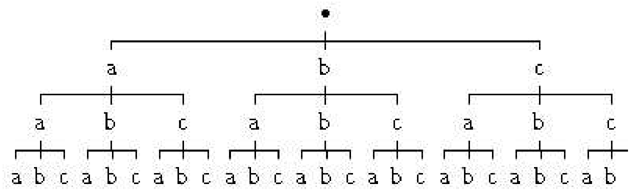
Instância Um grafo G=(V,E) com pesos p (distâncias) atribuídos aos links. $V = [1, n]$ sem perda de generalidade.

Solução Uma rota que visita todos vértices exatamente uma vez, i.e. uma permutação de $[1, n]$.

Objetivo Minimizar o custo da rota $\sum_{1 \le i < n} p_{\{i,i+1\}} + p_{\{n,1\}}$.

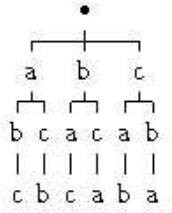
O problema do caixeiro viajante





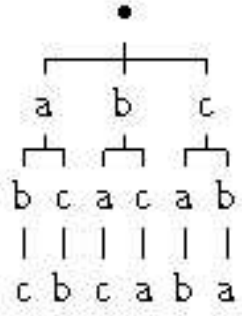
O problema do caixeiro viajante

- Para cada chamada recursiva existem diversos vértices que podem ser selecionados
- Vértices ainda não selecionados são os candidatos possíveis
- A busca exaustiva é gerada caso nenhuma restrição for imposta
- Todas as permutações de cidades geram as soluções factíveis ($P_n = n(n-1)(n-2)...1 = n!$)
- Este problema têm solução $n^2 * 2^n$ usando Programação Dinâmica
- Mas para resolver este problema em PD é necessário muita memória!



O problema do caixeiro viajante

- Alguma idéia de como diminuir ainda mais o espaço de busca?



Vantagens x Desvantagens e Características Marcantes

Vantagens

- Fácil implementação
- Linguagens da área de programação lógica (prolog, ops5) trazem mecanismos embutidos para a implementação de backtracking

Desvantagens

- Tem natureza combinatória. A busca exaustiva pode ser evitada com o teste de restrições, mas o resultado final sempre é combinatório

Característica Marcante

- Backtracking = "retornar pelo caminho": constroem o conjunto de soluções ao retornarem das chamadas recursivas.

Problema de Enumeração de conjuntos

ENUMERAÇÃO DE CONJUNTOS

Instância Um conjunto de n itens S=$a_1, a_2, a_3, ...a_n$.

Solução Enumeração de todos os subconjuntos de S.

- A enumeração de todos os conjuntos gera uma solução de custo exponencial 2^n.

Problema da Mochila

PROBLEMA DA MOCHILA

Instância Um conjunto de n itens $a_1, a_2, ...a_n$ e valores de importância v_i e peso w_i referentes a cada elemento i do conjunto; um valor K referente ao limite de peso da mochila.

Solução Quais elementos selecionar de forma a maximizar o valor total de "importância" dos objetos da mochila e satisfazendo o limite de peso da mochila?

- O problema da Mochila fracionário é polinomial
- O problema da Mochila 0/1 é NP-Completo
  - A enumeração de todos os conjuntos gera uma solução de custo exponencial 2^n
  - Solução via PD possui complexidade de tempo $O(Kn)$ (pseudo-polinomial) e de espaço $O(K)$

Problema de coloração em grafos

PROBLEMA DE COLORAÇÃO EM GRAFOS

Instância Um grafo G=(V,E) e um conjunto infinito de cores.

Solução Uma coloração do grafo, i.e. uma atribuição $c : V \to C$ de cores tal que vértices vizinhos não têm a mesma cor: $c(u) \neq c(v)$ para $(u, v) \in E$.

Objetivo Minimizar o número de cores $|C|$.

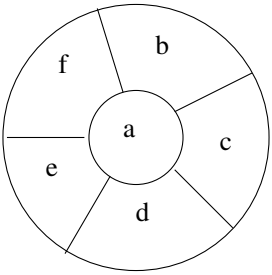
- Coloração de grafos de intervalo é um problema polinomial.
- Para um grafo qualquer este problema é NP-completo.
- Dois números são interessantes nesse contexto:
  - O *número de clique* $\omega(G)$: O tamanho máximo de uma clique que se encontra como sub-grafo de G.
  - O *número cromático* $\chi(G)$: O número mínimo de cores necessárias para colorir G.
- Obviamente: $\chi(V) \geq \omega(G)$
- Um grafo G é *perfeito*, se $\chi(H) = \omega(H)$ para todos sub-grafos H.
- Verificar se o grafo permite uma 2-coloração é polinomial (grafo bipartido).
- Um grafo **k-partido** é um grafo cujos vértices podem ser particionados em k conjuntos disjuntos, nos quais não há arestas entre vértices de um mesmo conjunto. Um grafo 2-partido é o mesmo que grafo bipartido.

Coloração de Grafos

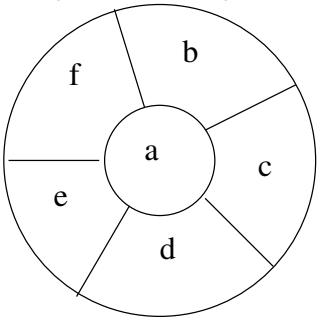
- A coloração de mapas é uma abstração do problema de colorir vértices.
- Projete um algoritmo de backtracking para colorir um grafo planar (mapa).
- Um grafo planar é aquele que pode ser representado em um plano sem qualquer intersecção entre arestas.
- Algoritmo $O(n^n)$, supondo o caso em que cada área necessite uma cor diferente

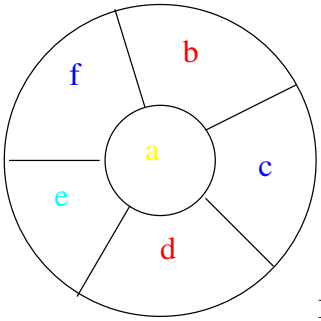
- Teorema de Kuratowski: um grafo é planar se e somente se não possuir *minor* K_5 ou $K_{3,3}$.
- Teorema das Quatro Cores: Todo grafo planar pode ser colorido com até quatro cores (1976, Kenneth Appel e Wolfgang Haken, University of Illinois)
- Qual o tamanho máximo de um clique de um grafo planar?
- Algoritmo $O(4^n)$, supondo todas combinações de área com quatro cores.
- Existem 3^{n-1} soluções possíveis (algoritmo $O(3^n)$) supondo que duas áreas vizinhas nunca tenham a mesma cor.

De quantas cores precisamos?



De quantas cores precisamos?





Precisamos de 4 cores!

Backtracking

BT-COLORING

Entrada Grafo não direcionado G=(V,E)

Saída Designação de cores para cada vértice, sem que haja vértices vizinhos com a mesma cor e minimizando o número de cores utilizadas

```
1  boolean bt coloring(int áreaID, int cor) {
2    if (áreaID > n) return true
3    if (corOK(áreaID, cor)) {
4      mapCor[áreaID] = cor
5      for (int cor = 1; cor <= 4; cor++) {
6        if (bt coloring(áreaID + 1, cor))
7          return true;
8      end for
9      } return false;
10   }
```

Backtracking

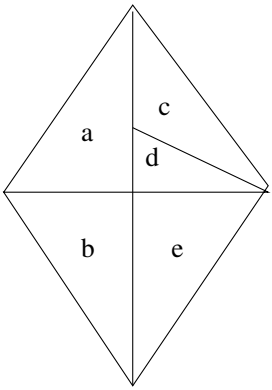
corOK (áreaID, cor)

Entrada Grafo G=(V,E) e duas cores.

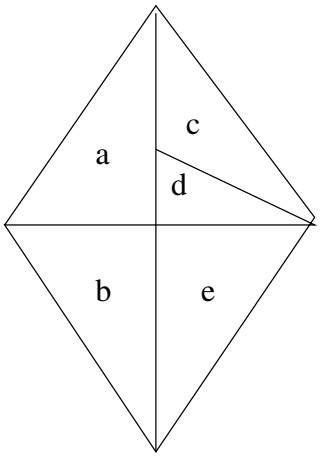
Saída *true* se a um nó vizinho de *áreaID* foi designada *cor* e *false* caso contrário.

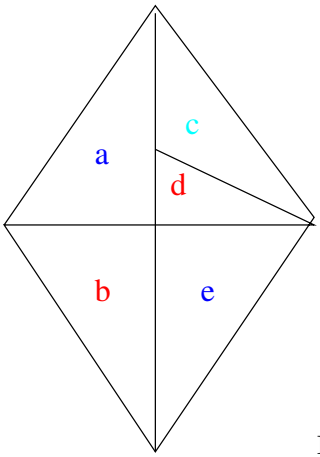
```
1  boolean corOK(int áreaID, int cor) {
2    for (int i=0; i<i|map[áreaID]|; i++) {
3      ithAdj = map[áreaID][i];
4      if (mapCor[ithAdj] = cor)
5        return false;
6      }
7    }
8    return(true)
9  }
```

De quantas cores precisamos?



De quantas cores precisamos?



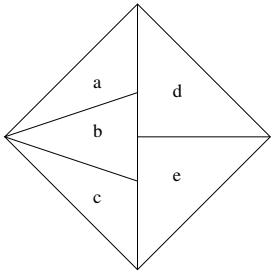


Precisamos de 3 cores!

Coloração de Grafos

- Existe um algoritmo $O(n^2)$ para colorir um grafo com 4 cores (1997, Neil Robertson, Daniel P. Sanders, Paul Seymour).
- Mas talvez sejam necessárias menos que 4 cores para colorir um grafo!
- Decidir se para colorir um grafo planar são necessárias 3 ou 4 cores é um problema NP-completo.

De quantas cores precisamos?



Roteamento de Veículos

Roteamento de Veículos

Instância Um grafo G=(V,A), um depósito v_0, frota de veículos com capacidade Q (finita ou infinita), demanda $q_i > 0$ de cada nó ($q_0 = 0$, distância $d_i > 0$ associada a cada aresta

Solução Rotas dos veículos.

Objetivo Minimizar a distância total.

- Cada rota começa e termina no depósito.
- Cada cidade $V \setminus v_0$ é visitada uma única vez e por somente um veículo (que atende sua demanda total).
- A demanda total de qualquer rota não deve superar a capacidade do caminhão.

Roteamento de Veículos

- Mais de um depósito
- veículos com capacidades diferentes
- Entrega com janela de tempo (período em que a entrega pode ser realizada
- Periodicidade de entrega (entrega com restrições de data)

- Entrega em partes (uma entrega pode ser realizada por partes)
- Entrega com recebimento: o veículo entrega e recebe carga na mesma viagem
- Entrega com recebimento posterior: o veículo recebe carga após todas entregas
- ...

Backtracking versus Branch & Bound

- Backtracking: enumera todas as possíveis soluções com exploração de busca em profundidade.
  - Exemplo do Problema da Mochila: enumerar os 2^n subconjuntos e retornar aquele com melhor resultado.
- Branch&Bound: usa a estratégia de backtracking
  - usa limitantes inferior (relaxações) e superior (heurísticas e propriedades) para efetuar cortes
  - explora a árvore da forma que convier
  - aplicado apenas a problemas de otimização.

Métodos Exatos

- Problemas de Otimização Combinatória: visam minimizar ou maximizar um objetivo num conjunto finito de soluções.
- Enumeração: Backtracking e Branch&Bound.
- Uso de cortes do espaço de busca: Planos de Corte e Branch&Cut.
- Geração de Colunas e Branch&Price.

Métodos não exatos

- Algoritmos de aproximação: algoritmos com garantia de aproximação $S = \alpha S^*$.
- Heurísticas: algoritmos aproximados sem garantia de qualidade de solução.
  - Ex: algoritmos gulosos não ótimos, busca locais, etc.

- Metaheurísticas: heurísticas guiadas por heurísticas (*meta*=além + *heuriskein* = encontrar).
  - Ex: algoritmos genéticos, busca tabu, GRASP (*greedy randomized adaptive search procedure*), simulating de annealing, etc.

8. Algoritmos de aproximação

8.1. Introdução

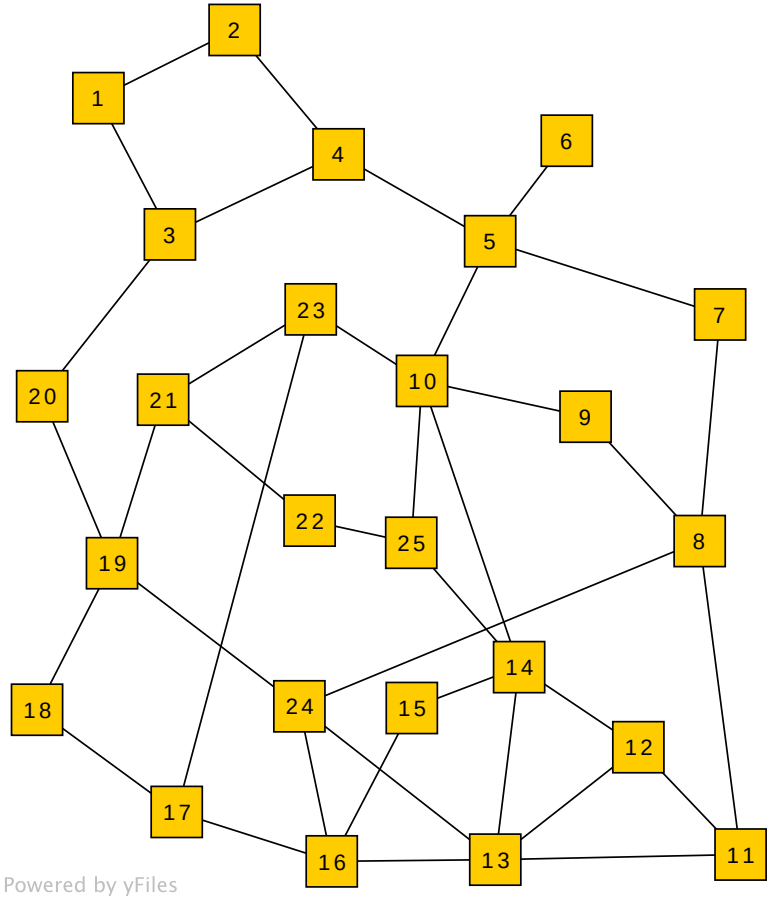
Vertex cover

Considere

> Cobertura por vértices (ingl. vertex cover)
>
> **Instância** Grafo não-direcionado $G = (V, E)$.
>
> **Solução** Uma cobertura $C \subseteq V$, i.e. $\forall e \in E : e \cap C \neq \emptyset$.
>
> **Objetivo** Minimiza $|C|$.

A versão de decisão de Cobertura por vértices é NP-completo.

O que fazer?



Simplificando o problema

- Vértice de grau 1: Usa o vizinho.
- Vértice de grau 2 num triângulo: Usa os dois vizinhos.

Redução de cobertura por vértices

Entrada Grafo não-direcionado $G = (V, E)$.

Saída Um conjunto $C \subseteq V$ e um grafo G', tal que cada cobertura de vértices contém C, e a união de C com a cobertura mínima de G' é uma cobertura mínima de G.

```
1  Reduz(G) :=
2    while (alguma regra em baixo se aplica) do
3      Regra 1:
4        if ∃u ∈ V : deg(u) = 1 then
5          seja {u, v} ∈ E
6          C := C ∪ {v}
7          G := G − {u, v}
8        end if
9      Regra 2:
10       if ∃u ∈ V : deg(u) = 2 then
11         seja {u, v}, {u, w} ∈ E
12         if {v, w} ∈ E then
13           C := C ∪ {v, w}
14           G := G − {u, v, w}
15         end if
16   end while
17   return (C, G)
```

Árvore de busca

Árvore de busca

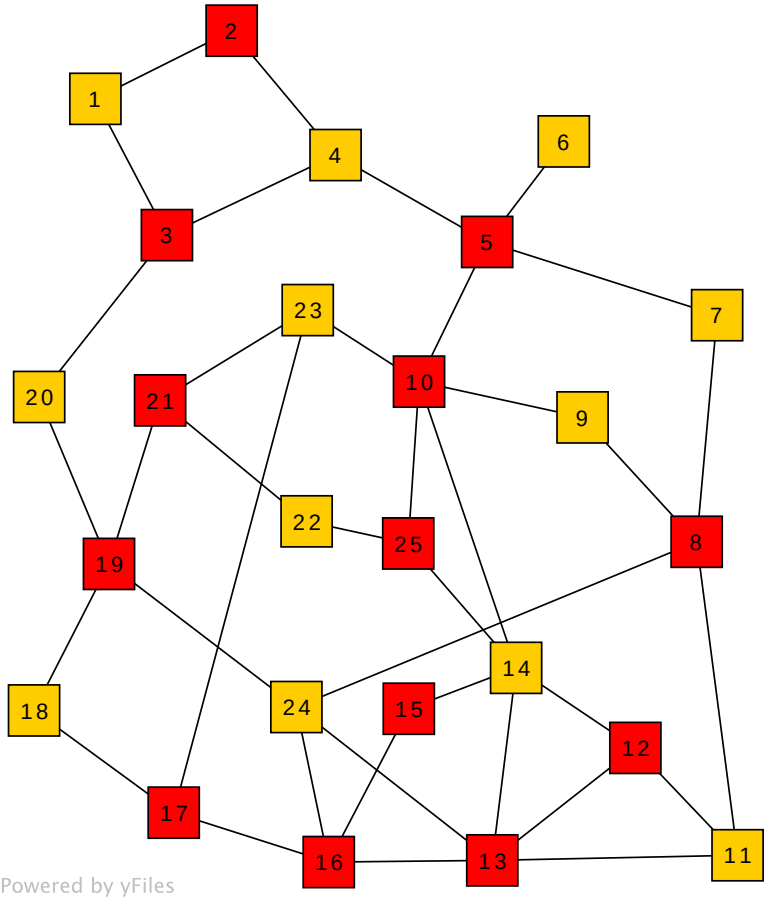
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura por vértices $S \subseteq V$ mínima.

```
1  minVertexCover(G):=
2    if E = ∅ return ∅
3    escolhe {u, v} ∈ E
4    C1 := minVertexCover(G − u) ∪ {u}
5    C2 := minVertexCover(G − v) ∪ {v}
6    if (|C1| < |C2|) then
7      return C1
8    else
```

```
 9        return C_2
10    end if
```

Solução ótima?



Problemas de otimização

Definição 8.1
Um *problema de otimização* é uma relação binária $\mathcal{P} \subseteq I \times S$ com instâncias $x \in I$ e soluções $y \in S$, tal que $(x, y) \in \mathcal{P}$ e com

- uma função de otimização $\varphi : \mathcal{P} \to \mathbb{N}$ (ou $\mathbb{Q}$).
- um objetivo: Encontrar mínimo ou máximo $\mathrm{OPT}(x) = \mathrm{opt}\{\phi(x, y) | (x, y) \in \mathcal{P}\}$.

Tipo de problemas

- Construção: Dado x, encontra solução ótima y e o valor $\varphi^*(x)$.
- Avaliação: Dado x, encontra valor ótimo $\mathrm{OPT}(x)$.
- Decisão: Dado x, $k \in \mathbb{N}$, decide se $\mathrm{OPT}(x) \geq k$ (maximização) ou $\mathrm{OPT}(x) \leq k$ (minimização).

Convenção
Escrevemos um problema de otimização na forma

NOME

Instância x

Solução y

Objetivo Minimiza ou maximiza $\phi(x, y)$.

Classes de complexidade

- PO: Problemas de otimização com algoritmo polinomial.
- NPO: Problemas de otimização tal que
  1. Instâncias reconhecíveis em tempo polinomial.
  2. A relação $\mathcal{P}$ é polinomialmente limita.
  3. Para y arbitrário, polinomialmente limitado: $(x, y) \in \mathcal{P}$ decidível em tempo polinomial.
  4. φ é computável em tempo polinomial.

- NPO contém todos problemas de otimização, que satisfazem critérios mínimos de tratabilidade.

Lembrança (veja definição 11.1): $\mathcal{P}$ é polinomialmente limitada, se para soluções $(x, y) \in \mathcal{P}$ temos $|y| \leq p(|x|)$, para um polinômio p.

Motivação

- Para vários problemas não conhecemos um algoritmo eficiente.
- No caso dos problemas NP-completos: solução eficiente é pouco provável.
- O quer fazer?
- Idéia: Para problemas da otimização, não busca o ótimo.
- Uma solução "quase" ótimo também ajuda.

O que é "quase"? Aproximação absoluta

- A solução encontrada difere da solução ótima ao máximo um *valor* constante.
- Erro absoluto:
$$D(x, y) = |\mathrm{OPT}(x) - \varphi(x, y)|$$
- Aproximação absoluta: Algoritmo garante um y tal que $D(x, y) \leq k$.
- Exemplo: Coloração de grafos.
- Contra-exemplo: Knapsack.

O que é "quase"? Aproximação relativa

- A solução encontrada difere da solução ótima ao máximo um *fator* constante.
- Erro relativo:
$$E(x, y) = D(x, y)/\max\{\mathrm{OPT}(x), \varphi(x, y)\}.$$
- Algoritmo ϵ-aproximativo ($\epsilon \in [0, 1]$): Fornece solução y tal que $E(x, y) \leq \epsilon$ para todo x.
- Soluções com $\epsilon \approx 0$ são ótimas.
- Soluções com $\epsilon \approx 1$ são péssimas.

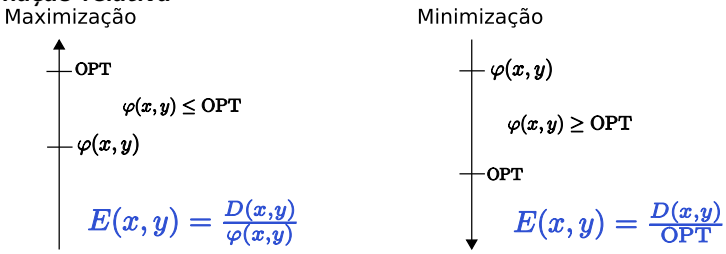
Aproximação relativa: Taxa de aproximação

- Definição alternativa
- Taxa de aproximação $R(x, y) = 1/(1 - E(x, y)) \geq 1$.
- Com taxa r, o algoritmo é r-aproximativo.
- (Não tem perigo de confusão com o erro relativo, porque $r \in [1, \infty]$.)

Aproximação relativa: Exemplos

- Exemplo: Knapsack, Caixeiro viajante métrico.
- Contra-exemplo: Caixeiro viajante.
- Classe correspondente APX: r-aproximativo em tempo polinomial.

Aproximação relativa



Exemplo: Cobertura por vértices gulosa

Cobertura por vértices

Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura por vértices $C \subseteq V$.

```
1  VC GV(G) :=
2    (C, G) := Reduz(G)
3    if V = ∅ then
4      return C
5    else
6      escolhe v ∈ V : deg(v) = Δ(G) { grau máximo }
7      return C ∪ {v} ∪ VC-GV(G − v)
8    end if
```

□

Proposição 8.1
O algoritmo VC-GV é uma $O(\log |V|)$-aproximação.
Prova. Seja G_i o grafo depois da iteração i e C^* uma cobertura ótima ($|C^*| = \mathrm{OPT}(G)$).
A cobertura ótima C^* é uma cobertura para G_i também. Logo, a soma dos graus dos vértices em C^* (contando somente arestas em G_i!) ultrapassa o número de arestas em G_i

$$\sum_{v \in C^*} \delta(v) \geq ||G_i||$$

e o grau médio satisfaz

$$\bar{\delta}(G_i) \geq \frac{||G_i||}{|C^*|} = \frac{||G_i||}{\mathrm{OPT}(G)}.$$

Como o grau máximo é maior que o grau médio temos também

$$\Delta(G_i) \geq \frac{||G_i||}{\mathrm{OPT}(G)}.$$

Com isso podemos estimar

$$\begin{aligned}\sum_{0 \leq i < \mathrm{OPT}} \Delta(G_i) &\geq \sum_{0 \leq i < \mathrm{OPT}} \frac{||G_i||}{|\mathrm{OPT}(G)|} \geq \sum_{0 \leq i < \mathrm{OPT}} \frac{||G_{\mathrm{OPT}}||}{|\mathrm{OPT}(G)|} \\ &= ||G_{\mathrm{OPT}}|| = ||G|| - \sum_{0 \leq i < \mathrm{OPT}} \Delta(G_i)\end{aligned}$$

ou

$$\sum_{0 \leq i < \mathrm{OPT}} \Delta(G_i) \geq ||G||/2$$

i.e. a metade das arestas foi removido em OPT iterações. Essa estimativa continua a ser válido, logo depois

$$OPT \lceil \lg ||G|| \rceil \leq OPT \lceil 2 \lg |G| \rceil = O(OPT\, lg|G|)$$

iterações não tem mais arestas. Como em cada iteração foi escolhido um vértice, a taxa de aproximação é $\lg |G|$. ■

Exemplo: Buscar uma Cobertura por vértices

Cobertura por vértices

Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura por vértices $C \subseteq V$.

```
1  VC B(G) :=
2    (C, G) := Reduz(G)
3    if V = ∅ then
4      return C
5    else
6      escolhe v ∈ V : deg(v) = Δ(G) { grau máximo }
7      C1 := C ∪ {v} ∪ VC-B(G − v)
8      C2 := C ∪ N(v) ∪ VC-B(G − v − N(v))
9      if |C1| < |C2| then
10       return C1
11     else
12       return C2
13     end if
14   end if
```

Aproximação: Motivação

- Queremos uma aproximação
- Quais soluções são aproximações boas?
- Problemas:
  - Tempo polinomial desejada
  - Aproximação desejada ("heurística com garantia")
  - Freqüentemente: a análise e o problema. Intuição:
  - Simples verificar se um conjunto é uma cobertura.
  - Difícil verificar a minimalidade.

Exemplo: Outra abordagem

Cobertura por vértices

Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura por vértices $C \subseteq V$.

```
1  VC GE(G) :=
2    (C,G) := Reduz(G)
3    if E = ∅ then
4      return C
5    else
6      escolhe e = {u,v} ∈ E
7      return C ∪ {u,v} ∪ VC-GE(G − {u,v})
8    end if
```

Proposição 8.2
Algoritmo VC-GE é uma 2-aproximação para VC.

Prova. Cada cobertura contém ao menos um dos dois vértices escolhidos, i.e.

$$|C| \geq \phi_{\text{VC-GE}}(G)/2 \Rightarrow 2\text{OPT}(G) \geq \phi_{\text{VC-GE}}(G)$$

■

Técnicas de aproximação

- Aproximações gulosas
- Randomização
- Busca local
- Programação linear
- Programação dinâmica
- Algoritmos seqüenciais ("online"), p.ex. para particionamento.

8.2. Aproximações com randomização

Randomização

- Idéia: Permite escolhas randômicas ("joga uma moeda")
- Objetivo: Algoritmos que decidem correta com probabilidade alta.

- Objetivo: Aproximações com *valor esperado* garantido.
- Minimização: $E[\varphi_A(x)] \leq 2\mathrm{OPT}(x)$
- Maximização: $2E[\varphi_A(x)] \geq \mathrm{OPT}(x)$

Randomização: Exemplo

Satisfatibilidade máxima, Maximum SAT

Instância Fórmula $\varphi \in \mathcal{L}(V)$, $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ em FNC.

Solução Uma atribuição de valores de verdade $a : V \to \mathbb{B}$.

Objetivo Maximiza o número de cláusulas satisfeitas

$$|\{C_i \mid [\![C_i]\!]_a = v\}|\,.$$

Nossa solução

```
1  SAT–R(φ) :=
2     seja φ = φ(v_1, ..., v_k)
3     for all i ∈ [1,k] do
4        escolhe v_i = v com probabilidade 1/2
5     end for
```

Aproximação?

- Surpresa: Algoritmo é 2-aproximação.

Prova. Seja X a variável aleatória que denota o número de claúsulas satisfeitas. Afirmação: $E[X] \geq (1 - 2^{-l})n$ com l o número mínimo de literais por claúsula. Portanto, com $l \geq 1$, temos $E[X] \geq n/2$.
Prova da afirmação: $P[[\![C_i]\!] = f] \leq 2^{-l}$ e logo $P[[\![C_i]\!] = v] \geq (1 - 2^{-l})$ e

$$E[X] = \sum_{1 \leq i \leq k} E[[\![C_i]\!] = v] = \sum_{1 \leq i \leq k} P[[\![C_i]\!] = v] = (1 - 2^{-l})n.$$

■

Outro exemplo

Cobertura por vértices guloso e randomizado.

```
1 VC–RG(G) :=
2   seja w̄ := Σ_{v∈V} deg(v)
3   C := ∅
4   while E ≠ ∅ do
5     escolhe v ∈ V com probabilidade deg(v)/w̄
6     C := C ∪ {v}
7     G := G − v
8   end while
9   return C ∪ V
```

Resultado: $E[\phi_{\text{VC-RG}}(x)] \leq 2\text{OPT}(x)$.

8.3. Aproximações gulosas

Problema de mochila (Knapsack)

KNAPSACK

Instância Itens $X = [1, n]$ com valores $v_i \in \mathbb{N}$ e tamanhos $t_i \in N$, para $i \in X$, um limite M, tal que $t_i \leq M$ (todo item cabe na mochila).

Solução Uma seleção $S \subseteq X$ tal que $\sum_{i \in S} t_i \leq M$.

Objetivo Maximizar o valor total $\sum_{i \in S} v_i$.

Observação: Knapsack é NP-completo.

Como aproximar?

- Idéia: Ordene por v_i/t_i ("valor médio") em ordem decrescente e enche o mochila o mais possível nessa ordem.

Abordagem

```
1 K–G(v_i, t_i) :=
2   ordene os itens tal que v_i/t_i ≥ v_j/t_j , ∀i < j .
3   for i ∈ X do
4     if t_i < M then
```

```
5          S := S ∪ {i}
6          M := M − t_i
7        end if
8      end for
9      return S
```

Aproximação boa?

- Considere

$$v_1 = 1, \ldots, v_{n-1} = 1, v_n = M - 1$$
$$t_1 = 1, \ldots, t_{n-1} = 1, t_n = M = kn \qquad k \in \mathbb{N} \text{ arbitrário}$$

- Então:

$$v_1/t_1 = 1, \ldots, v_{n-1}/t_{n-1} = 1, v_n/t_n = (M-1)/M < 1$$

- K-G acha uma solução com valor $\varphi(x) = n-1$, mas o ótimo é $\mathrm{OPT}(x) = M - 1$.

- Taxa de aproximação:

$$\mathrm{OPT}(x)/\varphi(x) = \frac{M-1}{n-1} = \frac{kn-1}{n-1} \geq \frac{kn-k}{n-1} = k$$

- K-G não possui taxa de aproximação fixa!

- Problema: Não escolhemos o item com o maior valor.

Tentativa 2: Modificação

```
1  K–G'( v_i , t_i ) :=
2      S_1 := K–G( v_i , t_i )
3      v_1 := Σ_{i∈S_1} v_i
4      S_2 := {argmax_i v_i}
5      v_2 := Σ_{i∈S_2} v_i
6      if v_1 > v_2 then
7        return S_1
8      else
9        return S_2
10     end if
```

Aproximação boa?

- O algoritmo melhorou?
- Surpresa

Proposição 8.3
K-G' é uma 2-aproximação, i.e. $\mathrm{OPT}(x) < 2\varphi_{\text{K-G}'}(x)$.

Prova. Seja j o primeiro item que K-G não coloca na mochila. Nesse ponto temos valor e tamanho

$$\bar{v_j} = \sum_{1 \le i < j} v_i \le \varphi_{\text{K-G}}(x)$$
$$\bar{a_j} = \sum_{1 \le i < j} a_i \le M$$

Afirmação: $\mathrm{OPT}(x) < \bar{v_j} + v_j$. Nesse caso

1. Seja $v_j \le \bar{v_j}$.
$$\mathrm{OPT}(x) < 2\bar{v_j} \le 2\varphi_{\text{K-G}}(x) \le 2\varphi_{\text{K-G}'}$$
2. Seja $v_j > \bar{v_j}$
$$\mathrm{OPT}(x) \le 2v_j \le 2v_{\max} \le 2\varphi_{\text{K-G}'}$$

Prova da afirmação:

$$\mathrm{OPT}(x) \le \bar{v_j} + \underbrace{(M - \bar{a_j})\frac{v_j}{a_j}}_{\text{densidade mais alto no resto}} < \bar{v_j} + v_j$$

porque $\bar{a_j} + a_j > M \iff M - \bar{a_j} < a_j$ $\quad a_j$ não cabe mais

■

Problemas de particionamento

- Bin packing (BP): "Particionamento em baldes" (PB).

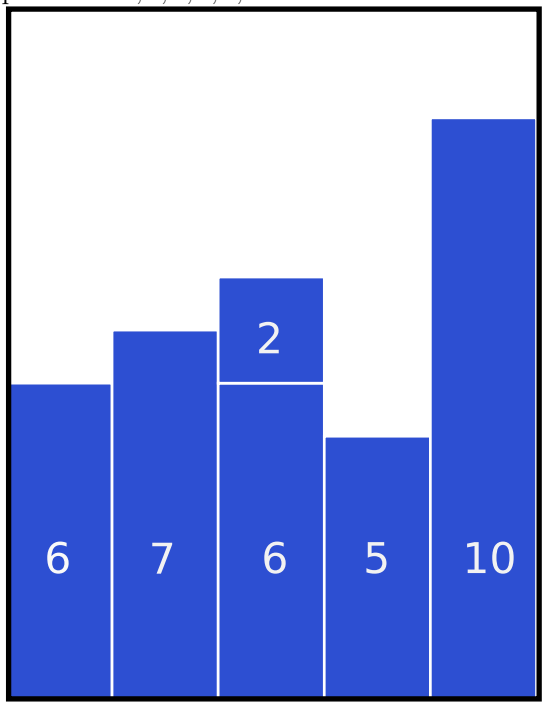
> BIN PACKING
>
> **Instância** Seqüencia de itens $v_1, \ldots, v_n$, $v_i \in]0, 1], v_i \in \mathbb{Q}^+$

Solução Partição $\bigcup_{1 \leq i \leq k} P_i = [1, n]$ tal que $\sum_{j \in Pi} v_j \leq 1$ para todos $i \in [1, n]$.

Objetivo Minimiza o número de partições ("baldes") k.

Abordagem?

- Idéia simples: Próximo que cabe (PrC).
- Por exemplo: Itens $6, 7, 6, 2, 5, 10$ com limite 12.



Aproximação?

- Interessante: PrC é 2-aproximação.
- Observação: PrC é um algoritmo on-line.

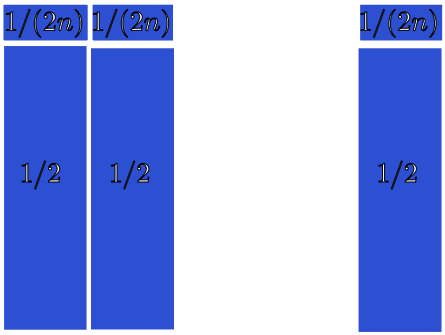
Prova. Seja B o número de baldes usadas, $V = \sum_{1 \le i \le n} v_i$. $B \le 2\lceil V \rceil$ porque duas baldes consecutivas contém uma soma > 1. Mas precismaos ao menos $B \ge \lceil V \rceil$ baldes, logo $\mathrm{OPT}(x) \ge \lceil V \rceil$. Portanto, $\varphi_{\mathrm{PrC}}(x) \le 2\lceil V \rceil \le 2\mathrm{OPT}(x)$. ■

Aproximação melhor?

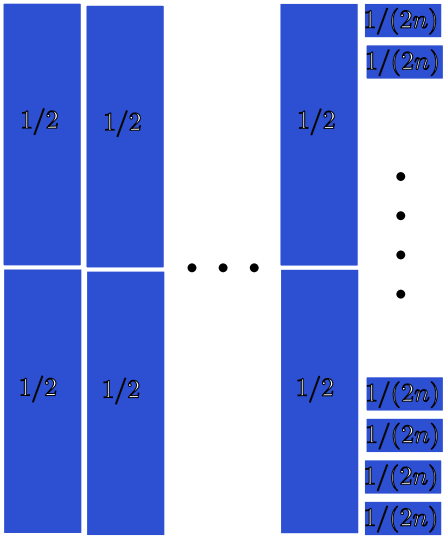
- Isso é o melhor possível!
- Considere os $4n$ itens

$$\underbrace{1/2, 1/(2n), 1/2, 1/(2n), \ldots, 1/2, 1/(2n)}_{2n \text{ vezes}}$$

- O que faz PrC? $\varphi_{\mathrm{PrC}}(x) = 2n$: baldes com



- Ótimo: n baldes com dois elementos de $1/2 +$ um com $2n$ elementos de $1/(2n)$. $\mathrm{OPT}(x) = n = 1$.



- Portanto: Assintoticamente a taxa de aproximação 2 é estrito.

Melhores estratégias

- Primeiro que cabe (PiC), on-line, com "estoque" na memória
- Primeiro que cabe em ordem decrescente: PiCD, off-line.
- Taxa de aproximação?

$$\varphi_{\text{PiC}}(x) \leq \lceil 1.7\text{OPT}(x) \rceil$$
$$\varphi_{\text{PiCD}}(x) \leq 1.5\text{OPT}(x) + 1$$

Prova. Da segunda taxa de aproximação: Considere a partição $A \cup B \cup C \cup D = \{v_1, \ldots, v_n\}$ com

$$A = \{v_i \mid v_i > 2/3\}$$
$$B = \{v_i \mid 2/3 \geq v_i > 1/2\}$$
$$C = \{v_i \mid 1/2 \geq v_i > 1/3\}$$
$$D = \{v_i \mid 1/3 \geq v_i\}$$

PiCD primeiro vai abrir $|A|$ baldes com os itens em A e depois $|B|$ baldes com os itens em B. Temos que analisar o que acontece com os itens em C e D.

Hipótese: Um balde contém somente itens do tipo D. Logo: Os outros baldes tem espaço livre menos que 1/3. Portanto:

$$B \leq \left\lceil \frac{V}{2/3} \right\rceil \leq 3/2V + 1 \leq 3/2\mathrm{OPT}(x) + 1$$

Caso contrário (nenhum balde contém somente itens tipo D): PiCD acha a solução ótima.

- 1a observação: O número de baldes sem itens tipo D é o mesmo (eles são os últimos distribuídos em não abrem novo balde). Logo é suficiente mostrar
$$\varphi_{\mathrm{PiCD}}(x \setminus D) = OPT(x \setminus D)$$
- 2a observação: Os itens tipo A não importam: Sem itens D, nenhum outro item cabe junto. Logo:
$$\varphi_{\mathrm{PiCD}}(x \setminus D) = |A| + \varphi_{\mathrm{PiCD}}(x \setminus (A \cup D))$$
- 3a observação: O melhor caso para os restantes itens são *pares* de elementos em B e C: Nesse situação, PiCD acha a solução ótima.

■

Aproximação melhor?

- Tese doutorado D. S. Johnson, 1973, 70 pág
$$\varphi_{\mathrm{PiCD}}(x) \leq \frac{11}{9}\mathrm{OPT}(x) + 4$$
- Baker, 1985
$$\varphi_{\mathrm{PiCD}}(x) \leq \frac{11}{9}\mathrm{OPT}(x) + 3$$

8.4. Esquemas de aproximação

Novas considerações

- Freqüentemente uma r-aproximação não é suficiente. $r = 2$: 100% de erro!
- Existem aproximações melhores? p.ex. para SAT? problema do mochila?

- Desejável: Esquema de aproximação em tempo polinomial (EATP); polynomial time approximation scheme (PTAS)
  - Para cada entrada e taxa de aproximação r:
  - Retorne r-aproximação em tempo polinomial.

Um exemplo: Mochila máxima (Knapsack)

- Problema de mochila (veja página 106):
  - n itens com valor v_i, peso w_i.
  - Mochila de tamanho W.
  - Escolher subconjunto de itens que cabe em W de maior peso total.
- Algoritmo MM-PD com programação dinâmica: tempo $O(n \sum_i v_i)$.
- Desvantagem: Pseudo-polinomial.

8.5. Exercícios

Exercício 8.1
Um aluno propõe a seguinte heurística para Binpacking: Ordene os itens em ordem crescente, coloca o item com peso máximo junto com quantas itens de peso mínimo que é possível, e depois continua com o segundo maior item, até todos itens foram colocados em bins. Temos o algoritmo

```
1 ordene itens em ordem crescente
2 m:=1; M:=n
3 while (m<M) do
4   abre novo balde, coloca v_M, M := M − 1
5   while (v_m cabe e m < M) do
6     coloca v_m no balde atual
7     m := m + 1
8   end while
9 end while
```

Qual a qualidade desse algoritmo? É um algoritmo de aproximação? Caso sim, qual a taxa de aproximação dele? Caso não porque?

Exercício 8.2
Um aluno propõe o seguinte pré-processamento para o algoritmo SAT-R de aproximação para MAX-SAT (página 161): Caso a instância contém claúsulas

com um único literal, vamos escolher uma delas, definir uma atribuição parcial que satisfaz-la, e eliminar a variável correspondente. Repetindo esse procedimento, obtemos uma instância cujas claúsulas tem 2 ou mais literais. Assim, obtemos $l \geq 2$ na análise do algoritmo, o podemos garantir que $E[X] \geq 3n/4$, i.e. obtemos uma 4/3-aproximação.
Este análise é correto ou não?

Parte III.

Teoria de complexidade

9. Do algoritmo ao problema

9.1. Introdução

Motivação

- Análise e projeto: Foca em *algoritmos*.
- Teoria de complexidade: Foca em *problemas*.
- Qual a complexidade intrínsica de problemas?
- *Classes de complexidade* agrupam problemas.
- Interesse particular: Relação entre as classes.

Abstrações: Alfabetos, linguagens

- Seja Σ um *alfabeto* finito de símbolos.
- Codificação: Entradas e saídas de um algoritmo são *palavras* sobre o alfabeto $\omega \in \Sigma^*$.
- Tipos de problemas:
  - Problema construtivo: Função $\Sigma^* \to \Sigma^*$ Alternativa: relação $R \subseteq \Sigma^* \times \Sigma^*$.
  - Problema de decisão: Função $\Sigma^* \to \{S, N\}$ Equivalente: conjunto $L \subseteq \Sigma^*$ (uma *linguagem*).
  - Problema de otimização: Tratado como problema de decisão com a pergunta "Existe solução $\geq$ k?" ou "Existe solução $\leq k$?".
- Freqüentemente: Alfabeto $\Sigma = \{0, 1\}$ com codificação adequada.

Definição 9.1
Uma *linguagem* é um conjunto de palavras sobre um alfabeto: $L \subseteq \Sigma^*$.

Modelo de computação: Máquina de Turing

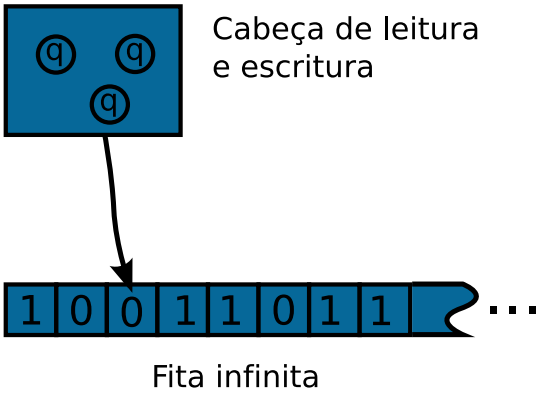
Alan Mathison Turing (*1912, +1954)

- Foco: Estudar as limitações da complexidade, não os algoritmos.
- Portanto: Modelo que facilite o estudo teórica, não a implementação.
- Solução: Máquina de Turing.

> But I was completely convinced only by Turing's paper.
> (Gödel em uma carta para Kreisel, em maio de 1968, falando sobre uma definição da computação.).

Computing is normally done by writing certain symbols on paper. "We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent j . The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same. [28].

Máquina de Turing



Máquina de Turing (MT)

$$M = (Q, \Sigma, \Gamma, \delta)$$

- Alfabeto de entrada Σ (sem branco $\llcorner$)
- Conjunto de estados Q entre eles três estados diferentes:
  - Um estado inicial $q_0 \in Q$, um que aceita $q_a \in Q$ e um que rejeita $q_r \in Q$.
- Alfabeto de fita $\Gamma \supseteq \Sigma$ (inclusive $\llcorner \in \Gamma$)
- Regras $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, escritas da forma

$$q, a \to q' a' D$$

(com $q, q' \in Q$, $a, a' \in \Sigma$ e $D \in \{L, R\}$).

Máquina de Turing: Operação

- Início da computação:
  - No estado inicial q_0 com cabeça na posição mais esquerda,
  - com entrada $w \in \Sigma^*$ escrita na esquerda da fita, resto da fita em branco.
- Computação: No estado q lendo um símbolo a aplica uma regra $qa \to q'a'D$ (um L na primeira posição não tem efeito) até

- não encontrar uma regra: a computação termina, ou
- entrar no estado q_a: a computação termina e *aceita*, ou
- entrar no estado q_r: a computação termina e *rejeita*.
- Outra possibilidade: a computação não termina.

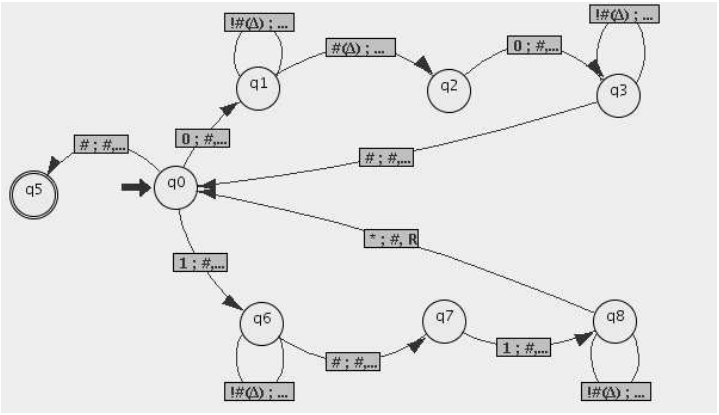
Exemplo 9.1 (Decidir ww^R)

Tabela da transição

Seja $\Sigma = \{0, 1\}$. Uma máquina de Turing que reconhece a linguagem $\{ww^R | w \in \Sigma^*\}$ é

q0	0	#	R	q1
q1	#(Δ)	Δ	L	q2
q1	!#(Δ)	Δ	R	q1
q2	0	#	L	q3
q3	!#(Δ)	Δ	L	q3
q3	#	#	R	q0
q0	#	#	R	q5
q0	1	#	R	q6
q6	!#(Δ)	Δ	R	q6
q6	#	#	L	q7
q7	1	#	L	q8
q8	!#(Δ)	Δ	L	q8
q8	*	#	R	q0

Notação gráfica



(convenções e abreviações do Turing machine simulator; veja página da disciplina). ◊

Máquinas não-determinísticas

- Observe: Num estado q lendo símbolo a temos exatamente uma regra da forma $qa \to q'a'D$.
- Portanto a máquina de Turing se chama *determinística* (MTD).
- Caso temos mais que uma regra que se aplica em cada estado q e símbolo a a máquina é *não-determinística* (MTND).
- A função de transição muda para

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Linguagem de uma MTD

- O conjunto de palavras que uma MTD M aceita é *a linguagem reconhecida* de M.

$$L(M) = \{w \in \Sigma^* \mid M \text{ aceita } w\}$$

- Uma linguagem tal que existe um MTD M que reconhece ela é *Turing-reconhecível por* M.

$$\text{L Turing-reconhecível} \iff \exists M : \ L = L(M)$$

- Observe que uma MTD nem sempre precisa parar. Se uma MTD M sempre para, ela *decide* a sua linguagem.
- Uma linguagem Turing-reconhecível por uma MTD M que sempre para é *Turing-decidível*.

$$\text{L Turing-decidível} \iff \exists M : \ L = L(M) \text{ e } M \text{ sempre para}$$

Linguagem de uma MTND

- Conjunto de linguagens Turing-reconhecíveis: linguagens *recursivamente enumeráveis* ou *computavelmente enumeráveis*.
- Conjunto de linguagens Turing-decidíveis: linguagens *recursivas* ou *computáveis*.

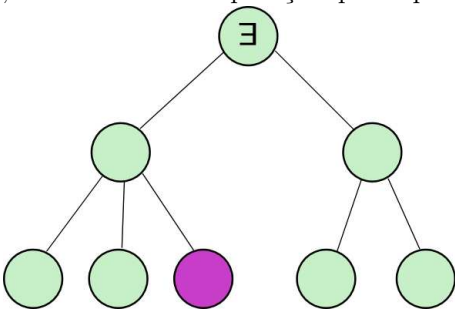
- Para uma máquina não-determinística temos que modificar a definição: ela precisa somente um estado que aceita e
  - ela reconhece a linguagem

    $$L(M) = \{w \in \Sigma^* \mid \text{existe uma computação tal que } M \text{ aceita } w\}$$

  - ela decide uma linguagem se todas computações sempre param.

Máquina de Turing não-determinística

- Resposta *sim*, se existe uma computação que responde *sim*.



Robustez da definição

- A definição de uma MTD e *robusto*: as seguintes modificações tem poder computacional equivalente:
  1. Uma MT com $k > 1$ fitas,
  2. uma MT com fita dupla infinita,
  3. uma MT com alfabeto restrito $\Sigma = \{0, 1\}$,
  4. uma MTND e
  5. uma MT com fita de dois ou mais dimensões.
- O poder computacional também é equivalente com vários outros modelos de computação (p.ex. cálculo lambda, máquina RAM, autômato celular): Tese de Church-Turing.

Prova. (Rascunho.)

1. Seja a_{ij} o símbolo na posição j da fita i e l_i o índice do última símbolo usado na fita i. A máquina com única fita representa os k fitas da forma

$$\#a_{11} \dots a_{1l_1} \# \dots \# a_{k1} \dots a_{kl_k}.$$

   Para representar as posições da cabeças, usamos símbolos marcados com um ponto da forma $\dot{a}$. A simulação procede em três passos: (i) Determinar os símbolos em baixa das cabeças em um passo (ii) executar as operações da k cabeças. Caso um cabeça ultrapassa a direita da representação, a MTD estende ela, copiando todos símbolos da direita mais uma para direita.

2. Seja a_i o símbolo na posição i da fita, com $i \in \mathbb{Z}$. A máquina com única fita meio-infinito representa essa fita como

$$a_0 \langle < a_{-1} a_1 \rangle \langle < a_{-2} a_2 \rangle \dots .$$

   com símbolos novos em $\Sigma \cup \Sigma^2$. Os estados desse máquina são $Q \times \{T, B\}$. Eles representam se a cabeça trabalho em cima ou em baixo, com regras modificadas adequadamente. Ela também tem regras para passar de cima para baixo e vice versa.

3. Cada símbolo é representado como seqüência de $0, 1$ do comprimento $w = \lceil \log |\Sigma| \rceil$. Na simulação, a MTD primeiro leia os w símbolos atuais, determina e escreve a representação do novo símbolo e depois se movimenta w posições para esquerda ou direita.

4. Uma simulação das configurações, listando todas as execuções possíveis sistematicamente.

■

Decibilidade versus complexidade

- Qual é o poder computacional?
- Surpreendentemente (?), vários problemas não são decidíveis.
- Exemplo: O "Entscheidungsproblem" de Hilbert, o problema de parada, etc.
- A equivalência dos modelos significa que o modelo concreto não importa?
- Sim para computabilidade, não para complexidade!

Exemplo 9.2 (Desvio: Máquina universal)
Considere a máquina $M = (\{u, d\}, \{a, b\}, \{a, b, \llcorner\}, \delta)$ com

$$\delta = \{ua \to ubL, ub \to uaL, u\llcorner \to dbR, da \to u\llcorner R, db \to daR, d\llcorner \to uaL\}.$$

Essa máquina é universal? Veja http://www.wolframscience.com/prizes/tm23.
Obs: Aparentemente o problema foi resolvido em 24/10/2007. ◊

10. Classes de complexidade

10.1. Definições básicas

Complexidade pessimista

- Recursos básicos: *tempo* e *espaço*.
- A *complexidade de tempo (pessimista)* é uma função

$$t : \mathbb{N} \to \mathbb{N}$$

  tal que $t(n)$ é o número máximo de passos para entradas de tamanho n.

- A *complexidade de espaço (pessimista)* é uma função

$$s : \mathbb{N} \to \mathbb{N}$$

  tal que $s(n)$ é o número máximo de posições usadas para entradas de tamanho n.

- Uma MTND tem complexidades de tempo $t(n)$ ou espaço $s(n)$, se essas funções são limites superiores para todas computações possíveis de tamanho n.

Funções construtíveis

- Em que segue, nos vamos considerar somente funções t e s que são *tempo-construtíveis* e *espaço-construtíveis*.
- $t(n)$ é tempo-construtível: existe uma MTD que tem a complexidade de tempo $t(n)$ e precisa $t(n)$ passos para uma entrada.
- $s(n)$ é espaço-construtível: existe uma MTD que tem a complexidade de espaço $s(n)$ e precisa $s(n)$ posições para uma entrada.
- Exemplos: $n^k, \log n, 2^n, n!, \ldots$.

Classes de complexidade fundamentais

- Uma *classe de complexidade* é um conjunto de linguagens.
- Classes fundamentais: Para $T, S : \mathbb{N} \to \mathbb{N}$ e um problema $X \subseteq \Sigma^*$
  - $L \in \mathsf{DTIME}[t(n)]$ se existe uma máquina Turing determinística tal que aceita L com complexidade de tempo $t(n)$.
  - $L \in \mathsf{NTIME}[t(n)]$ se existe uma máquina Turing não-determinística que aceita L com complexidade de tempo $t(n)$.
  - $L \in \mathsf{DSPACE}[s(n)]$ se existe uma máquina Turing determinística que aceita L com complexidade de espaço $s(n)$.
  - $L \in \mathsf{NSPACE}[s(n)]$ se existe uma máquina Turing não-determinística que aceita L com complexidade de espaço $s(n)$.

Hierarquia básica

- Observação

$$\mathsf{DTIME}[F(n)] \subseteq \mathsf{NTIME}[F(n)] \subseteq \mathsf{DSPACE}[F(n)] \subseteq \mathsf{NSPACE}[F(n)]$$

- Definições conhecidas:

$$\mathsf{P} = \bigcup_{k \geq 0} \mathsf{DTIME}[n^k]; \quad \mathsf{NP} = \bigcup_{k \geq 0} \mathsf{NTIME}[n^k]$$

- Definições similares para espaço:

$$\mathsf{PSPACE} = \bigcup_{k \geq 0} \mathsf{DSPACE}[n^k]; \quad \mathsf{NSPACE} = \bigcup_{k \geq 0} \mathsf{NSPACE}[n^k]$$

- Com a observação acima, temos

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{DSPACE} \subseteq \mathsf{NSPACE}\,.$$

Prova. (Da observação.) Como uma máquina não-determinística é uma extensão da uma máquina determinística, temos obviamente $\mathsf{DTIME}[F(n)] \subseteq \mathsf{NTIME}[F(n)]$ e $\mathsf{DSPACE}[F(n)] \subseteq \mathsf{NSPACE}[F(n)]$. A inclusão $\mathsf{NTIME}[F(n)] \subseteq \mathsf{DSPACE}[F(n)]$ segue, porque todas computações que precisam menos que $F(n)$ passos, precisam menos que $F(n)$ espaço também. ■

Classes de complexidade

Zoológico de complexidade

10.2. Hierarquias básicas

Aceleração

Teorema 10.1
Podemos comprimir ou acelerar computações com um fator constante. Para todos $c > 0$ no caso de espaço temos

$$L \in \mathsf{DSPACE}[s(n)] \Rightarrow L \in \mathsf{DSPACE}[cs(n)]$$
$$L \in \mathsf{NSPACE}[s(n)] \Rightarrow L \in \mathsf{NSPACE}[cs(n)]$$

e no caso do tempo, para máquinas de Turing com $k > 1$ fitas e $t(n) = \omega(n)$

$$L \in \mathsf{DTIME}[s(n)] \Rightarrow L \in \mathsf{DTIME}[cs(n)]$$
$$L \in \mathsf{NTIME}[s(n)] \Rightarrow L \in \mathsf{NTIME}[cs(n)]$$

Prova. (Rascunho.) A idéia é construir uma MT M' que simula uma MT M executando m passos em um passo. M' começa de copiar a entrada para uma outra fita, codificando cada m símbolos em um símbolo em tempo $n + \lceil n/m \rceil$. Depois a simulação começa. Em cada passo, M' leia os símbolos na esquerda e direta e na posição atual em tempo 4. Depois ela calcula os novos estados no controle finito, e escreve os três símbolos novos em tempo 4. Logo, cada m passos podem ser simulados em 8 passos em tempo

$$n + \lceil n/m \rceil + \lceil 8t(n)/m \rceil \leq n + n/m + 8t(n)/m + 2 \leq 3n + 8t(n)/n$$

que para $cm \geq 16 \Leftrightarrow 8/m \leq c/2$ e n suficientemente grande não ultrapassa $ct(n)$. O número finito de palavras que não satisfazem esse limite superior é reconhecido diretamente no controle finito. ■

Hierarquia de tempo (1)

- É possível que a decisão de todos problemas tem um limite superior (em termos de tempo ou espaço)? Não.

Teorema 10.2
Para $t(n)$ e $s(n)$ total recursivo, existe um linguagem L tal que $L \notin \mathsf{DTIME}[t(n)]$ ou $L \notin \mathsf{DSPACE}[s(n)]$, respectivamente.

Prova. (Rascunho). Por diagonalização. As máquinas de Turing são enumeráveis: seja $M_1, M_2, \ldots$ uma enumeração deles e seja $x_1, x_2, \ldots$ uma enumeração das palavras em Σ^*. Define

$$L = \{x_i \mid M_i \text{ naõ aceita } x_i \text{ em tempo } t(|x_i|)\}.$$

Essa linguagem é decidível: Uma MT primeira computa $t(n)$ (que é possível porque $t(n)$ é recursivo e total.). Depois com entrade x_i, ela determina i e a máquina M_i correspondente e simula M_i para $t(n)$ passos. Se M_i aceita, ela rejeita, senão ela aceita.
Essa linguagem não pertence a $\mathsf{DTIME}[t(n)]$. Prova por contradição: Seja $L = L(M_i)$. Então $x_i \in L$? Caso sim, M_i não aceita em tempo $t(n)$, uma contradição. Caso não, M_i não aceita em tempo $t(n)$, uma contradição também. ■

Hierarquia de tempo (2)

Além disso, as hierarquias de tempo são "razoavelmente densos":

Teorema 10.3 (Hartmanis,Stearns)
Para f, g com g tempo-construtível e $f \log f = o(g)$ temos

$$\mathsf{DTIME}(f) \subsetneq \mathsf{DTIME}(g).$$

Para funções f, g, com $g(n) \geq \log_2 n$ espaço-construtível e $f = o(g)$ temos

$$\mathsf{DSPACE}(f) \subsetneq \mathsf{DSPACE}(g).$$

Prova. (Rascunho.) Para provar o segundo parte (que é mais fácil) temos que mostrar que existe uma linguagem $L \subseteq \Sigma^*$ tal que $L \in \mathsf{DSPACE}[g]$ mas $L \notin \mathsf{DSPACE}[f]$. Vamos construir uma MT M sobre alfabeto de entrada $\Sigma = \{0, 1\}$ cuja linguagem tem essa característica. A ideia básica é diagonalização: com entrada w simula a máquina M_w com entrada w e garante de nunca reconhecer a mesma linguagem que M_w caso ela é limitada por f.
Considerações:

1. Temos que garantir que M precisa não mais que $g(n)$ espaço. Portanto, M começa de marcar $g(|w|)$ espaço no começo (isso é possível porque g é espaço-construtível). Caso a simulação ultrapassa o espaço marcado, M rejeita.

2. Nos temos que garantir que M pode simular todas máquinas que tem limite de espaço $f(n)$. Isso tem duas problemas (a) M tem um alfabeto de fita fixo, mas a máquina simulada pode ter mais símbolos de fita.

Portanto, a simulação precisa um fator c_1 de espaço a mais. (b) Por definição, para $f \in o(g)$ é suficiente de ter $f \leq cg$ a partir de um $n > n_0$. Logo para entrada $|w| \leq n_0$ o espaço $g(n)$ pode ser insuficiente para simular qualquer máquina que precisa espaço $f(n)$. Esses assuntos podem ser resolvidos usando uma enumeração de MT (com alfabeto Σ) tal que cada máquina tem codificações de comprimento arbitrário (por exemplo permitindo $\langle M \rangle 10^n$).

3. Além disso, temos que garantir que a simulação para. Portanto M usa um contador com $O(\log n)$ espaço, e rejeita caso a simulação ultrapassa $c_2^{f(n)}$ passos; c_2 depende das características da máquina simulada (número de estados, etc.).

Com essas preparações, com entrada w, M construa M_w, verifica que M_w é uma codificação de uma MT e depois simula M_w com entrada w. M rejeita se M_w aceita e aceita se M_w rejeita. Não tem máquina que, em espaço $f(n)$ reconhece $L(M)$. Senão, caso M' seria uma máquina desses, com entrada $w = \langle M' \rangle 01^n$ para n suficientemente grande M consegue de simular M' e portanto, se $w \in M'$ então $w \notin M$ e se $w \notin M'$ então $w \in M$, uma contradição. A idéia da prova do primeiro parte é essencialmente a mesma. A fator de $\log f$ surge, porque para simular um MT um dado número de passos por uma outra, é necessário contar o número de passos até $f(n)$ em $\log f(n)$ bits. Com uma simulação mais eficiente (que não é conhecida) seria possível de obter um teorema mais forte. ■

Espaço polinomial

Teorema 10.4 (Savitch)
Para cada função $S(n) \geq \log_2 n$ (espaço-construtível)

$$\mathsf{NSPACE}[S(n)] \subseteq \mathsf{DSPACE}[S(n)^2]$$

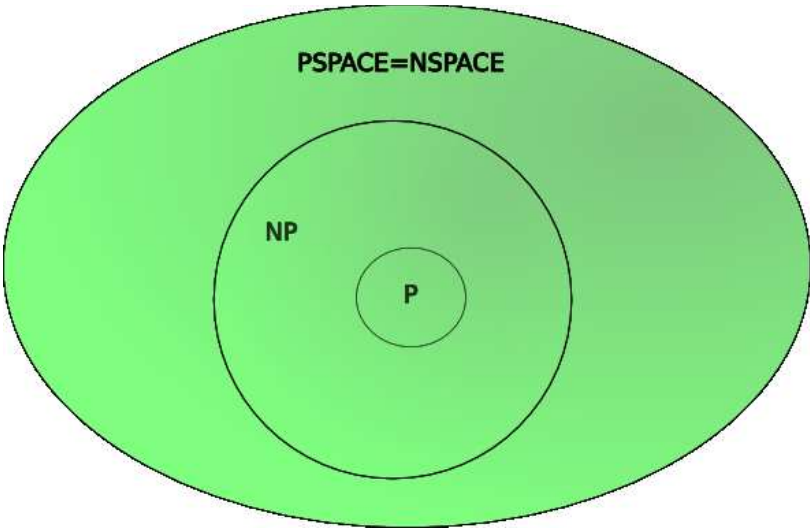
- Corolário: $\mathsf{DSPACE} = \mathsf{NSPACE}$
- Não-determinismo ajuda pouco para espaço!

Walter
J. Savitch
(*1943)

Prova. (Rascunho.) Se $X \in \mathsf{NSPACE}[T(n)]$, o tempo é limitado por um $c^{S(n)}$. A construção do Savitch procura deterministicamente uma transição do estado inicial para um estado final com menos que $c^{S(n)}$ passos. A abordagem

e divisão e conquista: Para ver, se tem uma transição $A \Rightarrow B$ com menos de 2^i passos, tenta se tem um caminho $A \Rightarrow I$ e $I \Rightarrow B$, cada um com 2^{i-1} passos para todas configurações I com menos que $S(n)$ lugares. A altura do árvore de procura é $O(S(n))$ e o espaço da cada nível também e $O(S(n))$, resultando em $O(S(n)^2)$ para tudo.
A função tem que ter $S(n) \geq \log_2 n$, por que para a simulação precisamos também gravar a posição de cabeça de entrada em cada nível, que precisa $\log_2 n$ bits. ■

Espaço polinomial (2)



11. Teoria da NP-completude

11.1. Caracterizações e problemas em NP

A hierarquia de Chomsky classifica linguagens em termos de autômatos e gramáticas que aceitam ou produzem elas:

Linguagem	Nome	Tipo	Autômato	Gramática
Regular	REG	3	Autômato finito (determinístico)	Regular
Livre de contexto	CFL	2	Autômato de pilha (não-determinístico)	Livre de contexto
Sensitiva ao contexto	CSL	1	MT linearmente limitada (não-determinístico)	Sensitiva ao contexto
Recursivamente enumerável	RE	0	MT	Sistema semi-Thue (sem restrição)

O seguinte teorema relaciona a hierarquia de Chomsky com as classes de complexidade (sem prova, referências em [5], [3, Th. 25.6] e [26, Th. 7.16]).

Teorema 11.1 (Complexidade das linguagens da hierarquia de Chomsky)

$$\begin{aligned}
\mathsf{REG} &= \mathsf{DSPACE}[O(1)] = \mathsf{DSPACE}[o(\log\log n)] \\
\mathsf{REG} &\subseteq \mathsf{DTIME}[n] \\
\mathsf{CFL} &\subseteq \mathsf{DSPACE}[n^3] \\
\mathsf{CSL} &= \mathsf{NSPACE}[n]
\end{aligned}$$

Normalmente, nosso interesse são soluções, não decisões: Ao seguir vamos definir P e NP em termos de soluções. As perguntas centrais como $\mathsf{P} \neq \mathsf{NP}$ acabam de ter respostas equivalentes.

P e NP em termos de busca

- A computação de uma solução pode ser vista como função $\Sigma^* \to \Sigma^*$
- Exemplo: Problema SAT construtiva: Uma solução é uma atribuição.
- Definição alternativa: Uma computação é uma relação $R \subseteq \Sigma^* \times \Sigma^*$.
- Vantagem: Permite mais que uma solução para cada entrada.

- Nosso interesse são soluções que podem ser "escritas" em tempo polinomial:

  Definição 11.1
  Uma relação binário R é *polinomialmente limitada* se

  $$\exists p \in \mathsf{poly} : \forall (x, y) \in R : |y| \leq p(|x|)$$

P e NP em termos de busca

- P e NP tem definição como classes de problemas de decisão.
- A linguagem correspondente a uma relação R é

  $$L_R = \{x | \exists y : (x, y) \in R\}$$

- A classe P: Linguagens L_R tal que existe uma MTD que, com entrada $x \in L_R$, em tempo polinomial, busque $(x, y) \in R$ ou responda, que não tem.
- Essa definição do P às vezes é chamado FP ou PF.
- A classe NP: Linguagens L_R tal que existe MTD que, com entrada (x, y), decida se $(x, y) \in R$ em tempo polinomial. y se chama *certificado*.

Exemplos de problemas em NP

$$\mathsf{CLIQUE} = \{\langle G, k\rangle | \text{Grafo não-direcionado } G \text{ com clique de tamanho } k\}$$
$$\mathsf{SAT} = \{\langle \phi\rangle | \phi \text{ fórmula satisfatível da lógica propositional em FNC}\}$$
$$\mathsf{TSP} = \{\langle M, b\rangle | \text{Matriz simétrica de distâncias } M \text{ que tem TSP-ciclo} \leq b\}$$
$$\mathsf{COMPOSITE} = \{\langle n\rangle | n = m_1 m_2 \text{ com } m_1, m_2 > 1\}$$

11.2. Reduções

Reduções

Definição 11.2 (Redução em tempo polinomial)
Uma (many-one) *redução* entre duas linguagens L, L' com alfabetos Σ e Σ' é um função total $f : \Sigma^* \to \Sigma'^*$ tal que $x \in L \iff f(x) \in L'$. Se f é computável em tempo polinomial, se chama uma *redução em tempo polinomial*; escrevemos $L \leq_P L'$.

Definição 11.3 (Problemas difíceis e completos)
Dado uma classe de problemas C e um tipo de redução $\leq$, um problema L é C-$\leq$-*difícil*, se $L' \leq L$ para todos $L' \in C$. Um problema L que é C-$\leq$-difícil é *completo*, se $L \in C$.

- Motivo: Estudar a complexidade *relativa* de problemas; achar problemas "difíceis" para separar classes.
- Do interesse particular: A separação de P e NP. Denotamos a classe de problemas NP-completos NPC.

Características de $\leq_P$

Proposição 11.1 (Fecho para baixo)
Se $A \leq_P B$ e $B \in \mathsf{P}$ então $A \in \mathsf{P}$.

Proposição 11.2 (Transitividade)
$\leq_P$ é transitivo, i.e. se $A \leq_P B$ e $B \leq_P C$ então $A \leq_P C$.

Prova. (Fecho para baixo.) Uma instância $w \in A$ pode ser reduzido em tempo polinomial para $w' \in B$. Depois podemos simular B com entrada w' em tempo polinomial. Como a composição de polinômios é um polinômio, $A \in \mathsf{P}$.
(Transitividade.) Com o mesmo argumento podemos reduzir $w \in A$ primeiro para $w' \in B$ e depois para $w'' \in C$, tudo em tempo polinomial. ■

O problema de parada

- O problema da parada

$$\mathsf{HALT} = \{\langle M, w\rangle | \text{MT } M \text{ para com entrada } w\}$$

  não é decidível.

- Qual o caso com

PARADA LIMITADA (INGL. BOUNDED HALT)

Instância MT M, entrada w e um número n (em codificação unária).

Questão M para em n passos?

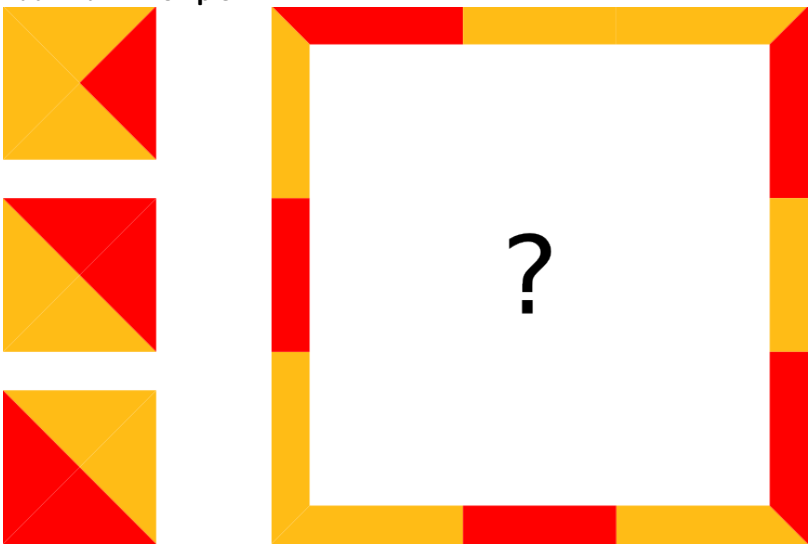
Teorema 11.2

$$\mathsf{BHALT} = \{\langle M, w, 1^n\rangle | \text{MT } M \text{ para com entrada } w \text{ em } n \text{ passos}\}$$

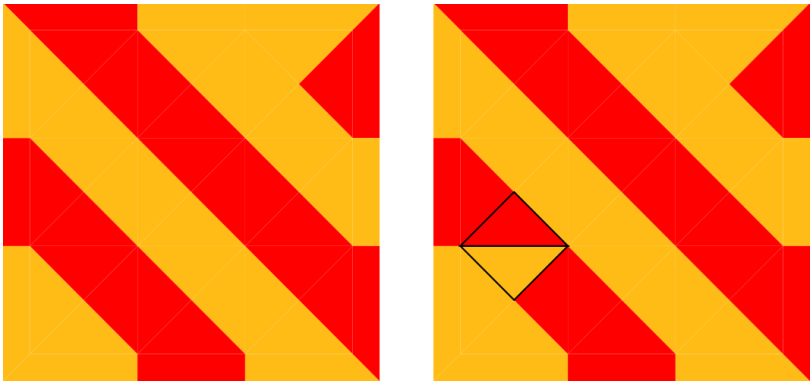
é NP-completo.

Prova. BHALT $\in$ NP porque podemos verificar uma execução em n passos em tempo polinomial. Observe que a codificação de uma execução em limitada polinomialmente em termos da entrada $\langle M, w, 1^n\rangle$ pela codificação de n em unário. Logo é suficiente de mostrar que qualquer problema em NP pode ser reduzido para BHALT.
Para alguma linguagem $L \in$ NP, seja M uma MTND com $L = L(M)$ que aceita L em tempo n^k. Podemos reduzir uma entrada $w \in L$ em tempo polinomial para $w' = \langle M, w, 1^{n^k}\rangle$, temos $w \in L \Leftrightarrow w' \in$ BHALT. Logo $L \leq_P$ BHALT. ■

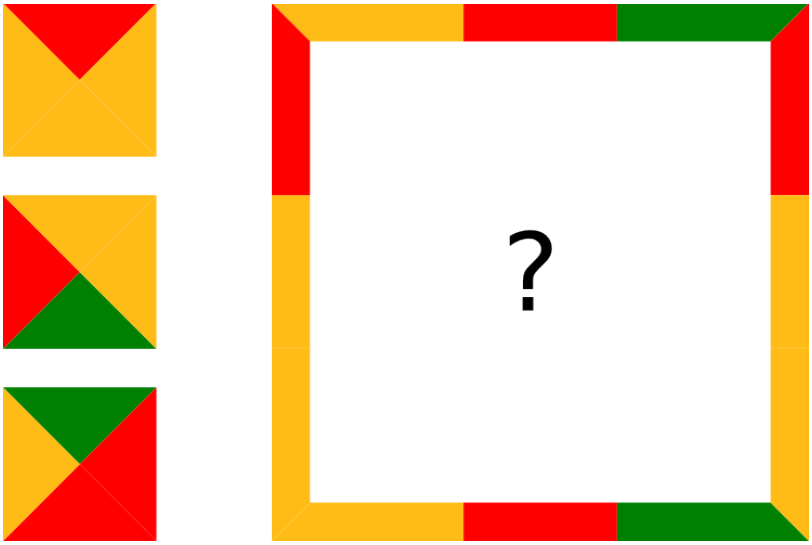
Ladrilhar: Exemplo



Ladrilhar: Solução



Ladrilhar: Exemplo



Ladrilhar: O problema

- Para um conjunto finito de cores C, o *tipo* de um ladrilho é uma função

$$t : \{N, W, S, E\} \to C.$$

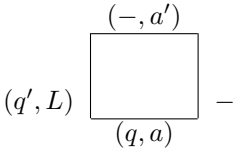
LADRILHAMENTO

Instância Tipos de ladrilhos $t_1, \ldots, t_k$ e um grade de tamanho $n \times n$ com cores nas bordas. (Cada ladrilho pode ser representado por quatro símbolos para as cores; a grade consiste de n^2 ladrilhos em branco e $4n$ cores; uma instância tem tamanho $O(k + n^2)$.

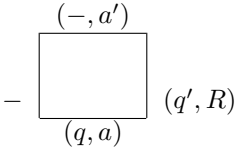
Questão Existe um ladrilhamento da grade tal que todas cores casam (sem rotar os ladrilhos)?

Teorema 11.3 (Levin)
Ladrilhamento é NP-completo.

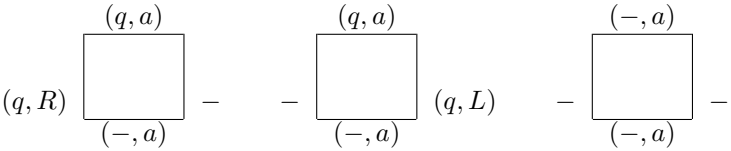
Prova. O problema é em NP, porque dado um conjunto de tipos de ladrilhos e um ladrilhamento, podemos verificar as restrições das cores em tempo polinomial.
Vamos reduzir qualquer problema em $L \in \mathsf{NP}$ para LADRILHAMENTO. Seja $L = L(M)$ para alguma MTND e seja k tal que M precisa tempo n^k. Para entrada w, vamos construir uma instância de LADRILHAMENTO do tamanho $(|w|^k)^2$. Idéia: os cores dos cantos de sul e de norte vão codificar um símbolo da fita a informação se a cabeça está presente e o estado da máquina. Os cores dos cantos oeste e este vão codificar informação adicional para mover a cabeça. O canto sul da grade vão ser colorido conforme o estado inicial da máquina, o canto norte com o estado final e vamos projetor as ladrilhas de tal forma que ladrilhar uma linha (de sul para o norte) e somente possível, se as cores no sul e norte representam configurações sucessores.
Nos vamos usar as cores $Q \cup \{-\} \times \Gamma$ na direção norte/sul e $Q \times \{L, R\} \cup \{-\}$ na direção oeste/este. Para uma regra $q, a \to q', a', L$ os ladrilhos tem a forma



e para $q, a \to q', a', R$



Além disso, tem ladrilhos



As cores no sul da grade representam a configuração inicial

$$(q_0, a_1)(-, a_2) \cdots (-, a_n)(-, \sqcup) \cdots (-, \sqcup)$$

as cores no norte a configuração final (supondo que a máquina limpa a fita depois, que sempre é possível)

$$(q_a, -)(-, \sqcup) \cdots (-, \sqcup)$$

e as cores dos lados oeste e este todos são $-$. Pela construção uma computação da MT que aceita corresponde com um ladrilhamento e vice versa. A construção do grade e das tipos de ladrilhos pode ser computado por uma máquina de Turing em tempo polinomial. ■

Resultado intermediário

- Primeiros problemas em NPC: Para uma separação é "só" provar que Ladrilhamento $\notin$ P ou BHALT $\notin$ P.
- Infelizmente: a prova é difícil, mesmo que a maioria das pesquisadores acredita P $\neq$ NP.
- Outro valor: Para provar que um problema $L \in$ NPC, é suficiente de mostrar que, por exemplo

$$\text{Ladrilhamento} \leq_P L.$$

Proposição 11.3
Se A $\subseteq$ B e A é fechado para baixo em relação à redução $\leq$ e L e B-$\leq$-completo então

$$L \in \mathsf{A} \iff \mathsf{A} = \mathsf{B}.$$

Exemplo: O problema SAT

SAT

Instância Fórmula proposicional em forma normal conjuntiva $\Phi(x_1, \ldots, x_n)$.

Questão Tem uma atribuição $a_1, \ldots, a_n \in \mathbb{B}$ que satisfaz Φ?

Teorema 11.4 (Cook)
SAT é NP-completo.

Prova (1)

Objetivo: Provar Ladrilhamento $\leq_P$ SAT.

Seja

$N_{x,y,c}$ variável "o norte da posição x, y tem cor c"
S, W, E analogamente

$$\begin{aligned} L_{i,x,y} := N_{x,y,t_i(N)} \wedge & \bigwedge_{\substack{c \in C \\ c \neq t_i(N)}} \neg N_{x,y,c} \\ \wedge W_{x,y,t_i(W)} \wedge & \bigwedge_{\substack{c \in C \\ c \neq t_i(W)}} \neg W_{x,y,c} \\ \wedge S_{x,y,t_i(S)} \wedge & \bigwedge_{\substack{c \in C \\ c \neq t_i(S)}} \neg S_{x,y,c} \\ \wedge E_{x,y,t_i(E)} \wedge & \bigwedge_{\substack{c \in C \\ c \neq t_i(E)}} \neg E_{x,y,c} \end{aligned}$$

Prova (2)

Sejam $c_{x,y}$ as cores na bordas. Seja ϕ a conjunção de

$$\bigwedge_{x\in[1,n]}\bigwedge_{y\in[1,n]}\bigvee_{c\in[1,k]} L_{c,x,y} \quad \text{Toda posição tem um ladrilho}$$

$$\bigwedge_{x\in[1,n]} S_{x,1,c_{x,1}} \wedge N_{x,n,c_{x,n}} \quad \text{Cores corretas nas bordas N,S}$$

$$\bigwedge_{y\in[1,n]} W_{1,y,c_{1,y}} \wedge E_{n,y,c_{n,y}} \quad \text{Cores corretas nas bordas W,E}$$

$$\bigwedge_{x\in[1,n[}\bigwedge_{y\in[1,n]} E_{x,y,c} \Rightarrow W_{x+1,y,c} \quad \text{Correspondência E-W}$$

$$\bigwedge_{x\in]1,n]}\bigwedge_{y\in[1,n]} W_{x,y,c} \Rightarrow E_{x-1,y,c} \quad \text{Correspondência W-E}$$

$$\bigwedge_{x\in[1,n[}\bigwedge_{y\in[1,n[} N_{x,y,c} \Rightarrow S_{x,y+1,c} \quad \text{Correspondência N-S}$$

$$\bigwedge_{x\in[1,n[}\bigwedge_{y\in]1,n]} S_{x,y,c} \Rightarrow N_{x,y-1,c} \quad \text{Correspondência S-N}$$

Prova (3)

- O número de variáveis e o tamanho de ϕ é polinomial em n, k; ϕ pode ser computado em tempo polinomial para uma instância de LADRILHAMENTO.
- Portanto, SAT é NP-difícil.
- SAT $\in$ NP, porque para fórmula ϕ e atribuição a, podemos verificar $a \models \phi$ em tempo polinomial.

O significado do P = NP

Kurt Gödel 1958: Uma carta para John von Neumann

Obviamente, podemos construir uma máquina de Turing, que decide, para cada fórmula F da lógica de predicados de primeira ordem e cada número natural n, se F tem uma prova do tamanho n (tamanho = número de símbolos). Seja $\Phi(F, n)$ o número de passos que a máquina precisa para isso, e seja $\Psi(n) = \max_F \Phi(F, n)$. A questão é como $\Phi(n)$ cresce para uma máquina ótima. É possível provar que $\Phi(n) \geq kn$. Se existisse uma máquina com $\Phi(n) \sim kn$ (ou pelo menos $\Phi(n) \sim kn^2$), isso teria conseqüências da maior

importância. Assim, seria óbvio, apesar da indecibilidade do Entscheidungsproblem, poderia-se substituir completamente o raciocínio do matemático em questões de sim-ou-não por máquinas. [25]
Em original a carta diz

> Man kann offenbar leicht eine Turingmaschine konstruieren, welche von jeder Formel F des engeren Funktionenkalküls u. jeder natürl. Zahl n zu entscheiden gestattet, ob F einen Beweis der Länge n hat [Länge = Anzahl der Symbole]. Sei $\Psi(F, n)$ die Anzahl der Schritte, die die Maschine dazu benötigt u. sei $\phi(n) = \max_F \Psi(F, n)$. Die Frage ist, wie rasch $\phi(n)$ für eine optimale Maschine wächst. Man kann zeigen $\phi(n) \geq k \cdot n$. Wenn es wirklich eine Maschine mit $\phi(n) \sim k \cdot n$ (oder auch nur $\sim k \cdot n^2$) gäbe, hätte das Folgerungen von der grössten Tragweite. Es würde nämlich offenbar bedeuten, dass man trotz der Unlösbarkeit des Entscheidungsproblems die Denkarbeit des Mathematikers bei ja-oder-nein Fragen vollständig durch Maschinen ersetzen könnte.

A significado do P = NP

- Centenas de problemas NP-completos conhecidos seriam tratável.
- Todos os problemas cujas soluções são reconhecidas facilmente (polinomial), teriam uma solução fácil.
- Por exemplo na inteligência artificial: planejamento, reconhecimento de linguagens naturais, visão, talvez também composição da música, escrever ficção.
- A criptografia conhecida, baseada em complexidade, seria impossível.

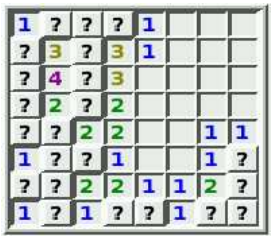
> I have heard it said, with a straight face, that a proof of P = NP would be important because it would airlines schedule their flight better, or shipping companies pack more boxes in their trucks! [2]

Mais um problema NP-completo

MINESWEEPER CONSISTÊNCIA

Instância Uma matriz de tamanho $b \times b$ cada campo ou livre, ou com um número ou escondido.

Decisão A matriz é consistente (tem uma configuração dos campos escondidos, que podem ser "bombas" ou livres tal que os números são corretos)?



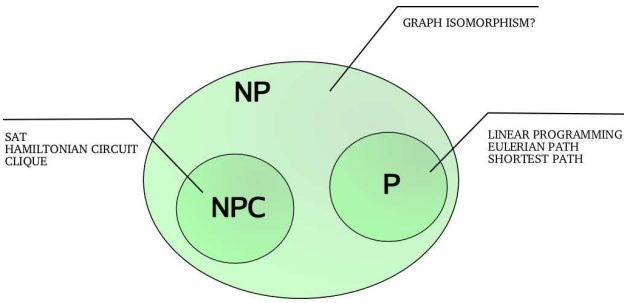
O mundo agora

- O milagre da NP-completude
  - Qualquer problema em NP tem uma redução polinomial para SAT!
  - Por que não usar só SAT? (soluções em 1.3^n)?

Teorema 11.5 (Ladner [19])

- Se $P \neq NP$, existe uma linguagem $L \in NP$ que nem é NP-completo nem em P.

Stephen Arthur Cook (*1939)



Leonid Levin (*1948)

12. Fora do NP

Classes fora do P-NP

$$\mathsf{L} = \mathsf{DSPACE}[\log n]$$
$$\mathsf{NL} = \mathsf{NSPACE}[\log n]$$
$$\mathsf{EXPTIME} = \bigcup_{k>0} \mathsf{DTIME}[2^{n^k}]$$
$$\mathsf{NEXPTIME} = \bigcup_{k>0} \mathsf{NTIME}[2^{n^k}]$$
$$\mathsf{EXPSPACE} = \bigcup_{k>0} \mathsf{DSPACE}[2^{n^k}]$$
$$\mathsf{NEXPSPACE} = \bigcup_{k>0} \mathsf{NSPACE}[2^{n^k}]$$

Co-classes

Definição 12.1 (Co-classes)
Para uma linguagem L, a *linguagem complementar* é $\overline{L} = \Sigma^* \setminus L$. Para uma classe de complexidade C, a *co-classe* $co - C = \{\overline{L} | L \in C\}$ e a classe das linguagens complementares.

Proposição 12.1
$\mathsf{P} = \text{co}-\mathsf{P}$.

- Qual problema pertence à NP?

$$\overline{\mathsf{CLIQUE}}, \overline{\mathsf{SAT}}, \overline{\mathsf{TSP}}, \overline{\mathsf{COMPOSITE}}.$$

Prova. Seja $L \in \mathsf{P}$. Logo existe um MTD M tal que $L = L(M)$ em tempo n^k. Podemos facilmente construir uma MTD que rejeita se M aceita e aceita se M rejeita. ■

A classe co-NP

- A definição da classe NP é unilateral. Por exemplo, considere

TAUT

Instância Fórmula proposicional em forma normal disjuntiva φ.

Decisão φ é uma tautologia (Todas as atribuições satisfazem φ)?

- Uma prova sucinta para esse problema não é conhecido, então *suponhamos* que $\mathsf{TAUT} \notin \mathsf{NP}$.

- Em outras palavras, NP parece de não ser fechado sobre a complementação:

$$\mathsf{co\text{-}NP} \neq \mathsf{NP}\,?$$

Proposição 12.2
Se $L \in \mathsf{NPC}$ então $L \in \mathsf{co\text{-}NP} \iff \mathsf{NP} = \mathsf{co\text{-}NP}$.

Proposição 12.3
TAUT é co-NP-completo.

Prova. (Proposição 12.2.) Seja $L \in \mathsf{NPC}$. ($\rightarrow$): Seja $L \in \mathsf{co\text{-}NP}$. Se $L' \in \mathsf{NP}$, temos $L' \leq_P L \in \mathsf{co\text{-}NP}$, logo $\mathsf{NP} \subseteq \mathsf{co\text{-}NP}$. Se $L' \in \mathsf{co\text{-}NP}$, então $\overline{L'} \in \mathsf{NP}$ e $\overline{L'} \leq_P L \in \mathsf{co\text{-}NP}$, logo $\overline{L'} \in \mathsf{co\text{-}NP}$ e $L' \in \mathsf{NP}$. ($\leftarrow$): Como $L \in \mathsf{NPC} \subseteq \mathsf{NP}$, e $\mathsf{NP} = \mathsf{co\text{-}NP}$, também $L \in \mathsf{co\text{-}NP}$. ■

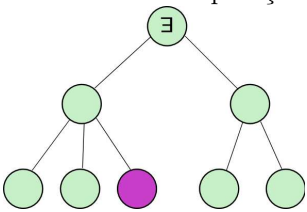
Prova. (Proposição 12.3, rascunho.) $\mathsf{TAUT} \in \mathsf{co\text{-}NP}$, porque uma MT com um estado universal pode testar todas atribuições das variáveis proposicionais e aceita se todas são verdadeiras.
Para provar a completude, temos que provar, que toda linguagem $\overline{L} \in \mathsf{co\text{-}NP} \leq_P \mathsf{TAUT}$. A prova é uma modificação da prova do teorema de Cook: Com entrada $w \in \overline{L}$ produzimos uma fórmula φ_w usando o método de Cook. Temos

$$\begin{aligned} w \in \overline{L} &\iff \varphi_w \text{ satisfatível} && \text{pela def. de } \varphi_w \\ w \in L &\iff \varphi_w \text{ insatisfatível} && \text{negação da afirmação} \\ &\iff \neg\varphi_w \text{ é tautologia} \end{aligned}$$

■

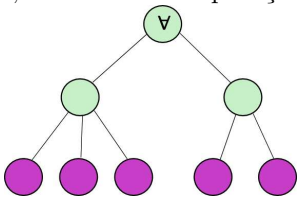
A classe co-NP

- NP: Resposta *sim*, se existe uma computação que responde *sim*.



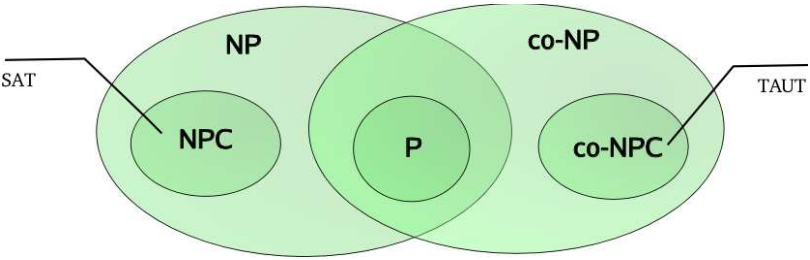
- Ou: Dado um certificado, *verificável* em tempo polinomial.

- co-NP: Resposta *sim*, se todas as computações respondem *sim*



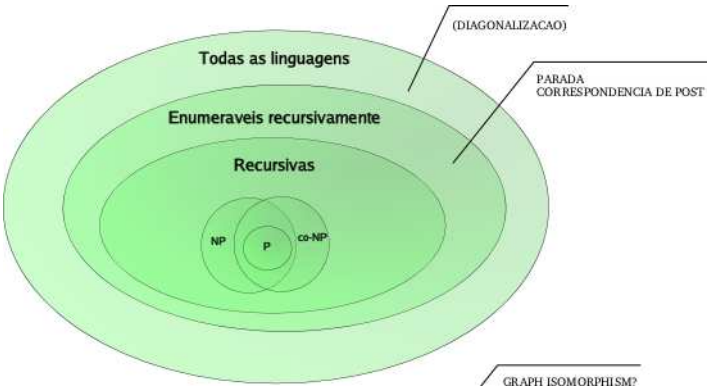
- Ou: Dado um "falsificado", *falsificável* em tempo polinomial.

O mundo da complexidade ao redor do P



12.1. De P até PSPACE

O mundo inteiro (2)



Problemas PSPACE-completos

- Não sabemos, se NP = PSPACE ou até P = PSPACE
- Como resolver isso? Da mesma forma que a questão P = NP: busque problemas PSPACE-completos (relativo a $\leq_P$).
- Considere

FORMULAS BOOLEANAS QUANTIFICADAS (INGL. QUANTIFIED BOOLEAN FORMULAS, QBF)

Instância Uma sentença booleana

$$\Phi := (Q_1 x_1)(Q_2 x_2) \cdots (Q_n x_n)[\varphi(x_1, x_2, \ldots, x_n)]$$

com $Q_i \in \{\forall, \exists\}$.

Decisão Φ é verdadeira?

- Exemplo:

$$(\forall x_1)(\exists x_2)(\forall x_3)(x_1 \vee x_3 \equiv x_2)$$

Teorema 12.1
QBF é PSPACE-completo.

Prova. (Rascunho.) É fácil de provar que QBF $\in$ PSPACE: Podemos verificar recursivamente que a sentença é verdadeira: Para uma fórmula $Qx_1\varphi(x_1, \ldots)$ com $Q \in \{\forall, \exists\}$ vamos aplicar o algoritmos para os casos $\varphi(0)$ e $\varphi(1)$.

Para provar a completude, temos que mostrar que toda linguagem $L \in$ PSPACE pode ser reduzido para QBF. Assume que existe uma MT que reconhece $L = L(M)$ em espaço n^k e seja $w \in L$. A idéia principal é construir uma fórmula $\phi_{q,s,t}$ que afirma que tem uma transição do estado q para s em ao máximo t passos. A idéia é testar $\phi_{q_0,q_f,2^{cf(n)}}$ com $2^{cf(n)}$ sendo o número máximo de estados para entradas de tamanho n.
Um estado pode ser codificado em um string de $|w|^k$ bits. Para $\phi_{q,r,1}$ podemos usar basicamente a mesma fórmula do teorema de Cook. Para $t > 1$ a fórmula

$$\phi_{q,s,t} = \exists r(\phi_{q,r,t/2} \wedge \phi_{r,s,t/2})$$

define se tem uma transição com estado intermediário r. Essa fórmula infelizmente tem t símbolos (que é demais para $2^{cf(n)}$), mas a fórmula

$$\phi_{q,s,t} = \exists r \forall (a,b) \in \{(q,r),(r,s)\}(\phi_{a,b,t/2})$$

evite a ocorrência dupla de ϕ a tem comprimento polinomial. ■

Outro exemplo

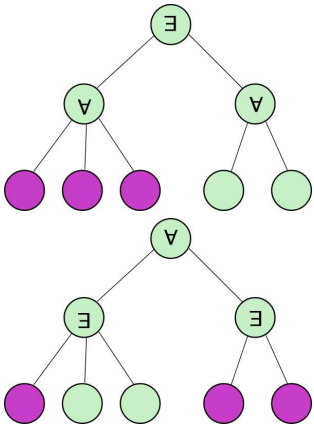
PALAVRA EM LINGUAGEM SENSÍVEL AO CONTEXTO

Instância Gramática Γ sensível ao contexto, palavra w.

Decisão $w \in L(\Gamma)$

Mais quantificações

O que acontece, se nós permitimos mais quantificações?



A hierarquia polinomial

- Estendemos relações para aridade $i+1$. Uma relação $R \subseteq (\Sigma^*)^{i+1}$ é *limitada polinomial*, se

$$\forall (x, y_1, \ldots, y_i) \in R \, \exists p \in \mathsf{poly} \, \forall i \, |y_i| \leq p(|x|)$$

- **Definição**: Σ_i é a classe das linguagens L, tal que existe uma relação de aridade $i+1$ que pode ser reconhecida em tempo polinomial, e

$$x \in L \iff \exists y_1 \forall y_2 \cdots Q_i : (x, y_1, \ldots, y_i) \in R$$

- **Definição**: Π_i é a classe das linguagens L, tal que existe uma relação de aridade $i+1$ que pode ser reconhecida em tempo polinomial, e

$$x \in L \iff \forall y_1 \exists y_2 \cdots Q_i : (x, y_1, \ldots, y_i) \in R$$

- As classes Σ_i e Π_i formam a *hierarquia polinomial*.
- Observação: $\Sigma_1 = \mathsf{NP}$, $\Pi_1 = \mathsf{co\text{-}NP}$.

Quantificações restritas ou não

- Conjunto das classes com quantificações restritas:

$$\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k$$

- Classe das linguagens reconhecidas por um máquina de Turing com alternações sem limite: APTIME
- As máquinas correspondentes são *máquinas de Turing com alternação* com tempo $t(n)$: $\mathsf{ATIME}[t(n)]$.

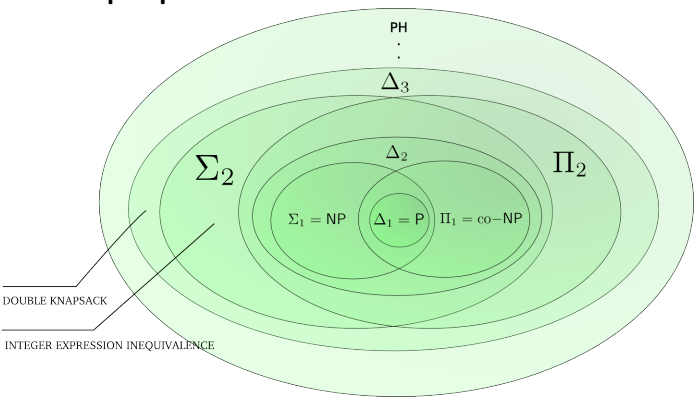
Teorema 12.2 (Chandra, Kozen, Stockmeyer)
Para $t(n) \geq n$

$$\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)] \subseteq \bigcup_{c>0} \mathsf{ATIME}[ct(n)^2].$$

Corolário 12.1
ATIME = PSPACE

- Esta caracterização facilite entender por que QBF é PSPACE-completo

A hierarquia polinomial



Mais exemplos da classe PSPACE

- Observação: Uma questão com alternação é típica para resolver jogos.
- Ganhar um jogo em um passo: "Existe um passo tal que possa ganhar?"
- Ganhar um jogo em dois passos: "Existe um passo, tal que para todos os passos do adversário, existe um passo tal que possa ganhar?"
- Ganhar um jogo:

$$\exists p_1\, \forall p_2\, \exists p_3\, \forall p_4 \cdots \exists p_{2k+1} :$$

$p_1, p_2, p_3, \ldots, p_{2k+1}$é uma seqüência de passos para ganhar.

- Portanto, vários jogos são PSPACE-completos: Generalized Hex, generalized Geography, ...

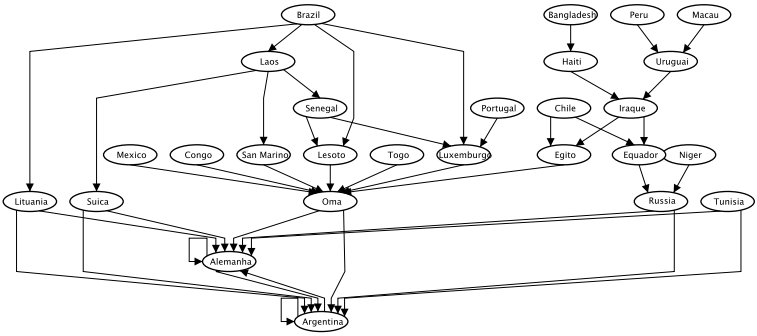
Mais exemplos da classe PSPACE **(2)**

Jogo de geografia para dois jogadores.

> Em alternação cada jogador diz o nome de um país. Cada nome tem que começar com a última letra do nome anterior. O primeiro jogador que não é capaz de dizer um novo país, perde.

Peru...

Mais exemplos da classe PSPACE (3)



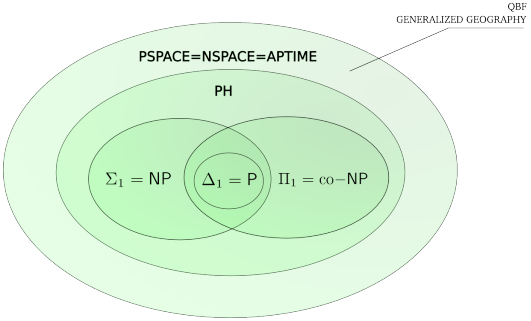
Geografia generalizada (ingl. Generalized geography)

Instância Um grafo $G = (V, E)$ e um nó $v_0 \in V$

Decisão Jogando "geografia"com este grafo, o primeiro jogador pode ganhar com certeza?

Teorema 12.3
Geografia generalizada é PSPACE-completo.

O mundo até PSPACE



12.2. De PSPACE até ELEMENTAR

Problemas intratáveis demonstráveis

- Agora, consideramos os seguintes classes

$$\mathsf{EXP} = \mathsf{DTIME}[2^{n^{O(1)}}] = \bigcup_{k \geq 0} \mathsf{DTIME}[2^{n^k}]$$

$$\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{O(1)}}] = \bigcup_{k \geq 0} \mathsf{NTIME}[2^{n^k}]$$

$$\mathsf{EXPSPACE} = \mathsf{DSPACE}[2^{n^{O(1)}}] = \bigcup_{k \geq 0} \mathsf{DSPACE}[2^{n^k}]$$

- Estas classes são as primeiras demonstravelmente separadas de P.
- Conseqüência: Uma linguagem completa em EXP não é tratável.
- Exemplo de um problema EXP-completo:

Xadrez generalizada (ingl. Generalized chess)

Instância Uma configuração de xadrez com tabuleiro de tamanho $n \times n$.

Decisão Branco pode forçar o ganho?

Problemas ainda mais intratáveis

- As classes $k - \mathsf{EXP}$, $k - \mathsf{NEXP}$ e $k - \mathsf{EXPSPACE}$ tem k níveis de exponenciação!
- Por exemplo, considere a torre de dois de altura três: $2^{2^{2^k}}$

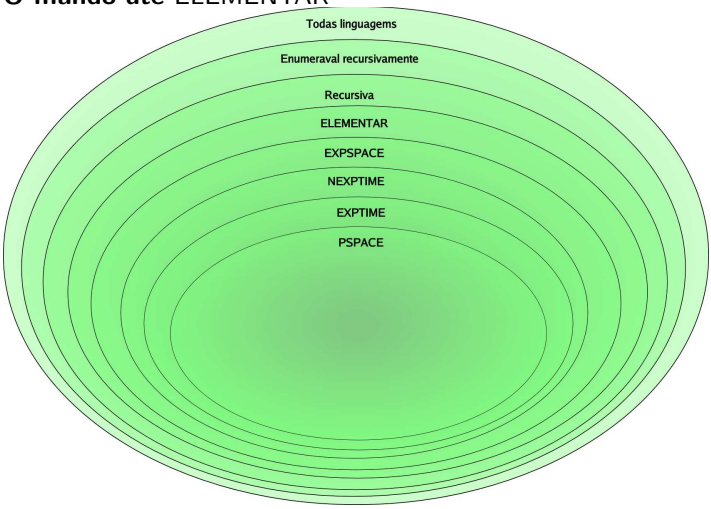
0	1	2	3
4	16	65536	$\approx 1.16 \times$

- Problemas desse tipo são *bem* intratáveis

$$\mathsf{ELEMENTAR} = \bigcup_{k \geq 0} k - \mathsf{EXP}$$

- Mas tem ainda problemas decidíveis fora desta classe!

O mundo até ELEMENTAR



Um corte final: Expressões regulares

- Uma expressão regular é
  - 0 ou 1 (denota o conjunto $L(0) = \{0\}$ e $L(1) = \{1\}$).
  - $e \circ f$, se $\circ$ é um operador, e e, f são expressões regulares.
- Operadores possíveis: $\cup$, $\cdot$, ^{2}, *, $\neg$.
- Decisão: Dadas as expressões regulares e, f, $L(e) \neq L(f)$?

Expressões regulare
$\cup, \cdot$
$\cup, \cdot, ^*$
$\cup, \cdot, ^2$
$\cup, \cdot, ^2, ^*$
$\cup, \cdot, \neg$

- O tempo do último problema de decisão cresce ao menos como uma torre de altura $\lg n$.

A. Conceitos matemáticos

Nessa seção vamos repetir algumas definições básicas da matemática.

A.1. Funções comuns

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ e $\mathbb{R}$ denotam os conjuntos dos números naturais sem 0, inteiros, racionais e reais, respectivamente. Escrevemos também $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, e para um dos conjuntos C acima, $C_+ := \{x \in C | x > 0\}$ e $C_- := \{x \in C | x < 0\}$. Por exemplo

$$\mathbb{R}_+ = \{x \in \mathbb{R} | x > 0\}.$$

Para um conjunto finito S, $\mathcal{P}(S)$ denota o conjunto de todos subconjuntos de S.

Definição A.1 (Valor absoluto)
O valor absoluta $|\cdot|$ é definido por

$$|x \models \begin{cases} x & \text{se } x \geq 0 \\ -x & \text{se } x < 0 \end{cases}$$

Proposição A.1 (Regras para valores absolutos)

$$\vdash x| = |x| \tag{A.1}$$
$$x \leq |x| \tag{A.2}$$
$$|x + y| \leq |x| + |y| \qquad \text{Desigualdade triangular} \tag{A.3}$$
$$|xy| = |x||y| \tag{A.4}$$

Prova. (i) Se $-x > 0$ temos $x < 0$, logo $\vdash x \models -x$ e $|x \models -x$. O casos restantes podem ser analisadas analogamente. (ii) Analise da casos. (iii) Para $x + y < 0$: $|x + y \models -(x + y) = (-x) + (-y) \leq \vdash x| + \vdash y \models |x| + |y|$. Para $x + y \geq 0$: $|x + y \models x + y \leq |x| + |y|$. (iv) Para $xy \geq 0$: Se $x = 0$ temos $|xy \models 0 = |x||y|$, se $x > 0$ temos $y > 0$ e $|xy \models xy = |x||y|$, se $x < 0$ temos $y < 0$ e $|xy \models xy = (-|x|)(-|y|) = |x||y|$. Caso $xy < 0$ similar. ■

Corolário A.1

$$\left|\sum_{1\leq i\leq n} x_i\right| \leq \sum_{1\leq i\leq n} |x_i| \tag{A.5}$$

$$\left|\prod_{1\leq i\leq n} x_i\right| = \prod_{1\leq i\leq n} |x_i| \tag{A.6}$$

$$\tag{A.7}$$

Prova. Prova com indução sobre n. ■

Proposição A.2 (Regras para o máximo)
Para $a_i, b_i \in \mathbb{R}$

$$\max_i a_i + b_i \leq \max_i a_i + \max_i b_i \tag{A.8}$$

Prova. Seja $a_k + b_k = \max_i a_i + b_i$. Logo

$$\max_i a_i + b_i = a_k + b_k \leq \left(\max_i a_i\right) + b_i \leq \max_i a_i + \max_i b_i.$$

■

Definição A.2 (Pisos e tetos)
Para $x \in \mathbb{R}$ o *piso* $\lfloor x \rfloor$ é o maior número inteiro menor que x e o *teto* $\lceil x \rceil$ é o menor número inteiro maior que x. Formalmente

$$\lfloor x \rfloor = \max\{y \in \mathbb{Z} | y \leq x\}$$
$$\lceil x \rceil = \min\{y \in \mathbb{Z} | y \geq x\}$$

O *parte fracionário* de x é $\{x\} = x - \lfloor x \rfloor$.

Observe que o parte fracionário sempre é positivo, por exemplo $\{-0.3\} = 0.7$.

Proposição A.3 (Regras para pisos e tetos)
Pisos e tetos satisfazem

$$x \leq \lceil x \rceil < x + 1 \tag{A.9}$$
$$x - 1 < \lfloor x \rfloor \leq x \tag{A.10}$$

Definição A.3
O *fatorial* é a função

$$n! : \mathbb{N} \to \mathbb{N} : n \mapsto \prod_{1 \leq i \leq n} i.$$

Temos a seguinte aproximação do fatorial (fórmula de Stirling)

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n)) \tag{A.11}$$

Uma estimativa menos preciso, pode ser obtido pelas observações

$$n! \leq n^n$$
$$e^n = \sum_{i \geq 0} \frac{n^i}{i!} > \frac{n^n}{n!}$$

que combinado ficam

$$(n/e)^n \leq n! \leq n^n.$$

Revisão: Logaritmos

$\log_a(1) = 0$		(A.12)
$a^{\log_a(n)} = n$	por definição	(A.13)
$\log_a(n \cdot m) = \log_a(n) + \log_a(m)$	propriedade do produto	(A.14)
$\log_a(\frac{n}{m}) = \log_a(n) - \log_a(m)$	propriedade da divisão	(A.15)
$\log_a(n^m) = m \cdot \log_a(n)$	propriedade da potência	(A.16)
$\log_a(n) = \log_b(n) \cdot \log_a(b)$	troca de base	(A.17)
$\log_a(n) = \frac{\log_c(n)}{\log_c(a)}$	mudança de base	(A.18)
$\log_b(a) = \frac{1}{\log_a(b)}$		(A.19)
$a^{\log_c(b)} = b^{\log_c(a)}$	expoentes	(A.20)

Os números harmônicos

$$H_n = \sum_{1 \leq i \leq n} \frac{1}{i}$$

ocorrem freqüentemente na análise de algoritmos.

Proposição A.4

$$\ln n < H_n < \ln n + 1.$$

Prova. Resultado da observação que

$$\int_1^{n+1} \frac{1}{x}\,\mathrm{d}x < H_n < 1 + \int_2^{n+1} \frac{1}{x-1}\,\mathrm{d}x$$

(veja figura A.1) e o fato que $\int 1/x = \ln x$. ■

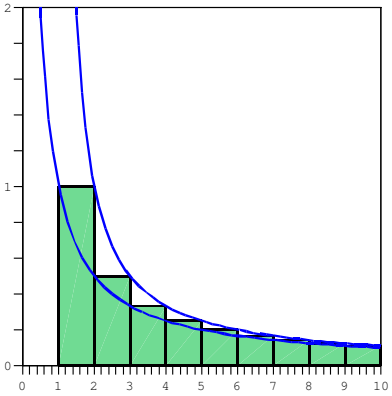


Figura A.1.: Cota inferior e superior dos números harmônicos.

A.2. Somatório

Revisão: Notação Somatório

Para k uma constante arbitrária temos

$$\sum_{i=1}^{n} ka_i = k\sum_{i=1}^{n} a_i \qquad \text{Distributividade} \tag{A.21}$$

$$\sum_{i=1}^{n} k = nk \tag{A.22}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m} a_i b_j = \left(\sum_{i=1}^{n} a_i\right)\left(\sum_{j=1}^{m} b_j\right) \qquad \text{Distributividade generalizada} \tag{A.23}$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \qquad \text{Associativiade} \tag{A.24}$$

$$\sum_{i=1}^{p} a_i + \sum_{i=p+1}^{n} a_i = \sum_{i=1}^{n} a_i \tag{A.25}$$

$$\sum_{i=0}^{n} a_{p-i} = \sum_{i=p-n}^{p} a_i \tag{A.26}$$

A última regra é um caso particular de troca de índice (ou comutação) para somas. Para um conjunto finito C e uma permutação dos números inteiros π temos

$$\sum_{i\in C} a_i = \sum_{\pi(i)\in C} a_{\pi(i)}.$$

No exemplo da regra acima, temos $C = [0, n]$ e $\pi(i) = p - i$ e logo

$$\sum_{0\le i\le n} a_{p-i} = \sum_{0\le p-i\le n} a_{p-(i-p)} = \sum_{p-n\le i\le p} a_i.$$

Parte da análise de algoritmos se faz usando somatórios, pois laços *while* e *for* em geral podem ser representados por somatórios. Como exemplo, considere o seguinte problema. Dadas duas matrizes $matA$ e $matB$, faça um algoritmo que copie a matriz triangular inferior de $matB$ para $matA$.

CopiaMTI

Entrada Matrizes quadráticos A e B e dimensão n.

Saída Matriz A com a matriz triangular inferior copiada de B.

```
1  for i := 1 to n do
2     for j := 1 to i do
3        A_ij = B_ij
4     end for
5  end for
```

Uma análise simples deste algoritmo seria:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2} = O(n^2)$$

Séries

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \text{série aritmética} \tag{A.27}$$

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} \qquad \text{série geométrica, para } x \neq 1 \tag{A.28}$$

se $|x| < 1$ então

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \text{série geométrica infinitamente decrescente} \tag{A.29}$$

(A.30)

Série geométrica com limites arbitrários:

$$\sum_{a \leq i \leq b} x^i = \frac{x^{b+1} - x^a}{x-1} \qquad \text{para } x \neq 1$$

Séries

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 2 \tag{A.31}$$

$$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6} \tag{A.32}$$

$$\sum_{i=0}^{n} i \cdot 2^i = 2 + (n-1) \cdot 2^{n+1} \tag{A.33}$$

Mais geral para alguma seqüência f_i temos

$$\begin{aligned}\sum_{1\le i\le n} i f_i &= \sum_{1\le i\le n}\sum_{1\le j\le i} f_i = \sum_{1\le j\le i\le n} f_i = \sum_{1\le j\le n}\sum_{j\le i\le n} f_i \\ &= \sum_{1\le j\le n}\left(\sum_{1\le i\le n} f_i - \sum_{1\le i<j} f_i\right).\end{aligned}$$

Uma aplicação:

$$\begin{aligned}\sum_{1\le i\le n} i x^i &= \sum_{1\le j\le n}\left(\sum_{1\le i\le n} x^i - \sum_{1\le i<j} x^i\right) = \sum_{1\le j\le n}\left(\frac{x^{n+1}-x^1}{x-1} - \frac{x^j - x^1}{x-1}\right) \\ &= \frac{1}{x-1}\sum_{1\le j\le n}\left(x^{n+1} - x^j\right) \\ &= \frac{1}{x-1}\left(nx^{n+1} - \frac{x^{n+1}-x^1}{x-1}\right) = \frac{x}{(x-1)^2}\left(x^n(nx-n-1)+1\right)\end{aligned}$$

e com $x = 1/2$ temos

$$\sum_{1\le i\le n} i2^{-i} = 2(2(2^{-1}-2^{-n-1})-n2^{-n-1}) = 2((1-2^{-n})-n2^{-n-1}) = 2-2^{-n}(n+2) \tag{A.34}$$

A.3. Indução

Revisão: Indução matemática

- Importante para provar resultados envolvendo inteiros.

- Seja P(n) uma propriedade relativa aos inteiros.
  - Se P(n) é verdadeira para n=1 e
  - se P(k) verdadeira implica que P(k+1) é verdadeira
  - então P(n) é verdadeira para todo inteiro $n \geq 1$.

Revisão: Indução matemática

- Para aplicarmos indução matemática deve-se:
  - Passo inicial: verificar se P(n) é verdadeira para a base n_0.
  - Hipótese: assumir P(n) válida.
  - Prova: provar que P(n) é válida para qualquer valor de $n \geq n_0$.
- Se os passos acima forem verificados, conclui-se que P(n) é valida para qualquer valor de $n \geq n_0$

Indução matemática: exercícios

- Mostre que $n! \leq n^n$.
- Mostre que $\frac{1}{\log_a(c)} = \log_c(a)$.
- Demonstre a propriedade dos expoentes.
- Encontre uma fórmula alternativa para

$$\sum_{i=1}^{n} 2i - 1$$

  e prove seu resultado via indução matemática.
- Use indução matemática para provar que

$$\sum_{i=0}^{n-1} q^i = \frac{q^n - 1}{q - 1}.$$

- Resolva os exercícios do capítulo 1.

A.4. Limites

Definição A.4 (Limites)
Para $f : \mathbb{N} \to \mathbb{R}$ o limite de n para ∞ é definido por

$$\lim_{n\to\infty} f(n) = c \iff \exists c\, \forall \epsilon > 0\, \exists n_0\, \forall n > n_0\, |f(n) - c| < \epsilon. \tag{A.35}$$

Caso não existe um $c \in \mathbb{R}$ a função é *divergente*. Uma forma especial de divergência é quando a função ultrapasse qualquer número real,

$$\lim_{n\to\infty} f(n) = \infty \iff \forall c\, \exists n_0\, \forall n > n_0\, f(n) > c \tag{A.36}$$

Também temos

$$\liminf_{n\to\infty} f(n) = \lim_{n\to\infty} \left(\inf_{m\geq n} f(m) \right)$$
$$\limsup_{n\to\infty} f(n) = \lim_{n\to\infty} \left(\sup_{m\geq n} f(m) \right)$$

Lema A.1 (Definição alternativa do limite)
É possível substituir $<$ com $\leq$ na definição do limite.

$$\lim_{n\to\infty} f(n) = c \iff \forall \epsilon > 0\, \exists n_0\, \forall n > n_0\, |f(n) - c| \leq \epsilon$$

Prova. $\Rightarrow$ é obvio. Para $\Leftarrow$, escolhe $\epsilon' = \epsilon/2 < \epsilon$. ■

A.5. Probabilidade discreta

Probabilidade: Noções básicas

- *Espaço amostral* finito Ω de *eventos elementares* $e \in \Omega$.
- Distribuição de probabilidade $\Pr[e]$ tal que

$$\Pr[e] \geq 0; \qquad \sum_{e\in\Omega} \Pr[e] = 1$$

- *Eventos* (compostos) $E \subseteq \Omega$ com probabilidade

$$\Pr[E] = \sum_{e\in E} \Pr[e]$$

Exemplo A.1
Para um dado sem bias temos $\Omega = \{1,2,3,4,5,6\}$ e $\Pr[i] = 1/6$. O evento $\text{Par} = \{2,4,6\}$ tem probabilidade $\Pr[\text{Par}] = \sum_{e\in\text{Par}} \Pr[e] = 1/2$. ◊

Probabilidade: Noções básicas

- *Variável aleatória*
$$X : \Omega \to \mathbb{N}$$
- Escrevemos $\Pr[X = i]$ para $\Pr[X^{-1}(i)]$.
- Variáveis aleatórias *independentes*
$$P[X = x \text{ e } Y = y] = P[X = x]P[Y = y]$$
- *Valor esperado*
$$E[X] = \sum_{e\in\Omega} \Pr[e]X(e) = \sum_{i\geq 0} i \Pr[X = i]$$
- Linearidade do valor esperado: Para variáveis aleatórias X, Y
$$E[X + Y] = E[X] + E[Y]$$

Prova. (Das formulas equivalentes para o valor esperado.)

$$\begin{aligned}\sum_{0\leq i} \Pr[X = i]i &= \sum_{0\leq i} \Pr[X^{-1}(i)]i \\ &= \sum_{0\leq i} \sum_{e\in X^{-1}(i)} \Pr[e]X(e) = \sum_{e\in\Omega} \Pr[e]X(e)\end{aligned}$$

■

Prova. (Da linearidade.)

$$\begin{aligned}E[X + Y] &= \sum_{e\in\Omega} Pr[e](X(e) + Y(e)) \\ &= \sum_{e\in\Omega} Pr[e]X(e) \sum_{e\in\Omega} Pr[e]Y(e)) = E[X] + E[Y]\end{aligned}$$

■

Exemplo A.2
(Continuando exemplo A.1.)
Seja X a variável aleatório que denota o número sorteado, e Y a variável aleatório tal que $Y =$ [a face em cima do dado tem um ponto no meio].

$$E[X] = \sum_{i \geq 0} \Pr[X = i]i = 1/6 \sum_{1 \leq i \leq 6} i = 21/6 = 7/2$$
$$E[Y] = \sum_{i \geq 0} \Pr[Y = i]i = \Pr[Y = 1] = 1/2E[X + Y] \quad = E[X] + E[Y] = 4$$

◊

A.6. Grafos

Seja $[D]^k$ o conjunto de todos subconjuntos de tamanho k de D.
Um *grafo* (ou grafo não-direcionado) é um par $G = (V, E)$ de *vértices* (ou nós ou pontos) V e *arestas* (ou arcos ou linhas) E tal que $E \subseteq [V]^2$. Com $|G|$ e $||G||$ denotamos o número de vértices e arestas, respectivamente. Dois vértices u, v são *adjacentes*, se $\{u, v\} \in E$, duas arestas e, f são adjacentes, se $e \cap f \neq \emptyset$. Para um vértice v, a *vizinhança* (de vértices) $N(v)$ é o conjunto de todas vértices adjacentes com ele, e a vizinhança (de arestas) $E(v)$ é o conjunto de todas arestas adjacentes com ele. O *grau* de um vértice v é o número de vizinhos $\delta(v) = |N(v)| = |E(v)|$.
Um *caminho* de comprimento k é um grafo $C = (\{v_0, \ldots, v_k\}, \{\{v_i, v_{i+1}\} \,|\, 0 \leq i < k\})$ com todo v_i diferente. Um ciclo de comprimento $k + 1$ é um caminho com a aresta adicional $\{v_n, v_0\}$. O caminho com comprimento k é denotado com P^k, o ciclo de comprimento k com C^k.
Um grafo G é *conexo* se para todo par de vértices u, v existe um caminho entre eles em G.
Um *subgrafo* de G é um grafo $G' = (V', E')$ tal que $V' \subseteq V$ e $E' \subseteq E$, escrito $G' \subseteq G$. Caso G' contém todas arestas entre vértices em V' (i.e. $E' = E \cap [V']^2$) ela é um *subgrafo induzido* de V' em G, escrito $G' = G[V']$.
Um *grafo direcionado* é um par $G = (V, E)$ de vértices V e arestas $E \subseteq V^2$. Cada aresta $e = (u, v)$ tem um *começo* u e um *termino* v.

B. Soluções dos exercícios

Solução do exercício 1.1.
As características correspondentes são

$$f = \Omega(f) \tag{B.1}$$
$$c\Omega(f) = \Omega(f) \tag{B.2}$$
$$\Omega(f) + \Omega(f) = \Omega(f) \tag{B.3}$$
$$\Omega(\Omega(f)) = \Omega(f) \tag{B.4}$$
$$\Omega(f)\Omega(g) = \Omega(fg) \tag{B.5}$$
$$\Omega(fg) = f\Omega(g) \tag{B.6}$$

Todas as características se aplicam para Ω também. As provas são modificações simples das provas das características 1.10 até 1.15 com $\leq$ substituído por $\geq$.
Prova.
Prova de B.1: Escolhe $c = 1$, $n_0 = 0$.
Prova de B.2: Se $g \in c\Omega(f)$, temos $g = cg'$ e existem $c' > 0$ e n_0 tal que $\forall n > n_0\ g' \geq c'f$. Portanto $\forall n > n_0\ g = cg' \geq cc'f$ e com cc' e n_0 temos $g \in \Omega(f)$.
Prova de B.3: Para $g \in \Omega(f) + \Omega(f)$ temos $g = h + h'$ com $c > 0$ e n_0 tal que $\forall n > n_0\ h \geq cf$ e $c' > 0$ e n_0' tal que $\forall n > n_0\ h' \geq c'f$. Logo para $n > \max(n_0, n_0')$ temos $g = h + h' \geq (c + c')f$.
Prova de B.4: Para $g \in \Omega(\Omega(f))$ temos $g \geq ch$ com $h \geq c'f$ a partir de índices n_0 e n_0', e logo $g \geq cc'h$ a partir de $\max(n_0, n_0')$.
Prova de B.5: $h = f'g'$ com $f' \geq c_f f$ e $g' \geq c_g g$ tal que $h = f'g' \geq c_f c_g fg$.
Prova de B.6: $h \geq cfg$. Escrevendo $h = fg'$ temos que mostrar $g' \in \Omega(g)$. Mas $g' = h/f \geq cfg/f = cg$. ■

Solução do exercício 1.2.
"$\Leftarrow$":
Seja $f + c \in O(g)$, logo existem c' e n_0 tal que $\forall n > n_0\ f + c \leq c'g$. Portanto $f \leq f + c \leq c'g$ também, e temos $f \in O(g)$.
"$\Rightarrow$":
Essa direção no caso geral não é válida. Um contra-exemplo simples é $0 \in O(0)$ mas $0 + c \notin O(0)$. O problema é que a função g pode ser 0 um número infinito

de vezes. Assim f tem que ser 0 nesses pontos também, mas $f + c$ não é. Mas com a restrição que $g \in \Omega(1)$, temos uma prova:
Seja $f \in O(g)$ logo existem c' e n_0' tal que $\forall n > n_0'\ f \leq c'g$. Como $g \in \Omega(1)$ também existem c'' e n_0'' tal que $\forall n > n_0''\ g \geq c''$. Logo para $n > \max(n_0', n_0'')$

$$f + c \leq c'g + c \leq c'g + \frac{c}{c''}g = (c' + \frac{c}{c''})g.$$

Solução do exercício 1.3.

1. Para $n \geq 2$ temos $\log 1 + n \leq \log 2n = \log 2 + \log n \leq 2 \log n$.
2. Seja $f \in \log O(n^2)$, i.e. $f = \log g$ com g tal que $\exists n_0, c\, \forall n > n_0\ g \leq cn^2$. Então $f = \log g \leq \log cn^2 = \log c + 2 \log n \leq 3 \log n$ para $n > \max(c, n_0)$.
3. Temos que mostrar que existem c e n_0 tal que $\forall n > n_0 \log \log n \leq c \log n$. Como $\log n \leq n$ para todos $n \geq 1$ a inequação acima está correto com $c = 1$.

Solução do exercício 1.4.
Para provar $f_n = O(n)$ temos que provar que existe um c tal que $f_n \leq cn$ a partir um ponto n_0. É importante que a constante c é a mesma para todo n. Na verificação do professor Veloz a constante c muda implicitamente, e por isso ela não é válida. Ele tem que provar que $f_n \leq cn$ para algum c fixo. Uma tentativa leva a

$$\begin{aligned} f_n &= 2f_{n-1} \\ &\leq 2cn \\ &\not\leq cn \quad \text{Perdido!} \end{aligned}$$

que mostra que essa prova não funciona.

Solução do exercício 1.5.
E simples ver que $f \in \hat{o}(g)$ implica $f \in o(g)$. Para mostrar a outra direção suponha que $f \in o(g)$. Temos que mostrar que $\forall c > 0 : \exists n_0$ tal que $f < cg$. Escolhe um c. Como $f \in o(g)$ sabemos que existe um n_0 tal que $f \leq c/2g$ para $n > n_0$. Se $g \neq 0$ para $n > n_0'$ então $c/2g < g$ também. Logo $f \leq c/2g < cg$ para $n > \max(n_0, n_0')$.

Solução do exercício 1.6.
Primeira verifique-se que Φ satisfaz $\Phi + 1 = \Phi^2$.

Prova que $f_n \in O(\Phi^n)$ com indução que $f_n \leq c\Phi^n$. Base: $f_0 = 0 \leq c$ e $f_1 = 1 \leq c\Phi$ para $c \geq 1/\Phi \approx 0.62$. Passo:

$$f_n = f_{n-1} + f_{n-2} \leq c\Phi^{n-1} + c\Phi^{n-2} = (c\Phi + c)\Phi^{n-2} \leq c\Phi^n$$

caso $c\Phi + c \leq c\Phi^2$.
Prova que $f_n \in \Omega(\Phi^n)$ com indução que $f_n \geq c\Phi^n$. Base: Vamos escolher $n_0 = 1$. $f_1 = 1 \geq c\Phi$ e $f_2 = 1 \geq c\Phi^2$ caso $c \leq \Phi^{-2} \approx 0.38$. Passo:

$$f_n = f_{n-1} + f_{n-2} \geq c\Phi^{n-1} + c\Phi^{n-2} = (c\Phi + c)\Phi^{n-2} \leq c\Phi^n$$

caso $c\Phi + c \geq c\Phi^2$.

Solução do exercício [27, 2.3].

1. $3n + 7 \leq 5n + 2 \iff 5 \leq 2n \iff 2.5 \leq n$ (equação linear)
2. $5n + 7 \leq 3n^2 + 1 \iff 0 \leq 3n^2 - 5n - 6 \iff 5/6 + \sqrt{97}/6 \leq n$ (equação quadrática)
3. $5 \log_2 n + 7 \leq 5n + 1 \iff 7^5 + 2^7 - 2 \leq 2^{5n} \iff 16933 \leq 2^{5n} \iff 2.809\ldots \leq n$
4. Veja item (b)
5. $52^n + 3 \geq 3n^2 + 5n \iff n \geq 2^n \geq (3n^2 + 5n - 3)/5 \Leftarrow 2^n \geq n^2$.
6. $n^2 3^n \geq n^3 2^n + 1 \Leftarrow n^2 3^n \geq n^3 2^{n+1} \iff 2 \log_2 n + n \log_2 3 \geq 3 \log_2 n + (n+1) \log_2 2 \iff n \log_2 3 \geq \log_2 n + (n+1) \Leftarrow n(\log_2 3 - 1)/2 \geq \log_2 n$

Solução do exercício [27, 2.9].
Com $f \in \Theta(n^r)$ e $g \in \Theta(n^s)$ temos

$$c_1 n^r \leq f \leq c_2 n^r; \quad d_1 n^s \leq g \leq d_2 n^s \qquad \text{a partir de um } n_0$$

(para constantes c_1, c_2, d_1, d_2.) Logo

$$\begin{aligned} & d_1 f^q \leq g \circ f \leq d_2 f^q \\ & \Rightarrow d_1 (c_1 n^p)^q \leq g \leq d_2 (c_2 n^p)^q \\ & \Rightarrow f_1 c_1^q n^{p+q} \leq g \leq d_2 c_2^q n^{p+q} \\ & \Rightarrow g \in \Theta(n^{p+q}) \end{aligned}$$

Solução do exercício 2.1.

$$C_p[\text{Alg1}] = \sum_{i=1}^{n} \sum_{j=1}^{2^{i-1}} c = \frac{c}{2} \cdot \sum_{i=1}^{n} 2^i = c \cdot 2^n - c = O(2^n)$$

$$\begin{aligned} C_p[\text{Alg2}] &= \sum_{1 \le i \le n} \sum_{\substack{1 \le j \le 2^i \\ j \text{ ímpar}}} j^2 \le \sum_{1 \le i \le n} \sum_{1 \le j \le 2^i} (2^i)^2 \\ &= \sum_{1 \le i \le n} 8^i = \frac{8^{n+1} - 8}{7} \le 8^{n+1} = O(8^n) \end{aligned}$$

$$\begin{aligned} C_p[\text{Alg3}] &= \sum_{i=1}^{n} \sum_{j=i}^{n} 2^i = \sum_{i=1}^{n} 2^i \cdot (n - i + 1) \\ &= \sum_{i=1}^{n} (n2^i - i2^i + 2^i) = \sum_{i=1}^{n} n.2^i - \sum_{i=1}^{n} i.2^i + \sum_{i=1}^{n} 2^i \\ &= n \cdot (2^{n+1} - 2) - (2 + (n-1) \cdot 2^{n+1}) + (2^{n+1} - 2) \\ &= n2^{n+1} - 2n - 2 - n2^{n+1} + 2^{n+1} + 2^{n+1} - 2 \\ &= 2^{n+2} - 2n - 4 = O(2^n) \end{aligned}$$

$$\begin{aligned} C_p[\text{Alg4}] &= \sum_{i=1}^{n} \sum_{j=1}^{i} 2^j = \sum_{i=1}^{n} (2^{i+1} - 2) \\ &= 2 \sum_{i=1}^{n} 2^i - \sum_{i=1}^{n} 2 = 2 \cdot (2^{n+1} - 2) - 2n \\ &= 4 \cdot 2^n - 4 - 2n = O(2^n) \end{aligned}$$

$$
\begin{aligned}
C_p[\text{Alg5}] &= \sum_{i=1}^{n}\sum_{j=i}^{n} 2^j = \sum_{i=1}^{n}\left(\sum_{j=1}^{n} 2^j - \sum_{j=1}^{i-1} 2^j\right) \\
&= \sum_{i=1}^{n}\left(2^{n+1} - 2 - (2^{i-1+1} - 2)\right) = \sum_{i=1}^{n}\left(2.2^n - 2 - 2^i + 2\right) \\
&= 2\cdot\sum_{i=1}^{n} 2^n - \sum_{i=1}^{n} 2^i = 2.n.2^n - (2^{n+1} - 2) \\
&= 2.n.2^n - 2.2^n + 2 = O(n2^n)
\end{aligned}
$$

Solução do exercício 2.2.
O problema é o mesmo da prova do exercício 1.4: Na prova a constante c muda implicitamente. Para provar $T_n = O(n)$ temos que provar $T_n \leq cn$ para c fixo. Essa prova vira

$$
\begin{aligned}
T_n &= n - 1 + 2/n \sum_{0\leq i<n} O(i) \\
&\leq n - 1 + 2c/n \sum_{0\leq i<n} i \\
&= n - 1 + c(n-1) = cn + (n - 1 - c) \\
&\not\leq cn \qquad \text{Não funciona para } n > c + 1
\end{aligned}
$$

Solução do exercício 2.3.
Uma solução simples é manter um máximo M e o segundo maior elemento m no mesmo tempo:

```
1  M := ∞
2  m := ∞
3  for i = 1, ..., n do
4     if a_i > M then
5        m := M
6        M := a_i
7     else if a_i > m do
8        m := a_i
```

```
    end if
end for
return m
```

O número de comparações é ao máximo dois por iteração, e esse limite ocorre numa seqüência crescendo $1, 2, \ldots, n$. Portanto, a complexidade pessimista é $2n = \Theta(n)$. Existem outras soluções que encontram o segundo maior elemento com somente $n + \log_2 n$ comparações.

Solução do exercício 2.4.
Uma abordagem simples com busca exaustiva é

```
m := ∑_{1≤i≤n} a_i
for C ⊆ [1, n] do
    m' := |∑_{i∈C} a_i − ∑_{i∉C} a_i|
    if m' < m then
        m := m'
    end if
end for
```

Ele tem complexidade $c_p = O(n) + O(2^n nc) = O(n2^n)$.

Solução do exercício 2.5.
Para um dado n temos sempre $n - \lfloor n/2 \rfloor$ atualizações. Logo, o número médio de atualizações é e mesma.

Solução do exercício 2.6.
Seja $A, A_1, \ldots, A_n$ as variáveis aleatórias que denotam o número total de atualizações, e o número de atualizações devido a posição i, respectivamente. Com a distribuição uniforme temos $E[A_i] = 1/6$ e pela linearidade

$$E[A] = E\left[\sum_{1 \le i \le n} A_i\right] = n/6.$$

Com o mesmo argumento a segunda distribuição leva a $E[A_i] = 1/10$ é $E[A] = n/10$ finalmente.

Solução do exercício 2.7.
Cada chave em nível $i \in [1, k]$ precisa i comparações e a árvore tem $\sum_{1 \le i \le k} 2^{i-1} = 2^k - 1$ nós e folhas em total. Para o número de comparações C temos

$$E[C] = \sum_{1 \le i \le k} P[C = i]i = \sum_{1 \le i \le k} \frac{2^{i-1}}{2^{k-1}} i = 2^{-k} \sum_{1 \le i \le k} 2^i i = 2(k-1) + 2^{1-k}.$$

Solução do exercício 4.1.
O seguinte algoritmo resolva o problema:

SUBSEQÜÊNCIA

Entrada Seqüência $S' = s'_1 \dots s'_m$ e $S = s_1 \dots s_n$.

Saída `true`, se $S' \subseteq S$ (S' é uma subseqüência de S)

```
1   if m > n then
2     return false
3   end if
4   i := 1
5   for j := 1, ..., n do
6     if s'_i = s_j then
7       i := i + 1
8       if i > m then
9         return true
10      end if
11    end if
12  end for
13  return false
```

e tem complexidade $O(n)$. A corretude resulta de observação que para cada subseqüência possível temos outra subseqüência que escolhe o elemento mais esquerda em S. Portanto, podemos sempre escolher gulosamente o primeiro elemento da seqüência maior.

Solução do exercício 4.2.
O seguinte algoritmo resolve o problema:

BASES

Entrada Uma seqüência de posições x_i de n cidades, $1 \leq i \leq n$.

Saída Uma seqüência mínima de posições b_i de bases.

```
1   Sejam S = x'_1 ... x'_n as posições em ordem crescente
2   B = ϵ
```

```
3        while S ≠ ∅ do
4          Seja S = x'S'
5          B := B, (x' + 4)  { aumenta a seqüência B }
6          Remove todos os elementos x ≤ x' + 8 de S
7        end while
```

O algoritmo tem complexidade $O(n)$ porque o laço tem ao máximo n iterações. Prova de corretude: Seja b_i as posições do algoritmo guloso acima, e b'_i as posições de alguma outra solução. Afirmação: $b_i \geq b'_i$. Portanto, a solução gulosa não contém mais bases que alguma outra solução. Prova da afirmação com indução: A base $b_1 \geq b'_1$ é correto porque toda solução tem que alimentar a primeira casa e o algoritmo guloso escolhe a última posição possível. Passo: Seja $b_i \geq b'_i$ e sejam h, h' as posições da próximas casas sem base. O algoritmo guloso escolha $h + 4$, mas como $b_i \geq b'_i$ e $h \geq h'$ temos $b'_{i+1} \leq h' + 4$ porque h' precisa uma base. Logo, $x_{i+1} = h + 4 \geq h' + 4 \geq b'_{i+1}$.

Solução do exercício 6.3.

1. Produto de dois números binários (exemplo 5.3.8 em [27]).

MULT-BIN

Entrada Dois números binários p, q com n bits.

Saída O produto $r = pq$ (que tem $\leq 2n$ bits).

```
1   if n = 1 then
2     return pq      { multiplica dois bits em O(1) }
3   else
4     x := p1 + p2
5     y := q1 + q2
6     z := MULT-BIN(x2, y2)
7     t := (x1y1)2^n + (x1y2 + x2y1)2^(n/2) + z
8     u := MULT-BIN(p1, q1)
9     v := MULT-BIN(p2, q2)
10    r := u2^n + (t − u − v)2^(n/2) + v
11    return r
12  end if
```

É importante de observar que x é a soma de dois números com $n/2$ bits e logo tem no máximo $n/2+1$ bits. A divisão de x em x_1 e x_2 é tal que x_1 represente o bit $n/2+1$ e x_2 o resto.

$$
\begin{aligned}
p &= \underbrace{\vert \quad p_1 \quad \vert}_{n/2\text{bits}} \underbrace{\quad p_2 \quad \vert}_{n/2\text{bits}} \\
x &= \underbrace{\vert \quad p_1 \quad \vert}_{n/2\text{bits}} \\
&+ \underbrace{\vert \quad p_2 \quad \vert}_{n/2\text{bits}} \\
&= \underbrace{\vert x_1 \vert}_{1\text{bit}} \underbrace{\quad x_2 \quad \vert}_{n/2\text{bits}}
\end{aligned}
$$

(y tem a mesma subdivisão.)

Corretude do algoritmo Temos a representação $p = p_1 2^{n/2} + p_2$ e $q = q_1 2^{n/2} + q_2$ e logo obtemos o produto

$$pq = (p_1 2^{n/2} + p_2)(q_1 2^{n/2} + q_2) = p_1 q_1 2^n + (p_1 q_2 + p_2 q_1) 2^{n/2} + p_2 q_2 \quad \text{(B.7)}$$

Usando $t = (p_1 + p_2)(q_1 + q_2) = p_1 q_1 + p_1 q_2 + p_2 q_1 + p_2 q_2$ obtemos

$$p_1 q_2 + p_2 q_1 = t - p_1 q_1 - p_2 q_2. \quad \text{(B.8)}$$

A linha 7 do algoritmo calcula t (usando uma chamada recursiva com $n/2$ bits para obter $z = x_2 y_2$). Com os produtos $u = p_1 q_1$, $v = p_2 q_2$ (que foram obtidos com duas chamadas recursivas com $n/2$ bits) temos

$$
\begin{aligned}
p_1 q_2 + p_2 q_1 &= t - u - v \\
pq &= u 2^n + (t - u - v) 2^{n/2} + v
\end{aligned}
$$

substituindo u e v nas equações B.7 e B.8.

Complexidade do algoritmo Inicialmente provaremos pelo método da substituição que a recorrência deste algoritmo é $\mathrm{O}(n^{\log_2 3})$. Se usarmos a hipótese de que $T(n) \leq c n^{\log_2 3}$, não conseguiremos finalizar a prova pois permanecerá um fator adicional que não podemos remover da equação.

Caso este fator adicional for menor em ordem que a complexidade que queremos provar, podemos usar uma hipótese mais forte como apresentado abaixo.

Hipótese: $T(n) \leq cn^{\log_2 3} - dn$

$$\begin{aligned} T(n) &\leq 3(c(n/2)^{\log_2 3} - d(n/2)) + bn \\ &\leq 3cn^{\log_2 3}/(2^{\log_2 3}) - 3d(n/2) + bn \\ &\leq cn^{\log_2 3} - 3d(n/2) + bn \\ &\leq cn^{\log_2 3} - dn \end{aligned}$$

A desigualdade acima é verdadeira para $-3d(n/2) + bn \leq -dn$, ou seja, para $d \leq -2b$: $T(n) \in O(n^{\log_2 3} - dn) \in O(n^{\log_2 3})$.

Com a hipótese que uma *multiplicação com* 2^k *precisa tempo constante* (shift left), a linha 7 do algoritmo também precisa tempo constante, porque uma multiplicação com x_1 ou y_1 precisa tempo constante (de fato é um `if`). O custo das adições é $O(n)$ e temos a recorrência

$$T_n = \begin{cases} 1 & \text{se } n = 1 \\ 3T(n/2) + cn & \text{se } n > 1 \end{cases}$$

cuja solução é $\Theta(n^{1.58})$ (com $1.58 \approx \log_2 3$, aplica o teorema Master).

Exemplo B.1
Com $p = (1101.1100)_2 = (220)_{10}$ e $q = (1001.0010)_2 = (146)_{10}$ temos $n = 8$, $x = p_1 + p_2 = 1.1001$ tal que $x_1 = 1$ e $x_2 = 1001$ e $y = q_1 + q_2 = 1011$ tal que $y_1 = 0$ e $y_2 = 1011$. Logo $z = x_2 y_2 = 0110.0011$, $t = y_2 2^4 + z = 21.0001.0011$, $u = p_1 q_1 = 0111.0101$, $v = p_2 q_2 = 0001.1000$ e finalmente $r = 1111.1010.111.1000 = 32120$. ◊

O algoritmo acima não é limitado para números binários, ele pode ser aplicado para números com base arbitrário. Ele é conhecido como algoritmo de Karatsuba [16]. Um algoritmo mais eficiente é do Schönhage e Strassen [24] que multiplica em $O(n \log n \log\log n)$. Em 2007, Fürer [10] apresentou um algoritmo que multiplica em $n \log n\, 2^{O(\log^* n)}$, um pouco acima do limite inferior $\Omega(n \log n)$.

2. Algoritmo de Strassen para multiplicação de matrizes.

O algoritmo está descrito na seção 6.3. A recorrência correspondente é

$$T(n) = 7T(n/2) + \Theta(n^2).$$

Analisando com a árvore de recorrência, obtemos 7^i problemas em cada nível, cada um com tamanho $n/2^i$ e custo $c(n/2^i)^2 = cn^2/4^i$ e altura $h = \lceil \log_2 n \rceil$ (com $h+1$ níveis) que leva a soma

$$\begin{aligned} T(n) &\leq \sum_{0 \leq i \leq h} cn^2(7/4)^i + 7^{h+1} \\ &= (4/3)cn^2((7/4)^{h+1} - 1) + 7^{h+1} \qquad \text{com } 4^{h+1} \geq 4n^2 \\ &\leq (7c/3 + 1)7^h - (4/3)cn^2 \qquad \text{com } 7^h \leq 7\,7^{\log_2 n} \\ &\leq (49c/3 + 1)n^{\log_2 7} = O(n^{\log_2 7}). \end{aligned}$$

Para aplicar o método de substituição, podemos estimar $T(n) \leq an^c - bn^2$ com $c = \log_2 7$ que resulta em

$$\begin{aligned} T(n) &= 7T(n/2) + dn^2 \\ &\leq 7a/2^c n^c - 7b/4n^2 + dn^2 \\ &= an^c - bn^2 + (d - 3b/4)n^2 \end{aligned}$$

que é satisfeito para $d - 3/4 \leq 0 \Leftrightarrow b \geq (4/3)d$.

Para aplicar o método Master, é suficiente de verificar que com $\Theta(n^2) = O(n^{\log_2 7 - \epsilon})$ se aplica o caso 1, e portanto a complexidade é $\Theta(n^{\log_2 7})$.

3. Algoritmo de seleção do k-gésimo elemento.

   Esse algoritmo obedece a recorrência

$$T(n) = T(n/5) + T(7n/10 + 6) + O(n)$$

   (sem considerar o teto). Na aplicação da árvore de recorrência, enfrentamos dois problemas: (i) Os ramos tem comprimento diferente, porque os subproblemas tem tamanho diferente (portanto o método Master não se aplica nesse caso). (ii) O tamanho $7n/10+6$ do segundo subproblema leva a somas difíceis.

   Por isso, vamos estimar o custo da árvore da seguinte forma: (i) Temos que garantir, que o segundo subproblema sempre é menor: $7n/10+6 < n$. Isso é satisfeito para $n > 20$. (ii) Vamos substituir o sub-problema $7n/10 + 6$ com a cota superior $(7 + \epsilon)n/10$ para um $\epsilon > 0$ pequeno. Isso é satisfeito para $n \geq 60/\epsilon$. (iii) Sejam $c_1 := 1/5$, $c_2 := (7 + \epsilon)/10$ e

$c := c_1 + c_2$. Então a árvore tem custo $c^i n$ no nível i e no ramo mais longo (que corresponde a c_2) uma altura de $h = \lceil \log_{c_2} 20/n \rceil$. Portanto, obtemos uma cota superior para o custo da árvore

$$\begin{aligned} T(n) &\leq n \sum_{0 \leq i \leq h} c^i + F(n) \\ &\leq n \sum_{0 \leq i < \infty} c^i + F(n) \quad \text{porque } c < 1 \quad = 10n/(1-\epsilon) + F(n) \end{aligned}$$

com o número de folhas $F(n)$. Caso $F(n) = O(n)$ obtemos a estimativa desejada $T(n) = O(n)$. Observe que a estimativa

$$F(n) = 2^{h+1} \leq 42^{\log_{c_2} 20} n^{\log_{1/c_2} 2} = \Omega(n^{1.94})$$

não serve! Como as folhas satisfazem a recorrência

$$F(n) \leq \begin{cases} F(\lceil n/5 \rceil) + F(\lfloor 7n/10 + 6 \rfloor) & \text{se } n > 20 \\ O(1) & \text{se } n \leq 20 \end{cases}$$

$F(n) \leq cn$ pode ser verificado com substituição (resolvido no livro do Cormen). O método master não se aplica nesta recorrência visto que esta não se encontra no formato em que podemos aplicar o Teorema Master.

Bibliografia

[1] Wilkinson microwave anisotropy probe. Online. http://map.gsfc.nasa.gov.

[2] S. Aaronson. NP-complete problems and physical reality. *ACM SIGACT News*, Mar. 2005.

[3] M. J. Atallah, editor. *Algorithms and theory of computation handbook*. CRC Press, 1999.

[4] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, 1962.

[5] Complexity zoo. Online.

[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, 1987.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] R. Diestel. *Graph theory*. Springer, 3rd edition, 2005.

[9] D.-Z. Du and K.-I. Ko, editors. *Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book*. Kluwer, 1997.

[10] M. Fürer. Faster integer multiplication. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 57–66, New York, NY, USA, 2007. ACM.

[11] Y. Gurevich and S. Shelah. Expected computation time for hamiltonian path problem. *SIAM J. on Computing*, 16(3):486–502, 1987.

[12] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[13] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. of the ACM*, 18(6):341–343, 1975.

[14] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[15] E. Kaltofen and G. Villard. On the complexity of computing determinants. *Computational complexity*, 13:91–130, 2004.

[16] A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145(2):293–294, 1962. Translation in Soviet Physics-Doklady 7 (1963), pp. 595–596.

[17] J. Kleinberg and E. Tardos. *Algorithm design*. Addison-Wesley, 2005.

[18] D. E. Knuth. *The art of computer programming*, volume III, Sorting and searching. Addison-Wesley, 2nd edition, 1998.

[19] R. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 1975.

[20] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, pages 707–710, 1966.

[21] S. Lloyd. Computational capacity of the universe. *Physical Review Letters*, 88(23), 2002. http://focus.aps.org/story/v9/st27.

[22] M. Magazine, G.L.Nemhauser, and L.E.Trotter. When the greedy solution solves a class of knapsack problems. *Operations research*, 23(2):207–217, 1975.

[23] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expression with squaring requires exponential time. In *Proc. 12th IEEE Symposium on Switching and Automata Theory*, pages 125–129, 1972.

[24] A. Schönhage and V. Strasse. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[25] M. Sipser. The history and status of the P versus NP question. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 603–619, 1992.

[26] M. Sipser. *Introduction to the theory of computation*. Thomson, 2006.

[27] L. V. Toscani and P. A. S. Veloso. *Complexidade de Algoritmos*. Editora Sagra Luzzatto, 2a edition, 2005.

[28] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London MathṠoc.*, 2(42):230–265, 1936.

[29] P. M. B. Vitányi and L. Meertens. Big omega versus the wild functions. *SIGACT News*, 16(4), 1985.

[30] J. Vitter and P. Flajolet. *Handbook of theoretical computer science*, volume A: Algorithms and complexity, chapter Average-case analysis of algorithms and data structures. MIT Press, 1990.

[31] J. Wang. *Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book*, chapter Average-Case Intractable NP Problems. Kluwer, 1997.

Índice