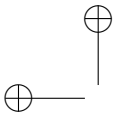
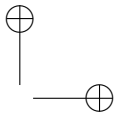
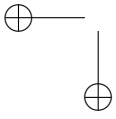
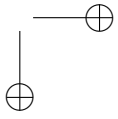
Dedicamos este livro a Jayme Luiz Szwarcfiter,
em seu 65º aniversário.

Prefácio

Este texto consiste das notas para um curso apresentado no 26º Colóquio Brasileiro de Matemática no IMPA, Rio de Janeiro, em julho de 2007.

O objetivo do curso "Introdução aos Algoritmos Randomizados" é apresentar a pesquisadores e estudantes da área de ciência da computação as técnicas fundamentais para o desenvolvimento de algoritmos randomizados (também chamados probabilísticos, por alguns autores). O curso tem caráter introdutório: não são assumidos conhecimentos avançados de probabilidade ou de algoritmos. Os conceitos teóricos que se fazem necessários são apresentados no próprio texto, em geral acompanhando os próprios problemas e algoritmos que os demandam. Ao completar este curso, o aluno terá travado contato com o instrumental básico dessa área e com um elenco representativo de algoritmos randomizados — e, em alguns casos, também determinísticos — para diversos problemas combinatórios. Este curso introdutório corresponde a tema de iniciação científica, tem um número mínimo de pré-requisitos, estimula o aluno à investigação científica e ainda não é oferecido regularmente nos currículos das universidades brasileiras.

O projeto para este curso a quatro autores nasceu da tese de doutorado de Vinícius, defendida no Programa de Engenharia de Sistemas e Computação (PESC) da COPPE/UFRJ em março de 2006. Durante a escrita dessa tese, realizada sob a orientação de Celina, foram criados vários algoritmos, entre determinísticos e randomizados, para um problema de teoria dos grafos. Alguns desses algoritmos foram desenvolvidos em co-autoria com Guilherme, que fez seu mestrado em estruturas de dados cinéticas (determinísticas e rando-

mizadas) também no PESC e orientado por Celina. Guilherme faz doutorado em geometria computacional na Universidade de Maryland. Manoel veio da Universidade Federal de Pernambuco participar como membro da banca da tese de doutorado de Vinícius, tendo naquela ocasião manifestado interesse em voltar ao tema que apresentara no Colóquio de 1989 para discutir o algoritmo randomizado de Rabin. Vinícius é, desde abril de 2006, pós-doutor junto ao PESC, onde lecionou uma versão preliminar destas notas.

Agradecemos ao comitê organizador do Colóquio Brasileiro de Matemática pela oportunidade de apresentar este curso. Agradecemos também à CAPES, ao CNPq e à FAPERJ pelo apoio concedido na forma de bolsas de doutorado, pós-doutorado, pesquisa e auxílios para viagens. Agradecemos a Antonio Carlos Rodrigues Monteiro e Raphael Carlos Santos Machado pelas atentas correções e sugestões de melhorias. Finalmente, agradecemos a Luiz Henrique de Figueiredo pelo cuidadoso trabalho de diagramação.

Celina, Guilherme, Manoel e Vinícius.
Rio de Janeiro, 30 de abril de 2007.

Conteúdo

Capítulo 1

Randomizados?

Algoritmo é uma seqüência de instruções para resolver um problema. Computadores são especialmente hábeis para lidar com algoritmos, pois, partindo de um estado inicial e seguindo à risca um encadeamento muito bem definido de passos, a resposta buscada será eventualmente anunciada.

Diz-se que experimentos são *aleatórios*, ou *randômicos*, quando seu resultado advém da interação de um número tão grande de variáveis que quaisquer tentativas de prevê-los com exatidão seriam simplesmente vãs. O lançamento de um dado ou de uma moeda é exemplo clássico: impossível conhecermos todos os fatores — posição e momentos linear e angular exatos no instante do lançamento, resistência do ar, grau de elasticidade das diversas colisões etc. — que influenciarão seu movimento até que, finalmente imóvel, apresente, naquela de suas faces que o "acaso" escolheu deixar voltada para cima, o resultado final do experimento.

Algoritmos *randomizados* são aqueles que utilizam experimentos randômicos para decidir, em um ou mais momentos durante sua execução, o que fazer ou para onde ir. Por motivo de clareza, algoritmos clássicos (não-randomizados) são também ditos *determinísticos*.

Na maioria dos casos, é conveniente imaginarmos um algoritmo randomizado como que lançando um dado ou uma moeda e, de acordo com o resultado obtido, decidindo entre a execução dessa ou daquela ação. Esta é a figura que estaremos constantemente evocando ao

longo do texto quando nos referirmos às escolhas aleatórias que nossos algoritmos precisarão tomar: o algoritmo simplesmente lança um dado — com qualquer número de faces.

É evidente, no entanto, que computadores não podem valer-se de tais expedientes físicos ou mesmo lúdicos. Portanto, na prática, o que entra em cena nos papéis de dados e moedas são os famosos *geradores de números aleatórios*[1].

Geradores de números aleatórios, por sua vez, nada mais são do que algoritmos — determinísticos! — que, utilizando *funções de dispersão* e valores obtidos do relógio interno da máquina, são capazes de simular o "sorteio" de um número, tirado de um conjunto finito deles, com distribuição de probabilidade tão próxima da uniforme[2] quanto melhor o gerador.

Recentemente, algoritmos randomizados têm encontrado aplicação em áreas tão distintas quanto computação algébrica, criptografia, geometria computacional, protocolos de redes, computação distribuída, teoria dos grafos, estruturas de dados e outras. O motivo: algoritmos randomizados são, quando comparados a seus correspondentes determinísticos, em geral mais simples ou mais eficientes — ou ambos.

Poderia soar bastante estranho que a introdução de aleatoriedade num reino digital já tão aparentemente imprevisível viesse — ao invés de piorar ainda mais as coisas — trazer ao mesmo tempo eficiência e simplicidade sem cobrar por isso qualquer preço. Mas há, de fato, um preço: a incerteza. Incerteza essa que pode aparecer em um de dois lugares: na própria qualidade da resposta obtida pelo algoritmo — que pode, em alguns casos, estar errada! — ou em seu tempo de execução, que passa a depender não mais exclusivamente da entrada do problema, mas também dos resultados dos experimentos aleatórios empregados pelo algoritmo.

Felizmente, como veremos, o preço cobrado pelos algoritmos randomizados cuja incerteza recai na qualidade das respostas por eles obtidas não é injustamente alto. De fato, não apenas é possível termos total conhecimento do quão (im)provável é a exibição de uma resposta incorreta por parte deles, como podemos também negociar

[1] Também chamados geradores de números *pseudo-aleatórios*, o que, embora possa soar um pouco pedante, é mais correto.

[2] Uma distribuição de probabilidade é *uniforme* quando todos os resultados do experimento aleatório ocorrem com idêntica probabilidade.

esse preço em troca da moeda tempo. Permitindo-lhes o emprego de uma maior quantidade de tempo computacional, consegue-se refinar a probabilidade de erro a níveis tão ínfimos quanto se queira. Veremos também que, quando essa diminuição da probabilidade de erro se dá exponencialmente com a quantidade de tempo empregada, como acontece na maioria dos casos interessantes, podemos estar diante de um algoritmo randomizado eficiente e útil, situação essa certamente bastante desejável.

Já a incerteza presente naqueles outros algoritmos — cujo tempo de execução não pode ser deterministicamente previsto — volta-se quase sempre a seu próprio favor. A idéia central é a de que, *por pior que seja a entrada do problema*, seu tempo de execução será, com alta probabilidade, bom. Um paradigma muito útil aqui é o do *adversário malicioso*, uma entidade supostamente conhecedora de cada linha de nosso algoritmo e que, implacável, submete-lhe sempre uma instância de entrada tão desfavorável quanto possível, isto é, aquela que dele exigirá a execução do número máximo de passos. Tal entidade — cujas ações assumem, na prática, formas tão prosaicas quanto seqüências pré-ordenadas de números ou grafos conexos 2-regulares — nada pode contra algoritmos randomizados, uma vez que desconhece as escolhas aleatórias que serão por eles feitas. Ou seja, a mesma incerteza que, por um lado, não nos permite prever exatamente o tempo de execução para uma determinada instância do problema (por exemplo, para a *pior* possível) é, por outro, o que nos garante que uma entrada *jamais será ruim o suficiente* para obrigar nosso algoritmo a executar um número excessivo de passos — passos que seriam, talvez, forçosamente executados por um algoritmo determinístico ao se ver diante dessa ou daquela instância desfavorável.

Este primeiro capítulo é a introdução, propriamente dita, aos algoritmos randomizados. Fazemos, nas seções 1.1 e 1.2, uma breve revisão de tópicos básicos de probabilidade, variáveis aleatórias e desigualdades de cauda, necessários ao bom entendimento das análises dos algoritmos que aparecem ao longo do texto. Na seção 1.3, descrevemos as duas principais famílias de algoritmos randomizados — Monte Carlo e Las Vegas — e apresentamos a argumentação combinatória que as justifica, ilustrando-as com exemplos fáceis e didáticos. Na seção 1.4 encontram-se as classes de complexidade às quais pertencem os problemas que podem ser resolvidos por algoritmos ran-

domizados. O capítulo se encerra com seções de exercícios e notas bibliográficas, como acontecerá também nos demais capítulos deste texto.

O Capítulo 2 aborda a análise probabilística de algoritmos, bem como alguns dos paradigmas combinatórios mais comuns relacionados a nosso tema. Algoritmos de ordenação bem conhecidos são utilizados para ilustrar as idéias principais.

Primalidade é o tema de nosso Capítulo 3. Por exigir este tema ferramentas matemáticas próprias e decerto mais complexas que aquelas necessárias aos demais capítulos, adotamos aqui uma abordagem mais lenta, no sentido em que nos concedemos voltar a definições básicas de onde resultados mais complexos são então, pouco a pouco, inferidos. Permitimos, assim, que o leitor acumule, ao longo de quase todo o capítulo, o conhecimento que se faz essencial para o entendimento de um dos algoritmos randomizados mais importantes, famosos e utilizados na prática: o algoritmo de Rabin para primalidade.

No Capítulo 4, discutimos alguns problemas de geometria computacional, área que vem se mostrando fértil para o florescimento de algoritmos randomizados interessantes e eficientes — como os que são nesse capítulo apresentados.

O Capítulo 5 fecha este texto apresentando o método probabilístico para provas de existência e de-randomização.

1.1 Probabilidade básica

É evidente que o estudo de algoritmos randomizados não poderia prescindir de alguma dose de cálculo de probabilidades, pelo que então fazemos agora uma breve e informal revisão de alguns de seus mais importantes conceitos. Procuramos ilustrar cada um dos tópicos com exemplos fáceis, e recomendamos as notas bibliográficas ao final deste capítulo para material mais aprofundado.

1.1.1 Axiomas e definições

Espaço probabilístico, espaço amostral e eventos

Sempre que se fala em probabilidade, é preciso deixar claro o *espaço probabilístico* sobre o qual as probabilidades são aferidas. Um espaço

probabilístico é constituído de:

- um *espaço amostral* $\Omega = \{r_1, r_2, \ldots\}$, que é o conjunto de todos os possíveis resultados de um experimento aleatório qualquer. *Eventos* são quaisquer subconjuntos de Ω. Os subconjuntos unitários $E_i = \{r_i\} \subseteq \Omega, i = 1, 2, \ldots$, definem os *eventos elementares* daquele experimento.

- uma *função de probabilidade* $\mathbf{Pr}$, que associa a cada evento $A \subseteq \Omega$ uma probabilidade $\mathbf{Pr}[A]$, que pode ser entendida como o valor para o qual converge a taxa de ocorrência daquele evento[3] caso o experimento seja repetido por um número muito grande de vezes.

Função de probabilidade

A função de probabilidade $\mathbf{Pr}$ precisa atender às seguintes condições:

1. Para todo evento $A \subseteq \Omega$, $0 \leq \mathbf{Pr}[A] \leq 1$.

2. $\mathbf{Pr}[\Omega] = 1$.

3. Para eventos $A_1, A_2, \ldots$ disjuntos dois-a-dois, vale $\mathbf{Pr}[\bigcup_i A_i] = \sum_i \mathbf{Pr}[A_i]$.

Seja, por exemplo, o experimento aleatório muito simples do lançamento de um dado honesto de seis faces e o espaço probabilístico definido pelo conjunto $\Omega = \{1, 2, 3, 4, 5, 6\}$ de seus possíveis resultados e pela função de probabilidade $\mathbf{Pr}$ que associa a cada evento elementar $E_i = \{i\}$ — entendido como "o resultado obtido foi i" $(i = 1, \ldots, 6)$ — a probabilidade $\mathbf{Pr}[E_i] = 1/6$. Podemos estar interessados em eventos um pouco mais complexos como "o resultado obtido foi par", que equivale ao evento $B = \{2, 4, 6\}$, ou "o resultado obtido foi maior do que 7", que é simplesmente o evento $C = \{\ \}$.

[3] Quando se pensa na *ocorrência* de um evento A, está-se pensando no(s) caso(s) em que o resultado do experimento pertence ao conjunto A.

Princípio da inclusão-exclusão

Sejam $A_1, A_2, \ldots$ eventos arbitrários quaisquer. Então,

$$\begin{aligned}\mathbf{Pr}\left[\bigcup_i A_i\right] &= \sum_i \mathbf{Pr}[A_i] \\ &\quad - \sum_{i<j} \mathbf{Pr}[A_i \cap A_j] \\ &\quad + \sum_{i<j<k} \mathbf{Pr}[A_i \cap A_j \cap A_k] \\ &\quad - \cdots + (-1)^{l+1} \sum_{i_1<i_2<\cdots<i_l} \mathbf{Pr}\left[\bigcap_{r=1}^{l} A_{i_r}\right] \\ &\quad + \cdots\end{aligned}$$

Ainda no exemplo do lançamento de um dado, suponha que estejamos interessados no evento "o resultado é par *ou* múltiplo de 3". É claro que, aqui, estamos diante do evento $D = \{2, 3, 4, 6\}$ e, pela condição 3 da definição da função de probabilidade, temos que $\mathbf{Pr}[D] = \mathbf{Pr}[\{2\}] + \mathbf{Pr}[\{3\}] + \mathbf{Pr}[\{4\}] + \mathbf{Pr}[\{6\}] = 4/6$. No entanto, poderíamos pensar em D como sendo a união dos eventos "resultado par" e "resultado múltiplo de 3", isto é, $D_1 = \{2, 4, 6\}$ e $D_2 = \{3, 6\}$, respectivamente. Sendo assim, e aplicando o princípio da inclusão-exclusão que queremos ilustrar, teríamos

$$\begin{aligned}\mathbf{Pr}[D] &= \mathbf{Pr}[D_1 \cup D_2] \\ &= \mathbf{Pr}[D_1] + \mathbf{Pr}[D_2] - \mathbf{Pr}[D_1 \cap D_2] \\ &= \mathbf{Pr}[\{2, 4, 6\}] + \mathbf{Pr}[\{3, 6\}] - \mathbf{Pr}[\{6\}] \\ &= 3/6 + 2/6 - 1/6 \\ &= 4/6.\end{aligned}$$

Limite da união

Decorre do princípio da inclusão-exclusão a seguinte desigualdade, tão simples quanto útil na obtenção de limites para diversas proba-

bilidades associadas a algoritmos randomizados:

$$\mathbf{Pr}\left[\bigcup_i A_i\right] \leq \sum_i \mathbf{Pr}[A_i].$$

Embora pareça pouco justo este *limite da união*, o fato é que não apenas é, na maioria das vezes, perfeitamente suficiente a utilização de tal limite quanto seria extremamente difícil o cálculo exato de probabilidades complexas pelo princípio da inclusão-exclusão. Além disso, como veremos, em muitos casos já estamos mesmo trabalhando com desigualdades e majorantes, donde um excesso de pragmatismo no cálculo exato de determinadas probabilidades não faria mais que adicionar páginas pouco úteis à análise do algoritmo em questão.

Probabilidades condicionais e eventos independentes

A *probabilidade condicional* de um evento A dado um evento B é escrita $\mathbf{Pr}[A|B]$ e corresponde à probabilidade de que o resultado do experimento aleatório pertença ao conjunto A sabendo-se que pertence ao conjunto B.

Podemos calcular $\mathbf{Pr}[A|B]$ como

$$\mathbf{Pr}[A|B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}.$$

Se $\mathbf{Pr}[A|B] = \mathbf{Pr}[A]$, dizemos que A e B são *independentes* entre si. Intuitivamente, conhecermos que o resultado do experimento pertence a B em nada altera a avaliação da probabilidade de que ele pertença a A.

Da definição das probabilidades condicionais advém a expressão para o cálculo da probabilidade de uma conjunção de eventos:

$$\mathbf{Pr}\left[\bigcap_{i=1}^{n} A_i\right] = \prod_{i=1}^{n} \mathbf{Pr}\left[A_i | \bigcap_{j<i} A_j\right].$$

E, no caso particular de eventos dois-a-dois independentes, temos

$$\mathbf{Pr}\big[\bigcap_{i=1}^{n} A_i\big] = \prod_{i=1}^{n} \mathbf{Pr}\big[A_i\big].$$

Probabilidade total e a Regra de Bayes

Se particionamos o espaço amostral Ω em eventos (disjuntos) $B_1, B_2, \ldots, B_n$, podemos calcular a probabilidade de um evento A qualquer pela expressão seguinte, conhecida como *probabilidade total*:

$$\mathbf{Pr}[A] = \sum_{i=1}^{n} \mathbf{Pr}[A|B_i]\mathbf{Pr}[B_i].$$

Dessa expressão decorre a chamada *Regra de Bayes* para probabilidades condicionais:

$$\mathbf{Pr}[B_k|A] = \frac{\mathbf{Pr}[B_k \cap A]}{\mathbf{Pr}[A]} = \frac{\mathbf{Pr}[A|B_k]\mathbf{Pr}[B_k]}{\sum_{i=1}^{n} \mathbf{Pr}[A|B_i]\mathbf{Pr}[B_i]}.$$

1.2 Variáveis aleatórias e esperança

Muitas vezes interessa-nos atribuir um valor numérico ao resultado de um experimento aleatório qualquer. Por exemplo, se nosso experimento consiste de seis lançamentos consecutivos de uma moeda, temos $2^6 = 64$ diferentes seqüências possíveis de caras e coroas, cada uma das quais constituindo um evento elementar do espaço amostral Ω associado àquele experimento. Se usarmos a notação K para cara e C para coroa, nossos eventos elementares podem ser escritos como "KKKKKK", "KKKKKC", "KKKKCK", "KKKKCC" etc. Pode ser, no entanto, que interesse-nos apenas, digamos, o tamanho da maior seqüência de caras consecutivas. Neste caso, eventos elementares como "CKKKKC" e "KKKKCC" correspondem a um mesmo resultado de um máximo de 4 caras consecutivas.

Uma variável aleatória X é, portanto, uma função de certo espaço amostral Ω no conjunto dos números reais (isto é, $X : \Omega \to \mathbb{R}$), de forma que, ao escrevermos $\mathbf{Pr}[X = x]$ estamos nos referindo à probabilidade de que o resultado r do experimento aleatório seja tal que $X(r) = x$.

Seja novamente o experimento do exemplo anterior, onde uma moeda é lançada seis vezes consecutivas. Se definimos uma variável aleatória Y como sendo o número total de coroas obtidas, escrevermos $\mathbf{Pr}[Y \leq 1]$ é o mesmo que escrevermos $\mathbf{Pr}[\{$"KKKKKK",

"CKKKKK", "KCKKKK", "KKCKKK", "KKKCKK", "KKKKCK", "KKKKKC"}], donde $\mathbf{Pr}[Y \leq 1] = 7/64$.

Analogamente à independência de eventos, dizemos que duas variáveis aleatórias X e Y são independentes se $\mathbf{Pr}[X = x, Y = y] = \mathbf{Pr}[X = x]\mathbf{Pr}[Y = y]$ ou, equivalentemente, $\mathbf{Pr}[X = x|Y = y] = \mathbf{Pr}[X = x]$.

A função *densidade de probabilidade* $p : \mathbb{R} \to [0, 1]$ de uma variável aleatória X é definida como $p_X(x) = \mathbf{Pr}[X = x]$.

A *esperança*, ou o *valor esperado*, de uma variável aleatória X é, intuitivamente, a média dos possíveis valores de X ponderados pelas freqüências com que X assume cada um daqueles valores. Mais formalmente,

$$\mathbf{E}[X] = \sum_x x p_X(x).$$

1.2.1 Linearidade da esperança

Um conceito absolutamente importante com respeito às variáveis aleatórias e sua aplicação em algoritmos randomizados é o da *linearidade da esperança*, que reza que, não importando se as variáveis aleatórias em questão são ou não independentes, vale

$$\mathbf{E}[h(X_1, X_2, \ldots, X_n)] = h\left(\mathbf{E}[X_1], \mathbf{E}[X_2], \ldots, \mathbf{E}[X_n]\right),$$

para toda função linear h.

Voltemos, ainda uma vez, ao exemplo dos seis lançamentos da moeda. Estamos interessados no valor esperado da variável aleatória Y que representa o total de coroas obtidas. Pela definição de esperança, poderíamos calcular $\mathbf{E}[Y]$ como

$$\mathbf{E}[Y] = 1 \cdot \mathbf{Pr}[Y = 1] + 2 \cdot \mathbf{Pr}[Y = 2] + \cdots + 6 \cdot \mathbf{Pr}[Y = 6],$$

onde $\mathbf{Pr}[Y = y]$ é dada por $\binom{6}{y}2^{-6}$. Pela linearidade da esperança, no entanto, é fácil obter $\mathbf{E}[Y]$ escrevendo primeiramente $Y = Y_1 + Y_2 + \cdots + Y_6$, onde cada Y_i representa o número de coroas obtidas no i-ésimo lançamento da moeda, e, então:

$$\mathbf{E}[Y] = \mathbf{E}\left[\sum_{i=1}^{6} Y_i\right] = \sum_{i=1}^{6} \mathbf{E}\left[Y_i\right] = 6 \cdot 1/2 = 3.$$

Aqui tivemos que calcular explicitamente apenas a esperança de Y_i, que é $0 \cdot \mathbf{Pr}[Y_i = 0] + 1 \cdot \mathbf{Pr}[Y_i = 1] = 1/2$, supondo "honesta" a moeda.

1.2.2 Limites de cauda

A esperança de uma variável aleatória nos dá a idéia de "média" e costuma ser extremamente valioso conhecê-la, especialmente se estamos interessados no valor acumulado após um número grande de repetições do experimento randômico. Por exemplo, se a variável aleatória em questão é o tempo de execução X de um algoritmo randomizado, é no mínimo útil — e talvez indispensável — sabermos que, após um número grande k de execuções daquele algoritmo, o tempo total gasto terá convergido para $k\mathbf{E}[X]$.

Entretanto, no caso de estarmos interessados em *uma* atribuição de valor àquela variável aleatória (por exemplo, uma execução particular do algoritmo), a esperança, apenas, não nos diz muito a respeito da densidade de probabilidade daquela variável (que é, em última instância, o que nos diria *tudo* sobre ela).

Na ausência da expressão exata para a densidade de probabilidade, podemos fechar algumas lacunas utilizando *limites de cauda* como a *desigualdade de Markov*, dada a seguir:

$$\mathbf{Pr}[X \geq a] \leq \frac{\mathbf{E}[X]}{a}, \text{ para todo } a > 0,$$

válida para variáveis aleatórias X que assumem apenas valores não-negativos.

Na verdade, a desigualdade de Markov é o melhor que podemos conseguir quando tudo o que conhecemos é a esperança da variável aleatória (e o fato de assumir ela apenas valores não-negativos).

Se, além da esperança, conhecermos também a *variância* de uma variável aleatória X,

$$\mathbf{Var}[X] = \mathbf{E}[(X - \mathbf{E}[X])^2] = \mathbf{E}[X^2] - (\mathbf{E}[X])^2,$$

podemos estabelecer, a partir da desigualdade de Markov, um limite mais forte conhecido como *desigualdade de Chebyshev*:

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq a] \leq \frac{\mathbf{Var}[X]}{a^2}, \text{ para todo } a > 0.$$

A desigualdade de Chebyshev também pode ser escrita como

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq k\sigma] \leq \frac{1}{k^2}, \text{ para todo } k > 1,$$

onde $\sigma = \sqrt{\mathbf{Var}[X]}$ é o *desvio padrão* de X, que dá intuitivamente o quão afastados da esperança os valores assumidos por X devem estar.

Desigualdades como as de Markov e Chebyshev são muito utilizadas na análise de algoritmos probabilísticos e são muitas vezes as responsáveis por prover o grau de confiança necessário à adoção de estratégias randomizadas em problemas práticos.

As desigualdades conhecidas como *limites de Chernoff* podem ser ainda mais poderosas, ainda que fujam ao escopo deste curso introdutório. O leitor pode encontrar maiores informações nas notas bibliográficas.

1.2.3 Algumas variáveis aleatórias importantes

Descreveremos agora, para referência, as variáveis aleatórias mais comuns e úteis para nosso tema.

Variável aleatória de Bernoulli

A variável aleatória *de Bernoulli* é comumente usada como *indicador* da ocorrência de algum evento, já que pode assumir apenas dois valores: 0 (normalmente associado à não-ocorrência de determinado evento) ou 1 (normalmente associado à sua ocorrência).

É comum chamarmos de "sucesso" ao evento para o qual o indicador de Bernoulli recebe valor 1, e de "fracasso" a seu complemento. Sendo p a probabilidade associada ao sucesso, temos a seguinte densidade de probabilidade para nossa variável de Bernoulli:

$$p_X(x) = \begin{cases} 1-p & \text{se } x = 0 \\ p & \text{se } x = 1 \\ 0 & \text{para todos os demais valores de } x. \end{cases}$$

O evento em questão, propriamente dito, pode ser qualquer. Por exemplo, suponha uma roleta numerada de 0 a 36 onde a probabilidade de se obter qualquer um dos números é idêntica e, portanto,

igual a 1/37. Estamos interessados em saber se, numa dada rodada da roleta, houve ou não, como resultado, o número zero (que, no famoso jogo de azar, é associado a ganho da "banca"). Teríamos, nesse caso, "sucesso" definido como o evento em que zero é o número sorteado, sendo fracasso o evento complementar em que sai qualquer outro número entre 1 e 36. A *probabilidade de sucesso* associada a nossa variável de Bernoulli seria, portanto, de 1/37.

É facilmente demonstrável que a esperança de uma variável aleatória de Bernoulli é igual à probabilidade de sucesso p. Sua variância é $p(1-p)$.

Variável aleatória binomial

A variável aleatória *binomial* aparece, normalmente, quando se deseja indicar o total de sucessos obtidos após uma seqüência de n experimentos randômicos idênticos e independentes. Em outras palavras, pode ser entendida como a soma de n indicadores de Bernoulli, cada um dos quais associado a uma mesma probabilidade de sucesso p.

Já vimos — sem o termos anunciado — a variável aleatória binomial, quando nos interessamos pelo número total de coroas obtidas em seis lançamentos consecutivos de uma moeda. Trata-se, portanto, de uma binomial X que pode ser entendida como a soma $X = X_1 + X_2 + \cdots + X_6$ de indicadores de Bernoulli, cada um dos quais associado a uma probabilidade de sucesso igual a 1/2.

Usualmente, abreviam-se variáveis aleatórias binomiais com parâmetros n (número de repetições de um experimento) e p (probabilidade de que uma execução do experimento resulte em sucesso) como $B(n, p)$. Sua densidade de probabilidade é dada por

$$p_X(x) = \binom{n}{x} p^x (1-p)^{n-x},$$

para $0 \leq x \leq n$ e x inteiro. Todos os demais valores reais de x resultam em $p_X(x) = 0$.

A esperança $\mathbf{E}[X]$ de uma variável aleatória X com distribuição binomial $B(n, p)$ é igual a np, como pode ser facilmente verificado. Para obtermos a variância da binomial, é preciso conhecermos $\mathbf{E}[X^2] = n(n-1)p^2 + np$, que resulta diretamente da definição de

esperança, mas cujos cálculos não apresentamos aqui. Chega-se, assim, a $\mathbf{Var}[X] = np(1-p)$.

Variável aleatória geométrica

Quando, ao invés de nos interessarmos pelo total de sucessos numa seqüência, estamos preocupados com o número de repetições de um experimento randômico até o momento em que o primeiro sucesso tenha sido observado, estamos diante de uma variável aleatória geométrica.

Suponha que, no exemplo da moeda, não limitássemos em 6 o número de lançamentos, mas, do contrário lançássemos a moeda o número de vezes que fosse necessário até o aparecimento da primeira, digamos, coroa. Este número é exatamente o valor assumido por nossa variável aleatória geométrica.

A densidade de uma geométrica X com probabilidade de sucesso p é dada por

$$p_X(x) = \begin{cases} p(1-p)^{x-1} & \text{para } x = 1, 2, 3, \ldots \\ 0 & \text{para todos os demais valores de } x. \end{cases}$$

A esperança de uma geométrica é o inverso de sua probabilidade de sucesso, ou $\mathbf{E}[X] = 1/p$. Sua variância é $(1-p)/p^2$.

1.3 Monte Carlo e Las Vegas

Voltemos, agora, ao assunto que mais nos interessa: algoritmos randomizados. Como o adiantáramos na introdução, portanto, existem dois grandes grupos de algoritmos randomizados, que se distingüem um do outro pela localização da incerteza que resulta da presença de experimentos aleatórios a dirigir-lhes de algum modo os passos: se na própria corretude da resposta apresentada, são ditos algoritmos *de Monte Carlo*; se em seu tempo de execução, são chamados algoritmos *de Las Vegas*.

Da aplicação real depende a maior adequação de um ou outro tipo. Se é preciso a garantia de que o tempo exigido pelo algoritmo será deterministicamente limitado, em todas as suas execuções, por uma certa função do tamanho da entrada, um algoritmo de Monte

Carlo é certamente o mais indicado, e uma margem diminuta de erro pode mesmo ser irrelevante diante da certeza de boa performance que ele venha a oferecer. Já a necessidade de se estar *sempre* diante da resposta correta aponta o uso de um algoritmo de Las Vegas.

Na maior parte das vezes, no entanto, o que permite ou sugere a construção de algoritmos de um ou outro tipo é o conjunto de características do problema em si — e alguma dose de experiência. Tanto é assim que nem sempre se conhece algoritmo de Las Vegas eficiente para um problema para o qual existe algoritmo de Monte Carlo arrasador; da mesma forma, muito embora seja sempre possível construirmos a partir de um algoritmo de Las Vegas um outro, de Monte Carlo (vide seção 1.3.3), nem sempre este último apresentará desempenho suficientemente interessante.

1.3.1 Monte Carlo

Algoritmos randomizados de Monte Carlo nem sempre dão a resposta certa. Existe uma chance, inerente ao algoritmo, de que a resposta apresentada esteja desastrada e inescrupulosamente errada.

Não é preciso, no entanto, que lhes votemos absolutamente qualquer sentimento prematuro de desconfiança ou antipatia. Afinal, a vida prática é também cheia de incertezas. Exames laboratoriais podem diagnosticar doenças inexistentes, mas não deixam de ser ferramentas valiosas nas mãos do bom médico. É preciso apenas saber lidar com a possibilidade de erro.

Algoritmos randomizados não estão limitados a problemas de decisão[4]. Estes, porém, permitem dividir os algoritmos de Monte Carlo em dois tipos: os de *erro unilateral* e os de *erro bilateral*.

Algoritmos de Monte Carlo de erro unilateral para problemas de decisão erram, como sugere o nome, apenas para um dos lados: aqueles que são *baseados-no-sim* nunca erram quando a resposta por eles encontradas é `SIM`; já os *baseados-no-não* estão sempre falando a verdade quando apresentam o `NÃO` como resposta. Algoritmos de Monte

[4]Problemas *de decisão* são aqueles em que se deseja descobrir se algo é verdadeiro: existe representação plana para este grafo? Tal número é primo? É possível ir da cidade A à cidade B passando por no máximo k cidades intermediárias? Esta amostra de sangue possui o vírus XYZ? Estes são todos exemplos de problemas de decisão — a resposta certa é um simples `SIM` ou `NÃO`.

Carlo de erro bilateral podem retornar tanto SIM quanto NÃO incorretos.

A unilateralidade do erro de um algoritmo de Monte Carlo é característica importante que permite-nos refinar eficientemente nosso "grau de confiança" na resposta obtida a níveis tão bons quanto desejemos.

Exemplo: identidade de polinômios

Seja o seguinte problema: dados dois polinômios de grau d, um deles apresentado na forma de um produto de polinômios de primeiro grau, e.g., $F(x) = (x - a_1)(x - a_2) \cdots (x - a_d)$, e outro na forma canônica da soma de monômios, e.g., $G(x) = b_d x^d + b_{d-1} x^{d-1} + \cdots + b_1 x + b_0$, é verdade que ambos os polinômios são idênticos?

Um algoritmo muito simples é aquele baseado na comparação dos coeficientes dos termos de mesmo grau dos dois polinômios, uma vez que ambos encontrem-se na forma canônica. Para isso, o algoritmo teria que, em primeiro lugar e inevitavelmente, transformar $F(x)$, executando, para isso, um número quadrático $O(d^2)$ de operações básicas de adição e multiplicação.

Eis um algoritmo randomizado que dá a resposta certa com probabilidade alta em tempo linear no grau dos polinômios, ou seja, executando um número $O(d)$ de operações básicas:

Entrada:
    F, G: dois polinômios de grau d.
Saída:
    SIM ou NÃO, decidindo se F e G são idênticos.

identidadePolinômios(F, G):
    sorteie aleatória e uniformemente um inteiro w entre 1 e $100d$
    avalie $F(w)$ e $G(w)$
    retorne NÃO se $F(w) \neq G(w)$; caso contrário, retorne SIM

Figura 1.1: Monte Carlo para verificar a identidade de polinômios.

Observe que o algoritmo da figura 1.1 é um algoritmo de Monte Carlo de erro unilateral baseado-no-não. De fato, quando o algoritmo responde `NÃO`, ele tem sempre razão. Há um *certificado* para esse `NÃO`, algo que garante a corretude dessa resposta. Ora, se existe um número w para o qual $F(w) \neq G(w)$, os polinômios não podem ser idênticos. Observe que, se os polinômios são idênticos o algoritmo jamais responderá erradamente que eles não o sejam. Não existe algo como um "certificado falso" para o `NÃO`.

Por outro lado, se a resposta dada pelo algoritmo é `SIM`, há uma chance de que ela esteja errada. É possível que os polinômios *não* sejam idênticos, mas que o inteiro w sorteado aleatoriamente apenas seja raiz do polinômio $H(x) = F(x) - G(x)$, caso este em que $F(w)$ e $G(w)$ avaliariam o mesmo valor *especificamente* para w. Não constituiria a igualdade $F(w) = G(w)$, portanto, um certificado para o `SIM`[5]; poder-se-ia concluir apenas que o algoritmo *não localizou* um certificado para o `NÃO`.

Se o algoritmo sempre acerta a resposta quando os polinômios são idênticos, a pergunta é: qual a probabilidade do algoritmo errar a resposta quando os polinômios *não são* idênticos?

Sabemos que um polinômio de grau d possui no máximo d raízes inteiras distintas. Dessa forma, a probabilidade de que o inteiro w sorteado aleatoriamente seja raiz de $F(x) - G(x)$ é menor ou igual a $d/100d = 1/100$.

O que ganhamos com esse algoritmo? Ora, é possível avaliar aritmeticamente polinômios de grau d em tempo linear $O(d)$. Rodando, portanto, em tempo $O(d)$, nosso algoritmo é extremamente mais eficiente do que o algoritmo determinístico $O(d^2)$ que mencionáramos.

Probabilidades associadas ao Monte Carlo de erro unilateral

A pergunta que precisamos responder, quando diante de um algoritmo de Monte Carlo, é: *qual a probabilidade* $p \geq 1 - \varepsilon$ *de que a resposta correta seja dada*?

[5] É evidente que poderia haver um certificado para o `SIM`, se o desejássemos. Bastaria nos certificarmos de que os polinômios avaliam os mesmos resultados para $d + 1$ valores distintos de w. Mas ter que realizar $O(d)$ avaliações, cada uma das quais em tempo $O(d)$, nos daria um algoritmo de tempo quadrático $O(d^2)$, justamente a performance ruim que queremos evitar. Não há, portanto, um certificado *eficiente* para o `SIM`.

Seja um algoritmo de Monte Carlo de erro unilateral e sejam os seguintes eventos associados a uma execução do algoritmo para uma determinada instância de um problema de decisão.

- A_S — o algoritmo responde `SIM`;
- A_N — o algoritmo responde `NÃO`;
- C_S — a resposta correta para aquela entrada é `SIM`;
- C_N — a resposta correta para aquela entrada é `NÃO`.

Para um melhor acompanhamento dos parágrafos seguintes, aconselhamos a consulta à figura 1.2, onde uma seta de X para Y representa a probabilidade condicional $\mathbf{Pr}[Y|X]$.

Há duas maneiras de se entender as probabilidades associadas a um algorimo de Monte Carlo de erro unilateral. A primeira é pensarmos nas probabilidades de acerto $\mathbf{Pr}[C_S|A_S]$ e $\mathbf{Pr}[C_N|A_N]$ ou de erro $\mathbf{Pr}[C_N|A_S]$ e $\mathbf{Pr}[C_S|A_N]$ condicionadas *à resposta apresentada*. Quando se constrói um algoritmo, no entanto, em geral estamos preocupados com as probabilidades de acerto $\mathbf{Pr}[A_S|C_S]$ e $\mathbf{Pr}[A_N|C_N]$ ou de erro $\mathbf{Pr}[A_N|C_S]$ e $\mathbf{Pr}[A_S|C_N]$ condicionadas *à entrada do problema*[6].

Aparentemente, é mais intuitivo pensarmos nas condicionais associadas à resposta do algoritmo. No entanto, o fato é que, embora o "certificado" apresentado para o `NÃO` (respectivamente, para o `SIM`) por um algoritmo de Monte Carlo de erro unilateral baseado-no-não (resp. baseado-no-sim) nos dê automaticamente $\mathbf{Pr}[C_N|A_N] = 1$ (resp. $\mathbf{Pr}[C_S|A_S] = 1$), nem sempre é fácil — ou possível — calcularmos as probabilidades condicionadas ao fato de que a resposta dada *não veio* acompanhada de um certificado (pontos de interrogação nos diagramas da figura 1.2). Em outras palavras, se um algoritmo baseado-no-não respondeu `SIM` ou se um baseado-no-sim respondeu `NÃO`, só é possível obter as probabilidades condicionais se conhecermos a distribuição de probabilidade da entrada do problema (vide exercício 2). Ou poderemos, no máximo, atualizar nosso "modelo de confiança" (vide exercício 3).

[6] Admitimos que pode parecer confuso trabalhar com as condicionais nos dois sentidos. Nossa recomendação é a de que o leitor prefira concentrar-se nas probabilidades condicionadas à entrada do problema.

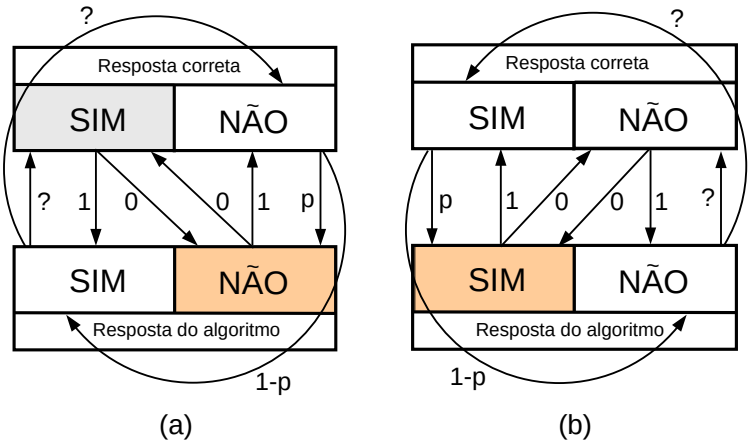


Figura 1.2: (a) MC baseado-no-não. (b) MC baseado-no-sim.

Se, por outro lado, concentrarmo-nos nas probabilidades associadas à entrada do problema, é fácil respondermos satisfatoriamente aquela pergunta fundamental. Algoritmos baseados-no-não responderão corretamente `SIM` *sempre que a entrada for uma instância* `SIM` (já que não inventarão jamais um certificado falso para o `NÃO`). Quando a entrada for uma instância `NÃO`, a corretude da resposta depende do algoritmo ter a capacidade (ou "sorte") de encontrar um certificado para o `NÃO`[7].

> *A probabilidade de acerto de um algoritmo de Monte Carlo de erro unilateral baseado-no-não é maior ou igual à probabilidade de que o algoritmo encontre um certificado para o* `NÃO` *caso a entrada seja uma instância* `NÃO`.

[7] Analogamente, algoritmos baseados-no-sim responderão corretamente `NÃO` *sempre que a entrada for uma instância* `NÃO`. Quando a entrada for `SIM`, a resposta só será um correto `SIM` se o algoritmo tiver a "sorte" de encontrar um certificado (o que, deseja-se, acontece com alta probabilidade).

A probabilidade de acerto de um algoritmo de Monte Carlo de erro unilateral baseado-no-sim é maior ou igual à probabilidade de que o algoritmo encontre um certificado para o `SIM` *caso a entrada seja uma instância* `SIM`*.*

Portanto, é com a (alta) probabilidade de encontrar um certificado para um dos lados que devemos nos preocupar quando do desenvolvimento e análise de algoritmos de Monte Carlo de erro unilateral.

Reduzindo a probabilidade de erro

Seja A um algoritmo de Monte Carlo de erro-unilateral baseado-no-não que erra com probabilidade menor ou igual a ε_1 e seja I uma instância qualquer para um problema de decisão.

Sigamos agora, o seguinte plano: executemos A, seguida e independentemente, diversas vezes, até que uma resposta `NÃO` tenha sido encontrada — e, portanto, certificada — ou até que um número máximo de t execuções tenha sido executado.

Com que probabilidade, após a adoção da estratégia acima, estaremos diante de uma resposta incorreta para o problema?

Ora, se a resposta correta é `SIM`, forçosamente teremos nas mãos um `SIM` após exatas t execuções. Então, para que o algoritmo erre, é preciso que a resposta correta seja `NÃO` e que ele falhe em encontrar um certificado para o `NÃO` por t vezes independentes e consecutivas. Sendo assim, e designando a notação A_S^k para o evento em que a k-ésima execução do algoritmo retorna `SIM`, a probabilidade global de erro ε_t pode ser dada por:

$$\begin{aligned}
\varepsilon_t &= \mathbf{Pr}\left[C_N, \bigcap_{k=1}^{t} A_S^k\right] \\
&= \mathbf{Pr}[C_N] \cdot \prod_{k=1}^{t} \mathbf{Pr}\left[A_S^k | C_N, \bigcap_{j=1}^{k-1} A_S^j\right] \\
&= \mathbf{Pr}[C_N] \cdot \prod_{k=1}^{t} \mathbf{Pr}[A_S^k | C_N] \\
&= \mathbf{Pr}[C_N] \cdot \left(\mathbf{Pr}[A_S^1 | C_N]\right)^t .
\end{aligned}$$

Na terceira linha, usamos o fato de que as repetições do algoritmo são todas independentes, de forma que o conhecimento dos resultados obtidos pelas $k-1$ execuções do algoritmo em nada altera as probabilidades associadas à k-ésima execução.

Como $\mathbf{Pr}[C_N] \leq 1$, temos

$$\varepsilon_t \leq \left(\mathbf{Pr}[A_S^1 | C_N]\right)^t = \varepsilon_1^t.$$

Vemos, assim, que a probabilidade de erro decresce exponencialmente com o aumento do número de repetições independentes do algoritmo[8].

Note que calculamos a probabilidade de erro como sendo a probabilidade da conjunção dos eventos C_N e $A_S^k, k = 1, \ldots, t$. Isto é possível pois o algoritmo possui erro unilateral (baseado-no-não, nesse caso), de forma que bastaria uma única resposta assertiva (com exibição de certificado para o `NÃO`, no caso) para que tivéssemos certeza da resposta correta. São necessárias, portanto, para que haja exibição de resposta *incorreta*, t execuções distintas e independentes do algoritmo, cada uma das quais falhando em encontrar um certificado (para o `NÃO`).

A probabilidade de acerto é, evidentemente, a complementar da probabilidade de erro, e, portanto, maior ou igual a $1 - \varepsilon_t$. Se, por outro lado, optássemos por calcular diretamente a probabilidade de acerto como sendo a probabilidade da disjunção dos eventos A_N^k, precisaríamos ou trabalhar com um limite pouco justo usando o *limite da união* (vide seção 1.1.1) ou obter a probabilidade da disjunção de eventos usando, a duras penas, o *princípio da inclusão-exclusão* (vide, igualmente, a seção 1.1.1).

[8] Em alguns casos, como no do algoritmo de Monte Carlo para a verificação da identidade de polinômios, a probabilidade de erro pode ser reduzida simplesmente alterando-se um parâmetro interno do algoritmo. Naquele caso, teria sido o tamanho do intervalo do qual o inteiro w é sorteado. Se, ao invés de utilizarmos um intervalo de tamanho $100d$, tivéssemos utilizado um de tamanho $1000d$, a probabilidade de erro teria sido menor ou igual a 1/1000, e não 1/100. Há, no entanto, um limite — imposto pelas características da máquina ou da linguagem utilizada — para esse intervalo, como há sempre um limite para o ajuste do "parâmetro interno", qualquer que seja. Além disso, evidentemente, nem sempre é inerente ao próprio algoritmo um tal parâmetro "ajustável" como o tamanho do intervalo, naquele caso. Já o número de repetições independentes de um algoritmo é ilimitado, sendo esta sim a maneira usualmente adotada para se reduzir a probabilidade de erro a níveis tão baixos quanto se queira.

Regra prática: calcule sempre a probabilidade global de erro pela probabilidade da conjunção dos erros nas independentes execuções do algoritmo. A probabilidade global de acerto é sua complementar.

1.3.2 Las Vegas

Ainda que dele consigamos apenas estimativas probabilísticas, algoritmos randomizados de Las Vegas têm, em geral, tempo de execução bom o suficiente para que seja justificada sua utilização — se é que apenas o aspecto simplicidade, tantas vezes presente, já por si só não a justificaria — e nada ficam devendo a algoritmos determinísticos quanto à qualidade de sua resposta, que está sempre correta.

O tempo computacional de um algoritmo de Las Vegas é uma *variável aleatória* e, como tal, está completamente definido por seu conjunto de *momentos*. Por não ser nosso objetivo abordar temas de probabilidade e estatística mais do que o fizemos em nossa breve porém suficiente — assim o esperamos! — revisão na seção 1.1.1, basta-nos aqui o entendimento de que, sendo uma variável aleatória cujo comportamento a análise do algoritmo torna muito bem conhecido, o tempo computacional de um algoritmo de Las Vegas pode ser e é avaliado em termos de seu *valor esperado* — e talvez variância, desvio padrão etc.

Exemplo: busca de elemento em lista com repetições

Suponha que desejamos localizar um algarismo qualquer (digamos, o 9) numa lista de tamanho n que contém todos os algarismos de 0 a 9 distribuídos em iguais quantidades, isto é, 1/10 de suas posições apresentam o algarismo 0, 1/10 de suas posições apresentam o algarismo 1 e assim por diante. Nada se sabe, no entanto, sobre a localização dos elementos.

Imagine um algoritmo determinístico para resolver este problema. Qualquer um. Aqui vão algumas sugestões (cada item corresponde a um algoritmo completo):

1. Examine uma a uma todas as posições da lista, a partir da primeira, até encontrar o primeiro 9.

2. Examine uma a uma todas as posições da lista, a partir da última e caminhando de trás para diante, até encontrar o primeiro 9.

3. Examine primeiro todas as posições ímpares da lista, isto é, a primeira, depois a terceira, quinta etc., depois (se nenhum 9 tiver ainda sido encontrado, evidentemente) venha voltando pelas posições pares de trás para diante.

4. Divida a lista em k sublistas de tamanho n/k cada: os primeiros n/k elementos irão para a primeira sublista, os n/k elementos seguintes irão para a segunda sublista e assim por diante. Examine agora o primeiro elemento de cada sublista, em seguida o segundo elemento de cada sublista, em seguida o terceiro etc. até encontrar um 9.

Agora vejamos: como se comportará o primeiro algoritmo se os elementos da lista que lhe for submetida estiverem dispostos em ordem crescente ($000\ldots0111\ldots1222\ldots2\ldots999\ldots9$)? É evidente que o algoritmo terá investigado $9n/10+1$ posições no momento em que encontrar seu desejado algarismo 9.

E o segundo algoritmo? Como evitar que gaste também um tempo muito longo percorrendo quase toda a lista, caso a entrada esteja organizada em ordem decrescente? O terceiro algoritmo também não se comportará nada bem caso os algarismos 9 apareçam nas $n/10$ primeiras posições pares da lista. E tampouco o quarto algoritmo terá melhor desempenho se os algarismos 9 ocuparem as últimas $n/10k$ posições de cada sublista...

Resumindo, *qualquer que seja* a estratégia adotada, sempre há de existir entradas que exigirão do algoritmo um tempo "ruim" (linear no tamanho da entrada, no nosso exemplo). Dependendo da aplicação e da distribuição das instâncias de entrada — por força de algum agente externo, malicioso ou não, ou ainda que intermitentemente, durante determinados períodos, por exemplo — pode ser que o algoritmo determinístico seja *constantemente* levado a ter um desempenho lento.

Seja, agora, o algoritmo seguinte:

5. Escolha, aleatória e uniformemente, uma posição qualquer, das n possíveis. Verifique-a. Repita até encontrar um 9.

Não, não é sequer preciso deixar menos simples o algoritmo adicionando algum tipo de controle das posições já examinadas. Perceba que, a cada verificação, a probabilidade de encontrarmos um 9 é de 1/10. O número de verificações, portanto, até que o primeiro 9 seja encontrado é uma simples variável aleatória geométrica cuja probabilidade de sucesso é 1/10 (vide seção 1.2.3). O valor esperado para o número de verificações a serem executadas por este algoritmo é, portanto, igual a 10, independentemente do tamanho da entrada.

Em resumo: como alternativa aos algoritmos determinísticos de tempo linear (no pior caso), conseguimos um algoritmo de Las Vegas de *tempo esperado constante* para *todas* as entradas[9].

1.3.3 Certeza ou desempenho?

Como vimos, a certeza da resposta correta dada por um algoritmo de Las Vegas pode torná-lo bastante atraente. Ocorre que, mesmo em se tratando de algoritmos de Las Vegas cujo tempo esperado é bom, não podemos saber ao certo se uma determinada execução do algoritmo demandará, talvez, tempo muito maior.

Algoritmos de Monte Carlo, por outro lado, permitem que calculemos deterministicamente seu tempo assintótico de pior caso, o que pode ser, em muitos casos, essencial.

Transformando Las Vegas em Monte Carlo

Uma maneira simples de transformarmos um algoritmo de Las Vegas em um algoritmo de Monte Carlo é: execute o algoritmo de Las

[9] É evidente que, com probabilidade baixa, o tempo de uma execução em particular de um algoritmo de Las Vegas pode ser muito maior do que seu valor esperado (vide exercício 5). Há mesmo, em nosso exemplo, uma probabilidade infinitamente pequena de que seu tempo de execução seja infinitamente grande. Se, no entanto, modificássemos ligeiramente o algoritmo proposto, evitando que uma mesma posição da lista fosse verificada mais do que uma vez, a *pior execução possível* do algoritmo demandaria tempo $O(n)$ — seria, portanto, pelo menos tão eficiente quanto qualquer algoritmo determinístico.

Vegas durante certo tempo, ou durante um número de passos limitado por certa função do tamanho da entrada. Se encontrar a resposta, retorne-a, evidentemente, e pare; do contrário, responda `NÃO`[10] após aquele número-limite de passos.

Note que o algoritmo obtido dessa forma está sempre certo quando responde `SIM` (baseado-no-sim), pois o `SIM` terá sido forçosamente respondido dentro do limite de tempo pré-estabelecido e, portanto, durante a execução normal do algoritmo de Las Vegas (cuja resposta é sempre correta). Uma resposta `NÃO`, por outro lado, pode ter sido informada durante a execução normal do Las Vegas ou, arbitrariamente, após o estouro do limite de tempo.

A probabilidade de erro ε de um algoritmo de Monte Carlo baseado-no-sim dessa forma obtido será majorada pela probabilidade de que o algoritmo de Las Vegas demande, para responder `SIM` quando a resposta correta é `SIM`[11], tempo maior do que o limite estabelecido. Como o tempo computacional do algoritmo de Las Vegas é uma variável aleatória muito bem definida, o cálculo exato dessa probabilidade — ou, pelo menos, a determinação de bons limites inferiores e/ou superiores — é factível. Seja, por exemplo, X a variável aleatória que representa o tempo computacional de nosso algoritmo de Las Vegas, e seja $\mu = \mathbf{E}[X]$. Podemos, por exemplo, definir o limite de tempo como $k\mu$ e empregarmos a desigualdade de Markov (vide seção 1.2.2) para escrevermos

$$\varepsilon \leq \mathbf{Pr}[X \geq k\mu] \leq \frac{\mathbf{E}[X]}{k\mu} = \frac{1}{k}.$$

Transformando Monte Carlo em Las Vegas

A transformação de um algoritmo de Monte Carlo de erro unilateral em um algoritmo de Las Vegas costuma ser menos eficaz: no caso de um Monte Carlo baseado-no-não, por exemplo, teríamos que repetí-lo indefinidamente até que um `NÃO` fosse encontrado. Mas e se a resposta correta for `SIM`? Rodaria um número infinito de vezes?! Bem, em alguns casos é possível (leia-se pouco custoso) fazer com

[10] Totalmente arbitrário. Poderíamos ter escolhido responder `SIM` após o limite de tempo, e teríamos, então, um algoritmo de Monte Carlo baseado-no-não.

[11] Se tivéssemos optado por um algoritmo de Monte Carlo baseado-no-não, releia-se este parágrafo substituindo-se todos os "sim" por "não".

que as sucessivas execuções do algoritmo de Monte Carlo *não sejam* independentes, de forma a se evitar a repetição das mesmas escolhas aleatórias. O algoritmo, nesse caso, pararia após todas as possíveis seqüências de escolhas terem sido exauridas, ou após certo número de escolhas distintas ter sido feito, o que pode constituir, em alguns casos (vide nota de rodapé número 5 no exemplo da identidade de polinômios da seção 1.3.1), certificado para o `SIM`[12].

Quando, no entanto, há *dois* algoritmos de Monte Carlo para um problema, um deles baseado-no-sim (que, portanto, exibe um certificado para o `SIM` com probabilidade maior ou igual a certo valor p_S caso a resposta correta seja `SIM`) e outro baseado-no-não (que, portanto, exibe um certificado para o `NÃO` com probabilidade maior ou igual a um p_N caso a resposta correta seja `NÃO`), então é sempre possível criarmos um algoritmo de Las Vegas para o problema em questão como veremos a seguir.

Seja alg_S o algoritmo de Monte Carlo baseado-no-sim e alg_N o algoritmo de Monte Carlo baseado-no-não para um determinado problema Π. Seja p o menor entre p_S e p_N. O algoritmo de Las Vegas é exibido na figura 1.3.

Como a probabilidade de que uma resposta (necessariamente correta!) seja retornada a cada iteração do algoritmo acima é maior ou igual a p, o número de iterações do algoritmo é uma variável aleatória geométrica X com probabilidade de sucesso maior ou igual a p e, portanto, o número esperado de iterações é menor ou igual a $1/p$. Sendo ambos Alg_S e Alg_N polinomiais, teremos obtido um algoritmo de Las Vegas para Π de tempo esperado igualmente polinomial.

Caso semelhante, em que a transformação de Monte Carlo em Las Vegas é sempre possível, é aquele em que se deseja *localizar* uma estrutura que possua determinada propriedade e cuja existência é sabida. Suponhamos que exista um algoritmo de Monte Carlo que encontra a estrutura desejada (por exemplo, um corte com pelo menos metade do número de arestas do grafo — vide seção 5.1.2 para o problema do corte máximo) com probabilidade p em tempo polinomial. A figura 1.4 mostra como podemos obter um algoritmo de Las Vegas polinomial de forma bastante simples.

[12]Mais uma vez, aqui, `SIM` e `NÃO` foram escolhidos arbitrariamente. O leitor pode reler todo o parágrafo trocando os "sim" por "não" e vice-versa.

Entrada:
    alg_S: algoritmo de Monte Carlo baseado-no-sim;
    alg_N: algoritmo de Monte Carlo baseado-no-não;
    I: instância do problema.
Saída:
    `SIM` ou `NÃO`, decidindo I.

LasVegas-decisão(alg_S, alg_N, I):
    repita:
        se $alg_S(I)$ retorna `SIM`:
            retorne `SIM`
        se $alg_N(I)$ retorna `NÃO`:
            retorne `NÃO`
    até alguma resposta ser retornada

Figura 1.3: Las Vegas obtido de dois Monte Carlos.

Entrada:
    alg: um algoritmo de Monte Carlo;
    I: instância do problema.
Saída:
    A estrutura desejada, presente em I.

LasVegas-localização(alg, I):
    repita:
        seja x a estrutura retornada por $alg(I)$
        se x possui a propriedade desejada, retorne x
    até a estrutura desejada ser encontrada

Figura 1.4: Las Vegas obtido de Monte Carlo para localização.

Mais uma vez, o número de iterações do algoritmo de Las Vegas será uma variável aleatória geométrica associada a uma probabilidade de sucesso p, donde o número esperado de iterações é $1/p$. Note que, aqui, é necessário que seja possível verificar em tempo polinomial se a estrutura retornada pelo algoritmo de Monte Carlo possui ou não a propriedade desejada (por exemplo, se o número de arestas do corte retornado é maior ou igual à metade do número de arestas do grafo).

1.4 Classes de complexidade

Além das classes de complexidade usuais em que são classificados os problemas de decisão de acordo com o esforço computacional que sua resolução demanda, algoritmos randomizados deram origem a novas classes, que ora listamos para rápida referência. Nas notas bibliográficas referimos o leitor a textos mais aprofundados.

1. Classes de complexidade relacionadas a algoritmos determinísticos:

   - ***EXP*** – classe dos problemas que podem ser decididos em tempo exponencial no tamanho da entrada.
   - ***P*** – classe dos problemas que podem ser decididos em tempo polinomial no tamanho da entrada.
   - ***NP*** – classe dos problemas para os quais uma resposta `SIM` pode ser verificada em tempo polinomial.
   - ***co-NP*** – classe dos problemas para os quais uma resposta `NÃO` pode ser verificada em tempo polinomial.

2. Classes de complexidade relacionadas a algoritmos randomizados:

   - ***RP*** (*randomized polynomial time*) – classe dos problemas para os quais existe algoritmo randomizado polinomial que responde `SIM` com probabilidade maior ou igual a 1/2 caso a resposta correta seja `SIM` e responde `NÃO` com probabilidade 1 caso a resposta correta seja `NÃO`[13]. Em outras pala-

[13] O valor 1/2 foi definido arbitrariamente.

vras, é a classe dos problemas para os quais há algoritmo de Monte Carlo baseado-no-sim.

- ***co-RP*** – analogamente, é a classe dos problemas para os quais existe algoritmo randomizado polinomial que responde `NÃO` com probabilidade maior ou igual a 1/2 caso a resposta correta seja `NÃO` e responde `SIM` com probabilidade 1 caso a resposta correta seja `SIM`. Em outras palavras, é a classe dos problemas para os quais há algoritmo de Monte Carlo baseado-no-não.
- ***ZPP*** (*zero-error probabilistic polynomial time*) – classe dos problemas para os quais há algoritmo randomizado de Las Vegas de tempo esperado polinomial[14].
- ***BPP*** (*bounded-error probabilistic polynomial time*) – classe dos problemas para os quais há algoritmo de Monte Carlo de erro bilateral onde tanto a probabilidade de responder `SIM` dado que a resposta correta é `SIM` quanto a probabilidade de responder `NÃO` dado que a resposta correta é `NÃO` são maiores ou iguais a 3/4.[15]
- ***PP*** (*probabilistic polynomial time*) – classe dos problemas para os quais há algoritmo de Monte Carlo de erro bilateral onde tanto a probabilidade de responder `SIM` dado que a resposta correta é `SIM` quanto a probabilidade de responder `NÃO` dado que a resposta correta é `NÃO` são maiores que 1/2.

Por definição, a classe ***BPP*** está contida na classe ***PP***. A diferença fundamental entre os problemas em ***BPP*** e os em $\boldsymbol{PP} \backslash \boldsymbol{BPP}$ é o número de repetições do algoritmo randomizado necessárias para que a probabilidade de erro seja menor do que $\varepsilon > 0$: para os primeiros, um número polinomial (no tamanho da entrada) de repetições; para os últimos, apenas com um número exponencial delas é possível chegar-se ao erro ε.

[14] O algoritmo de Las Vegas da figura 1.3 funciona como prova de que ($\boldsymbol{RP} \cap \boldsymbol{co\text{-}RP}$) $\subseteq$ ***ZPP***. Como, evidentemente, ***ZPP*** $\subseteq$ ($\boldsymbol{RP} \cap \boldsymbol{co\text{-}RP}$), temos $\boldsymbol{ZPP} = \boldsymbol{RP} \cap \boldsymbol{co\text{-}RP}$.

[15] Novamente, o valor 3/4 foi aqui escolhido arbitrariamente; é suficiente que o limite inferior para as probabilidades em questão seja igual a $1/2 + 1/\beta$, onde β é um polinômio qualquer no tamanho da entrada do problema.

1.5 Exercícios

1. Considere uma seqüência de n lançamentos de uma moeda honesta. Seja H_i o valor da diferença entre o número de caras e o número de coroas que foram obtidos nos primeiros i lançamentos, e seja $H = \max_i H_i$. Mostre que $\mathbf{E}[H_i] = \Theta(\sqrt{i})$ e que $\mathbf{E}[H] = \Theta(\sqrt{n})$.

2. Certo exame de sangue pode ser entendido como um algoritmo de Monte Carlo baseado-no-sim que, com probabilidade $p \geq 95\%$, diagnostica determinada doença X caso o dono do sangue examinado de fato a possua. Uma epidemia de X fez com que um terço dos habitantes de uma cidade estivesse com aquela doença, cujo tratamento é, no entanto, muito penoso e não deve ser administrado a pessoas sãs. Quantas vezes aquele exame precisará ser repetido até que possa ser avaliada como "desprezível" (menor do que 1%) a chance de que uma pessoa daquela cidade apresente a doença X?

3. Seja o mesmo exame de sangue do exercício 2. Não há epidemia alguma, desta vez. Um médico experiente, porém, baseado nos diversos sintomas clínicos apresentados por um paciente seu, avalia em 80% a probabilidade de que aquele paciente tenha a doença X. O exame que se segue, no entanto, não revela a existência da doença. Como aquele médico deve reavaliar sua confiança inicial de que seu paciente é um doente de X?

4. Seja um algoritmo de Monte Carlo de erro bilateral para um problema da classe ***PP***. Mostre que um número polinomial de repetições independentes do algoritmo podem não ser suficientes para reduzir a probabilidade de erro para 1/4. (Considere a taxa de erro como sendo $1/2 - 1/2^n$.)

5. Seja t o número de verificações realizadas pelo algoritmo de Las Vegas proposto para a busca de elemento em lista com repetições da seção 1.3.2. Use as desigualdades de Markov e Chebyshev para obter majorantes para a probabilidade de que t seja:

(a) maior do que uma constante k;

(b) da ordem do tamanho da entrada, ou seja, maior ou igual a cn, para uma constante c.

1.6 Notas bibliográficas

O livro de Cormen, Leiserson, Rivest e Stein [11] é largamente adotado nos cursos de graduação em estruturas de dados e algoritmos, e contém não só um bom capítulo com as ferramentas de probabilidade para algoritmos randomizados, como também seções onde discute aplicações como Quick Sort (que será visto na seção 2.2.1) e o algoritmo de Rabin para primalidade (discutido na seção 3.9). Uma referência também geral, mais recente, é o livro de Kleinberg e Tardos [32], que contém um capítulo dedicado a algoritmos randomizados.

O livro clássico para o estudo de algoritmos randomizados é o de Motwani e Raghavan [41]. Mais recentemente, foi lançado o livro de Mitzenmacher e Upfal [39], que consideramos mais indicado para uma introdução ao assunto e que contém excelente capítulo sobre desigualdades de cauda e limites de Chernoff.

Uma introdução em português é o livro de Martinhon [35], incluindo uma boa apresentação das classes de complexidade a que pertencem os problemas que podem ser resolvidos por algoritmos randomizados, bem como demonstrações detalhadas das relações de pertinência e igualdade entre as diferentes classes.

Capítulo 2

Paradigmas combinatórios e análise probabilística

Na solução de problemas combinatórios, em geral, e em particular no estudo de algoritmos randomizados, é comum nos depararmos com novas situações, modelos probabilísticos ou experimentos aleatórios que nos remetem a outros já vistos ou analisados anteriormente. Quando estudamos as variáveis aleatórias, por exemplo, e atentamos às distribuições probabilísticas mais comuns (Bernoulli, binomial etc.), o que fazemos é preparar um certo repertório de paradigmas, um arcabouço de ferramentas básicas que será, quando propício, evocado. Evitamos, assim, dispender muito tempo ou energia analisando idéias básicas em detrimento do fluxo de raciocínio demandado pelo problema específico que se está a discutir.

Veremos, agora, alguns paradigmas combinatórios que são bastante recorrentes nas análises de algoritmos randomizados: o modelo de bolas-e-latas, na seção 2.1.1, com a discussão do paradoxo do aniversário; e o paradigma do colecionador de cupons, na seção 2.1.2.

A segunda parte deste capítulo apresenta, na seção 2.2, a análise probabilística de algoritmos, forma interessante de se avaliar a performance média de algoritmos a partir de certas hipóteses a respeito

das instâncias de entrada que são a eles submetidas.

2.1 Paradigmas combinatórios

2.1.1 O modelo de bolas-e-latas

Seja a seguinte situação: existem m objetos indistingüíveis (*bolas*), cada qual a ser associado aleatoriamente a um de n objetos distintos (*latas*). Esse cenário tão simples, convenientemente chamado *modelo de bolas-e-latas*, encontra aplicação em um sem-número de problemas reais. A idéia é: cada bola será colocada com a mesma probabilidade $1/n$ em qualquer das latas.

As questões que se pretende responder são, em geral, do tipo: quantas latas permanecem vazias?, qual o número esperado de latas com mais do que k bolas?, quantas bolas se deve distribuir até que seja mais provável haver do que não haver alguma lata com mais do que uma bola?, qual o número de bolas na lata mais cheia? etc.

O paradoxo do aniversário

O caso particular em que o número de latas é igual a 365 remete-nos ao famoso "paradoxo do aniversário", que, de paradoxo, não tem nada — exceto o fato de serem as probabilidades envolvidas algo contra-intuitivas para a maioria das pessoas.

Há 23 pessoas num campo de futebol, durante uma partida (onze jogadores em cada time, mais o juiz). Qual a probabilidade de que haja duas pessoas quaisquer naquele grupo aniversariando exatamente no mesmo dia do ano? Para verificar a contra-intuitividade da resposta correta, normalmente o proponente do "paradoxo" não exige o cálculo exato da probabilidade em questão, mas apenas uma estimativa, a que o desprevinido interpelado costuma responder algo como "baixa" ou "muito baixa". Na verdade, a probabilidade exata é de 50,72972343%, sendo, portanto, mais provável haver do que não haver algum dia do ano com mais de um aniversariante dentre aquelas 23 pessoas[1].

[1] Com menos do que 23 pessoas, a probabilidade de haver aniversários coincidentes é menor do que 50%.

Numa situação mais geral do modelo de bolas-e-latas com n latas e m bolas, pode-se calcular a probabilidade de não haver qualquer lata com mais do que uma bola raciocinando em cima do seguinte experimento: sortearemos uma lata, aleatória e uniformemente, para colocar cada uma das bolas, uma por vez. Seja A_i $(i = 1, \ldots, m)$ o evento em que a i-ésima bola *não* é colocada numa lata que já possua alguma bola. A probabilidade p que buscamos é

$$\begin{aligned} p &= \mathbf{Pr}\left[\bigcap_{i=1}^{m} A_i\right] \\ &= \prod_{i=1}^{m} \mathbf{Pr}\left[A_i | \bigcap_{j<i} A_j\right]. \end{aligned}$$

O valor $\mathbf{Pr}[A_i | \bigcap_{j<i} A_j]$ é a probabilidade de que a lata escolhida para a i-ésima bola não seja uma das latas já ocupadas, dado que as $i-1$ bolas anteriores foram posicionadas cada qual numa lata distinta, sendo portanto igual a $1-(i-1)/n$. Substituindo na expressão de p, temos

$$p = \prod_{i=1}^{m} \left(1 - \frac{i-1}{n}\right) = \prod_{i=1}^{m} \frac{n-i+1}{n}.$$

A probabilidade de que *exista* alguma lata com mais do que uma bola é, evidentemente, $1-p$. Resolvendo a equação $1-p = 1/2$, consegue-se chegar, após algumas aproximações envolvendo exponenciais, a $m \approx \sqrt{2n \ln 2} = O(\sqrt{n})$.

O paradoxo do aniversário é apenas uma das situações que se pode associar ao modelo de bolas-e-latas. O valor crítico $m = O(\sqrt{n})$ como limítrofe dos casos em que a probabilidade de "colisão" é menor ou maior do que $1/2$ é bem conhecido e aparece, com alguma freqüência, em aplicações do modelo.

A análise probabilística do algoritmo de ordenação Bucket Sort que veremos na seção 2.2.3 assume um modelo de bolas-e-latas para as possíveis entradas do problema. O paradigma do colecionador de cupons, que veremos a seguir, é mais um de seus desdobramentos.

2.1.2 O colecionador de cupons

Seja o seguinte experimento: sorteia-se, aleatória e uniformemente, um elemento de um conjunto $D = \{d_1, d_2, \ldots, d_n\}$ de objetos distintos. Faz-se isto repetidamente, gerando uma seqüência de variáveis aleatórias independentes $X_1, X_2, \ldots$, cada X_i indicando o índice do elemento obtido no $i-$ésimo sorteio.

Costuma-se pensar neste experimento como o de um colecionador que adquire itens para sua coleção aleatoriamente, cada item do universo de todos os n itens do que seria a coleção completa podendo ser adquirido com a mesma probabilidade $1/n$ a cada vez. Imagine, por exemplo, caixas de cereal que trazem, em seu interior, cupons numerados de 1 a 10, um cupom por caixa.

Define-se a variável aleatória $W_{n,k}$ como sendo o número de sorteios realizados até que se tenha obtido k itens distintos, dos n existentes. Ou seja, o número de caixas de cereal que foram compradas até que o colecionador tivesse k cupons distintos em sua coleção. Em geral, estamos interessados em $\mathbf{E}[W_{n,k}]$.

Especial atenção é dada à variável aleatória $W_{n,n}$, que é o número de caixas que o colecionador precisa comprar até completar sua coleção com todos os n cupons. Como se vê, o *paradigma do colecionador de cupons* não é mais do que o modelo de bolas-e-latas com n latas e um número ilimitado de bolas (as caixas são as bolas e o cupom existente na i-ésima caixa é a lata que recebe a i-ésima bola). Nesse caso, estamos interessados na quantidade de bolas que deve ser distribuída até que não haja mais latas vazias.

Para $i = 1, 2, \ldots, n$, seja Z_i a quantidade de caixas de cereal necessárias até que o número de cupons distintos possuídos pelo colecionador aumente de $i-1$ para i. Ora, quando o colecionador possui $i-1$ cupons distintos, a probabilidade p_i de que um cupom, adquirido aleatória e uniformemente do universo de todos os n coupons, seja um dos que ele ainda não possui é igual a $1-(i-1)/n$. Cada Z_i é, portanto, uma variável aleatória geométrica com probabilidade de sucesso p_i, donde a esperança de $Z_i (i = 1, \ldots, n)$ é $\mathbf{E}[Z_i] = n/(n-i+1)$. Escrevendo $W_{n,k} = Z_1 + Z_2 + \cdots + Z_k$, chegamos ao valor esperado

$$\mathbf{E}[W_{n,k}] = \sum_{i=1}^{k} \frac{n}{n-i+1}.$$

Para o importante caso em que $k = n$, temos

$$\mathbf{E}[W_{n,n}] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i}.$$

Lembrando que o número harmônico $H(n) = \sum_{k=1}^{n} 1/k = \ln n + c$, para uma constante c, chegamos ao número esperado $\mathbf{E}[W_{n,n}] = n \ln n + \Theta(n)$ de caixas de cereal compradas até que todos os n cupons distintos tenham sido obtidos pelo colecionador.

Exemplo: mapeamento de roteadores

Diversos são os problemas que admitem algoritmos randomizados cuja análise recai, de alguma forma, no paradigma do colecionador de cupons: o problema dos casamentos estáveis, o problema do ciclo hamiltoniano em grafos aleatórios, e muitos outros.

Uma aplicação bem simples é aquela em que uma mensagem é enviada de uma origem (cliente) a um destino (servidor), numa rede de computadores. A mensagem é quebrada em pacotes de informação, cada qual transmitido através de um caminho fixo de roteadores (o mesmo caminho para todos os pacotes). Suponha que o servidor precise saber quais são os roteadores existentes no caminho pelo qual passou a mensagem que lhe foi enviada pelo cliente (para que, em caso de erro, por exemplo, possa investigar possíveis roteadores que estejam corrompendo a informação).

Uma maneira simples seria, evidentemente, fazer com que cada pacote armazenasse, em um campo específico de seu descritor (*header*), a seqüência de roteadores pela qual passou. Ocorre que pode não haver espaço suficiente, no descritor de cada pacote, para guardar a identificação de *todos* os roteadores do caminho percorrido, sem falar da carga extra e redundante de informação que se faria transmitir pela rede.

Uma abordagem randomizada seria: cada pacote guardaria a identificação de apenas um dos roteadores pelos quais passasse. A escolha de *qual* roteador deverá ter sua identificação armazenada no descritor de um dado pacote deve ser feita aleatoriamente, de forma uniforme. Ou seja, para cada pacote transmitido, todos os n roteadores daquele caminho terão a mesma probabilidade $1/n$ de ser o

escolhido para ter sua identificação armazenada. Do ponto de vista do servidor, cada pacote que chega é como uma caixa de cereal contendo — com distribuição uniforme de probabilidade — algum dos n cupons existentes. O número esperado de pacotes que precisam ser recebidos até que o servidor tenha tido conhecimento de todos os roteadores daquele caminho é, como foi visto, $n \ln n + \Theta(n)$.

Resta-nos resolver a questão muito prática de como fazer para que um pacote em trânsito tenha a mesma probabilidade $1/n$ de armazenar a identificação de qualquer um dos n roteadores que encontrará pela frente. Pode mesmo ser o caso em que *não se saiba de antemão* o número de roteadores do caminho em questão.

Isto pode ser resolvido usando-se a técnica conhecida como *reservoir sampling*. Seja um caminho de n roteadores. Consideremos agora um pacote que começa a ser transmitido. Quando passar pelo primeiro roteador, ele armazena a identificação daquele roteador com probabilidade 1. Em seguida, quando passar pelo k-ésimo roteador, decide *trocar* a identificação que está no momento armazenando pela identificação deste k-ésimo roteador com probabilidade $1/k$. O que precisamos mostrar é que, dessa forma, garantimos distribuição uniforme de probabilidade entre os roteadores[2], ou seja, mostrar que, para $k = 1, \ldots, n$, a probabilidade de que o k-ésimo roteador seja aquele cuja identificação será apresentada por um pacote qualquer ao servidor é $1/n$.

Para que o k-ésimo roteador seja o escolhido final, não importa qual seja o roteador armazenado no descritor do pacote em trânsito no momento em que este chega ao k-ésimo roteador. Tudo o que precisa acontecer é que, naquele momento, o pacote opte por trocar a identificação correntemente armazenada pela do k-ésimo (o que ocorre com probabilidade $1/k$) e que, chegando em cada roteador que lhe está à frente no caminho até o servidor, o pacote opte por *não* trocar.

Seja T_k (respectivamente, $\overline{T_k}$) o evento em que opta-se (resp. não se opta) por trocar pela do k-ésimo roteador a identificação que está correntemente armazenada no descritor do pacote no momento em

[2]Note que, ainda que *nós* saibamos que $k = 1, \ldots, n$, por hipótese, o número n de roteadores não precisa ser conhecido pelo pacote, já que apenas k tem algum papel na decisão sobre trocar ou não o roteador cuja identificação está sendo armazenada.

que este passa pelo k-ésimo roteador. O evento F_k, em que a identificação do k-ésimo roteador é a que chega ao servidor por meio de um dado pacote, corresponde a $T_k \cap \overline{T_{k+1}} \cap \overline{T_{k+2}} \cap \ldots \cap \overline{T_n}$. Para $k = 1, \ldots, n$, temos

$$\mathbf{Pr}[F_k] = \mathbf{Pr}[T_k] \prod_{i=k+1}^{n} \mathbf{Pr}[\overline{T_i}],$$

pois todas as escolhas são independentes. Como

$$\mathbf{Pr}[\overline{T_i}] = 1 - \frac{1}{i} = \frac{i-1}{i},$$

a expressão de $\mathbf{Pr}[F_k]$ fica

$$\mathbf{Pr}[F_k] = \frac{1}{k} \cdot \frac{k}{k+1} \cdot \frac{k+1}{k+2} \cdots \frac{n-2}{n-1} \cdot \frac{n-1}{n},$$

e, após os cancelamentos de numeradores e denominadores em cascata, chegamos a $\mathbf{Pr}[F_k] = 1/n$, como queríamos demonstrar.

2.2 Análise probabilística de algoritmos

Até agora, vimos como a presença de escolhas aleatórias faz com que diferentes execuções de um algoritmo *para uma mesma entrada* possam levar tempos distintos ou até mesmo apresentar respostas distintas. No entanto, vimos também como a presença dessa mesma aleatoriedade permite-nos conseguir algoritmos simples e eficientes, que são assim considerados quando a esperança da variável aleatória em que se constitui seu tempo de execução é polinomial no tamanho da entrada.

Por outro lado, diferentes execuções de um algoritmo determinístico para uma mesma entrada sempre resultarão em resposta idêntica e após exatamente o mesmo número de passos. Dessa forma, algoritmos determinísticos são considerados eficientes quando executam um número polinomial de passos *para a pior entrada possível*.

Na prática, porém, nem sempre o algoritmo mais adequado é o que apresenta o melhor tempo para a pior entrada possível. Pode ser que,

Entrada:
    S: conjunto de elementos comparáveis.
Saída:
    Os elementos de S em ordem crescente.

quickSort(S):
    se $|S| < 2$, retorne S
    tome x, o primeiro elemento de S, como pivô
    crie duas listas S_1 e S_2 inicialmente vazias
    para cada elemento y de S:
        se $y < x$, coloque y em S_1
        se $y > x$, coloque y em S_2
    retorne quickSort(S_1), x, quickSort(S_2)

Figura 2.1: Algoritmo Quick Sort para ordenação.

para entradas típicas de uma aplicação qualquer, algoritmos teoricamente menos eficientes tenham desempenho muito melhor! E é justamente o conhecimento da "entrada típica" — ou, melhor dizendo, das freqüências de ocorrência (probabilidades) das diferentes entradas — que permite, em muitos casos, avaliarmos a performance de algoritmos de forma sensível a um modelo probabilístico das possíveis entradas para o problema. É o que chamamos de *análise probabilística* de algoritmos (randomizados ou determinísticos!), como veremos nos exemplos que se seguem.

2.2.1 Quick Sort

Diversos algoritmos existem para o famoso problema da ordenação. A entrada, cujos elementos se deseja dispor em ordem, digamos, crescente, é uma seqüência $x_1, x_2, \ldots, x_n$ de elementos comparáveis dois-a-dois. Podemos considerar, por simplicidade, que os elementos sejam números e que não haja dois números iguais na seqüência.

Um dos algoritmos mais simples de ordenação é o chamado *Quick Sort*, cujo pseudo-código encontra-se na figura 2.1.

A idéia é a de escolher arbitrariamente um dos n elementos da lista (o primeiro, por exemplo) e utilizá-lo como o "pivô" de uma

estratégia do tipo divisão-e-conquista. A lista numérica original é quebrada em duas sub-listas: uma contendo os elementos que são menores do que o pivô, outra contendo aqueles que são maiores do que o pivô. Em seguida, as sub-listas são recursivamente submetidas à mesma estratégia de ordenação. O algoritmo retorna, então, nesta ordem, os elementos já ordenados da primeira sub-lista, o pivô, e os elementos já ordenados da segunda sub-lista. Note que listas vazias ou unitárias, por já se encontrarem trivialmente ordenadas, são retornadas imediatamente, não dando origem a novas chamadas recursivas.

Um algoritmo tal como o descrito permite-nos facilmente pensar numa entrada ruim, ou seja, uma que exija uma grande quantidade de operações básicas de comparação entre dois números. Imaginemos, por exemplo, uma entrada *já ordenada* na forma $x_1 < x_2 < \cdots < x_n$. Como o pivô é escolhido como sendo o primeiro elemento da lista, a primeira divisão em sub-listas dará origem a uma sub-lista vazia (dos elementos menores que o pivô x_1) e a uma sub-lista com $n-1$ elementos (aqueles maiores do que o pivô). Sendo assim, a chamada recursiva ao Quick Sort para ordenar a sub-lista não-vazia constituirá problema praticamente idêntico ao original, apenas com um elemento a menos: o próprio pivô. A ordenação da lista contendo $n-1$ elementos, por sua vez, fará com que $n-2$ comparações sejam executadas, ao término das quais, novamente, uma sub-lista vazia (dos elementos menores do que o pivô x_2) e uma contendo $n-2$ elementos (maiores do que x_2) terão sido obtidas. E assim sucessivamente, até que as chamadas recursivas sejam interrompidas no momento em que as comparações contra o $(n-1)$-ésimo pivô tenham dado origem a uma sub-lista com apenas um elemento (além, é claro, de uma sub-lista vazia). O número total de comparações para uma tal entrada[3] seria, portanto, igual a

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2).$$

Digamos, agora, que nós *saibamos de antemão* que nossa entrada

[3]Salientamos que entradas pré-ordenadas não são as únicas que exigem tempo quadrático do algoritmo Quick Sort. Qualquer entrada em que aconteça de o pivô ser sistematicamente escolhido de forma a dividir a lista atual, a cada chamada recursiva, em sub-listas de tamanho muito desequilibrado (uma delas de tamaho limitado por uma constante, por exemplo), são igualmente ruins.

típica não é do tipo pré-ordenado, nem foi escolhida por qualquer forma de adversário malicioso, mas seja uma seqüência escolhida *uniforme e aleatoriamente* do universo de todas as possíveis permutações de seus elementos. Qual o comportamento esperado do Quick Sort, nesse caso? Em outras palavras, qual o número médio de comparações que serão executadas para entradas dessa natureza?

Seja $S = x_1, x_2, \ldots, x_n$ uma instância de entrada para o algoritmo Quick Sort (obedecendo a nosso hipotético modelo probabilístico da entrada) e seja $S' = y_1, y_2, \ldots, y_n$ a saída que o algoritmo se dispõe a apresentar. Ou seja, a seqüência S' é exatamente a seqüência S em ordem crescente.

Seja X uma variável aleatória que corresponde ao número total de comparações efetuadas. Interessa-nos $\mathbf{E}[X]$. Como quaisquer y_i e y_j são comparados no máximo uma única vez ao longo do algoritmo (quando um deles for escolhido como pivô de uma lista que ainda contenha o outro), podemos escrever

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{1 \le i < j \le n} Y_{ij}\right] = \sum_{1 \le i < j \le n} \mathbf{E}[Y_{ij}],$$

onde Y_{ij} é um indicador de Bernoulli que assume valor 1 quando os números y_i e y_j são comparados durante a execução do Quick Sort para aquela entrada. A segunda igualdade é possível pela linearidade da esperança.

Uma vez que o valor esperado de uma variável aleatória de Bernoulli é igual à probabilidade de que ela assuma valor 1, só o que precisamos saber é a probabilidade p_{ij} de que y_i e y_j $(i < j)$ sejam comparados ao longo da execução do algoritmo. Esta probabilidade é igual à probabilidade de *não haver*, na seqüência S, qualquer elemento do conjunto $\{y_{i+1}, y_{i+2}, \ldots, y_{j-2}, y_{j-1}\}$ aparecendo à esquerda de ambos y_i e y_j (de forma que algum elemento maior que y_i e menor que y_j seria escolhido como pivô *antes* que y_i ou y_j o fossem, separando-os em sub-listas distintas). Em outras palavras, interessa-nos a probabilidade de que y_i ou y_j seja o elemento mais à esquerda, em S, dentre aqueles do conjunto $\{y_i, y_{i+1}, \ldots, y_{j-1}, y_j\}$. Como, por hipótese, a entrada foi escolhida aleatória e uniformemente de todas as permutações possíveis dos elementos de S', todos os elementos

entre y_i e y_j (ambos incluídos) em S' têm a mesma chance de ser, dentre eles, o elemento mais à esquerda em S. Conseqüentemente,

$$p_{ij} = \frac{2}{j-i+1},$$

resolvendo nosso somatório:

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{1\le i<j\le n} \frac{2}{j-i+1} \\
&= 1+\frac{2}{3}+\frac{2}{4}+\cdots+\frac{2}{n-2}+\frac{2}{n-1}+\frac{2}{n} \\
&\quad +1+\frac{2}{3}+\frac{2}{4}+\cdots+\frac{2}{n-2}+\frac{2}{n-1} \\
&\quad +1+\frac{2}{3}+\frac{2}{4}+\cdots+\frac{2}{n-2} \\
&\quad +\cdots \\
&\quad +1+\frac{2}{3} \\
&\quad +1.
\end{aligned}$$

Ao re-escrevermos o somatório original reagrupando as parcelas iguais (que formam as colunas do desenvolvimento acima), obtemos

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{k=2}^{n}(n-k+1)\frac{2}{k} \\
&= (n+1)\sum_{k=2}^{n}\frac{2}{k} - 2(n-1) \\
&= (2n+2)\sum_{k=1}^{n}\frac{1}{k} - 4n.
\end{aligned}$$

Como já o lembramos na seção 2.1.2, $H(n) = \sum_{k=1}^{n} 1/k = \ln n + c$, para uma constante c. Sendo assim, obtemos, finalmente, $\mathbf{E}[X] = 2n\ln n + cn = O(n\log n)$.

Como pudemos ver, portanto, por meio da análise probabilística, o tempo médio do Quick Sort para entradas obtidas do modelo considerado é tão bom quanto $O(n\log n)$.

Entrada:
    S: conjunto de elementos comparáveis.
Saída:
    Os elementos de S em ordem crescente.

quickSortRandomizado(S):
    escolha, aleatória e uniformemente, uma permutação de S
    retorne quickSort(S)

Figura 2.2: Algoritmo Quick Sort Randomizado para ordenação.

2.2.2 Quick Sort Randomizado

Na análise que acabamos de ver, estivemos diante de algoritmo perfeitamente determinístico. A aleatoriedade do experimento esteve presente no momento da escolha da *entrada* do problema, e não durante a execução do algoritmo. Há casos, porém, em que podemos inserir a aleatoriedade dentro do próprio algoritmo, na forma de uma pequena etapa de pré-processamento. Procedendo assim, a boa performance do algoritmo deixa de depender de um modelo pré-estabelecido para as entradas que lhe são submetidas, como que *forçando-as* internamente a conformarem com a distribuição probabilística do modelo desejado.

Um exemplo muito simples é o do próprio algoritmo Quick Sort, que pode ser facilmente transformado num Quick Sort *Randomizado*, como esboçado na figura 2.2.

Sendo idênticas as análises nos dois casos, pode-se ver que o tempo esperado do Quick Sort Randomizado *para uma entrada qualquer* será o mesmo $O(n \log n)$ que tem o Quick Sort determinístico para uma entrada obtida aleatória e uniformemente do universo de todas as permutações de seus elementos[4].

[4]Uma versão bem conhecida do Quick Sort Randomizado é aquela em que não há pré-processamento algum da entrada, mas o *pivô* é escolhido aleatoriamente, a cada passo, entre todos os elementos da lista. É fácil ver que o tempo esperado das duas versões randomizadas do algoritmo é absolutamente o mesmo.

A inserção do pré-processamento da entrada para transformar algoritmos determinísticos em algoritmos randomizados com bom tempo esperado nem sempre é possível.

2.2.3 Bucket Sort

Damos agora um segundo exemplo de algoritmo determinístico para o problema da ordenação. O algoritmo é o chamado *Bucket Sort*, que, dada uma certa distribuição probabilística das instâncias de entrada, roda em tempo linear $O(n)$, quebrando portanto o conhecido limite inferior $O(n \log n)$ para ordenações baseadas em comparação.

O modelo probabilístico da entrada que assumimos aqui é aquele em que uma entrada é formada por $n = 2^m$ números, que se deseja ordenar, e cada número foi escolhido aleatória e uniformemente do intervalo $[0, 2^k[$, onde $k > m$.

O algoritmo Bucket Sort funciona em duas etapas: na primeira, cada um dos n números que se deseja ordenar é colocado em uma de n listas, ou *buckets*. Os *buckets* são rotulados de 0 a $n-1$ em binário, ou seja, "000...000", "000...001", "000...010", "000...011", ..., "111...111", onde o número de dígitos é igual a m. Cada *bucket* α conterá os elementos da entrada que, se representados como numerais de k dígitos binários, apresentam seus m primeiros dígitos correspondendo ao rótulo de α.

Por exemplo, suponha que tenhamos $n = 8$ números para ordenar, escolhidos aleatória e uniformemente do intervalo $[0, 128[$, por exemplo[5]: $126, 7, 2, 39, 70, 91, 120$ e 66. Serão criados 8 *buckets* conforme indicado abaixo juntamente com os elementos da entrada que serão a cada qual atribuídos:

"000": 7 (*000*0111), 2 (*000*0010)
"001": vazio
"010": 39 (*010*0111)
"011": vazio
"100": 70 (*100*0110), 66 (*100*0010)
"101": 91 (*101*1011)
"110": vazio
"111": 126 (*111*1110), 120 (*111*1000)

[5] Neste exemplo, $m = 3, k = 7$.

Assumindo que cada elemento pode ser posicionado no seu devido *bucket* em tempo constante, a primeira etapa roda em tempo $O(n)$.

Na segunda etapa, cada *bucket* é ordenado usando, para isso, qualquer algoritmo de ordenação de tempo quadrático (Quick Sort, por exemplo). Finalmente, concatena-se as listas correspondentes a cada um dos *buckets* já ordenados, e isto é tudo. Queremos mostrar que o tempo esperado da segunda etapa é também $O(n)$.

Seja X_j o número de elementos da entrada que caem no bucket j. Dado o modelo probabilístico da entrada que assumimos, cada elemento tem probabilidade idêntica de pertencer a qualquer um dos n *buckets* — e, portanto, igual a $1/n$. Note o leitor que estamos exatamente diante do modelo de bolas-e-latas (vide seção 2.1.1), onde os *buckets* são as latas e os números que se deseja ordenar são as bolas. Sendo assim, X_j é uma variável aleatória binomial $B(n, 1/n)$.

Para ordenar os X_j elementos do j-ésimo *bucket*, um algoritmo de ordenação de tempo quadrático levará tempo $c(X_j)^2$ para alguma constante c e, dessa forma, o valor esperado para o tempo total X da segunda etapa do Bucket Sort será dado por:

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{j=1}^{n} c(X_j)^2\right] = cn\mathbf{E}[X_1^2],$$

onde usamos a linearidade da esperança e o fato de que todas as variáveis X_j têm idêntica distribuição de probabilidade.

Já vimos, na seção 1.2.3, que, para uma variável aleatória X com distribuição binomial $B(n, p)$, temos $\mathbf{E}[X^2] = n(n-1)p^2 + np$. Como todas as nossas X_j e, em particular, X_1, são binomiais $B(n, 1/n)$, temos $\mathbf{E}[X_1^2] = 2 - 1/n < 2$ e, portanto, o tempo esperado de toda a segunda etapa é no máximo $2cn$.

Conseqüentemente, o algoritmo Bucket Sort roda em tempo linear $O(n)$ para entradas do modelo probabilístico considerado.

2.3 Exercícios

1. Suponha uniforme a distribuição de nascimentos entre os sete dias da semana.

(a) Qual o tamanho mínimo de um grupo de pessoas escolhidas aleatoriamente para que haja pelo menos duas pessoas, naquele grupo, nascidas num mesmo dia da semana com probabilidade $p = 1$?

(b) Qual o tamanho mínimo de um grupo de pessoas escolhidas aleatoriamente para que haja pelo menos duas pessoas, naquele grupo, nascidas num mesmo dia da semana com probabilidade $p \geq 1/2$?

2. Um dado honesto é lançado repetidamente até que todos os seis resultados possíveis tenham sido obtidos. Avalie a probabilidade de que o número de lançamentos seja maior ou igual a 10.

3. Suponha um sistema de computação distribuída no qual processos disputem a utilização de recursos mas desistam, temporariamente, em face de conflitos com outros processos. O seguinte modelo de bolas-e-latas — em que as bolas são os processos e as latas são os recursos em disputa — descreve o funcionamento do sistema: a cada *rodada*, bolas são atiradas independentemente, e de forma aleatória e uniforme, nas n latas existentes. Bolas que, após todas terem sido distribuídas, estejam localizadas sozinhas numa lata serão imediatamente atendidas (o recurso é concedido ao processo que o requisita) e tiradas do conjunto de bolas sob consideração. As demais bolas serão distribuídas novamente na rodada seguinte. Este procedimento continua até que não haja mais bolas (isto é, até que todos os processos tenham sido atendidos).

(a) Se há b bolas no início de uma rodada, qual o número esperado de bolas no início da rodada seguinte?

(b) Suponha que, a cada rodada, o número de processos atendidos seja exatamente o valor esperado do número de bolas que não compartilham sua lata com nenhuma outra. Mostre que um número inicial de n bolas é totalmente servido em $O(\log \log n)$ rodadas.

(*Dica*: se x_j é o número esperado de bolas após j rodadas, demonstre e use o fato de que $x_{j+1} \leq x_j^2/n$.)

Entrada:
Um conjunto S de elementos comparáveis e um inteiro k.
Saída:
O k-ésimo elemento de S.

seleção(S, k):
se $|S| = 1$, retorne o único elemento de S
tome x, o primeiro elemento de S, como pivô
crie duas listas S_1 e S_2 inicialmente vazias
para cada elemento y de S:
se $y < x$, coloque y em S_1
se $y > x$, coloque y em S_2
se $k \leq |S_1|$, retorne seleção(S_1, k)
se $k - 1 = |S_1|$, retorne x
senão retorne seleção$(S_2, k - 1 - |S_1|)$

Figura 2.3: Algoritmo para seleção do k-ésimo elemento.

4. Dados um conjunto S e um inteiro k, definimos o k-ésimo elemento de S como o elemento na k-ésima posição da lista que contém os elementos de S em ordem crescente. É possível determinar o k-ésimo elemento de S em tempo $O(n \log n)$ usando ordenação. Analise o tempo médio do algoritmo da figura 2.3, que determina o k-ésimo elemento sem entretanto ordenar S. (Considere que S é escolhido aleatória e uniformemente do universo de todas as possíveis permutações de seus elementos.)

5. Seja $x_1, x_2, \ldots, x_n$ uma lista de n números distintos. Dizemos que x_i e x_j estão invertidos se $i < j$ mas $a_i > a_j$. O algoritmo de ordenação conhecido como *Bubble Sort* troca a posição de dois números vizinhos na lista que estejam invertidos, até que não haja mais vizinhos invertidos — quando então a lista estará ordenada. Suponha que a entrada do algoritmo Bubble Sort é uma permutação escolhida aleatória e uniformemente de todas as possíveis permutações de um conjunto de n números distintos. Determine o número esperado de inversões realizadas pelo algoritmo para ordenar os elementos daquela entrada.

2.4 Notas bibliográficas

A lista de aplicações do modelo de bolas-e-latas é extensa. Alguns exemplos recentes o leitor encontrará nos artigos de Edmonds e Pruhs [17] e de Strumpen e Krishnamurthy [51]. Uma aplicação importante é o estudo dos *grafos aleatórios*, em especial aqueles do largamente utilizado modelo $G_{n,M}$, onde cada possível grafo com n vértices e M arestas tem a mesma probabilidade de ser obtido do experimento aleatório que os gera[6]. Grafos aleatórios foram definidos por Erdős e Rényi em 1959 [18], sendo hoje o livro de Bollobás [3] uma das melhores referências no tema.

O paradigma do colecionador de cupons é também largamente empregado e aparece em algoritmos randomizados para uma série de problemas clássicos. Bons exemplos são o problema dos casamentos estáveis [24, 28], para o qual o leitor encontra algoritmo de Las Vegas sendo discutido no livro de Motwani e Raghavan [41], e o problema da paginação, com um belo algoritmo randomizado proposto por Fiat *et al.* [21].

O problema do ciclo hamiltoniano em grafos aleatórios [9, 10] não apenas admite algoritmo randomizado eficiente (baseado no paradigma do colecionador de cupons) como é também exemplo bastante representativo da técnica da análise probabilística de algoritmos, como é apresentado bastante didaticamente por Mitzenmacher e Upfal [39].

Vasta é a literatura sobre algoritmos de ordenação (como os algoritmos Quick Sort e Bucket Sort apresentados neste capítulo). Uma boa referência é o terceiro volume do célebre trabalho de Knuth [33]. Para algoritmos randomizados no tema, publicação recente é o artigo de Dean [16].

[6] O modelo $G_{n,p}$ é também muito conhecido, e é aquele em que grafos aleatórios de n vértices são obtidos adicionando-se cada possível aresta entre dois de seus vértices com probabilidade p.

Capítulo 3

Primalidade

O primeiro sistema de criptografia com chave pública foi desenvolvido por Rivest, Shamir e Adleman [48] e é hoje largamente utilizado para garantir a segurança e a privacidade na troca de informações através da rede mundial de computadores. O RSA, assim chamado devido às iniciais de seus criadores, atinge seus objetivos porque é relativamente fácil encontrar números primos grandes e é praticamente impossível fatorar o produto de dois destes números[1]. Neste capítulo, apresentamos um algoritmo randomizado, desenvolvido por Rabin [45], que decide em tempo polinomial se determinado número é primo. Este algoritmo é muito empregado, na prática, para a importante tarefa de se localizar primos grandes.

Em 2002, uma descoberta matemática foi noticiada na primeira página dos principais jornais do mundo. Agrawal, Kayal e Saxena obtiveram um algoritmo polinomial, batizado AKS, para decidir a primalidade de um inteiro [1]. Foge ao escopo deste texto, no entanto, a análise do determinístico AKS. Além disso, para todos os fins práticos, o algoritmo randomizado de Rabin lhe é superior. Em primeiro lugar, sua mecânica — bem como a matemática necessária para

[1] Shor [50] desenvolveu um algoritmo quântico rápido para fatoração. Contudo, para ser executado, o computador quântico necessitaria de pelo menos tantos *q-bits* (bits quânticos) quanto o número de dígitos na base 2 do número a ser fatorado — a construção de tal computador está absolutamente fora de nosso alcance, na atual fase do conhecimento humano.

justificá-la — é bem menos complexa que a demandada pelo AKS. Em segundo, o algoritmo de Rabin tem custo $O(\log^2 n(\log\log n)^{O(1)})$, enquanto o custo do AKS é $O(\log^6 n(\log\log n)^{O(1)})$. Mais uma vez, portanto, sai-se melhor o algoritmo randomizado em simplicidade e eficiência.

Veremos, das seções 3.1 a 3.8, conteúdo matemático básico para o entendimento do algoritmo de Rabin. Na seção 3.1, definimos operações de adição e multiplicação em $\mathbb{Z}_n$. O algoritmo de Euclides para o cálculo do maior divisor comum de dois inteiros é descrito e analisado na seção 3.2. Uma aplicação deste algoritmo é encontrada na seção 3.3, ao recordarmos o Teorema Fundamental da Aritmética. Na seção 3.4, estabelecemos o Pequeno Teorema de Fermat como uma conseqüência do Teorema de Euler e estudamos uma de suas variantes, central para a descrição do algoritmo de Rabin. Na seção 3.5, abordamos o Teorema Chinês do Resto. Na seção 3.6, mostramos que qualquer elemento inversível de $\mathbb{Z}_n$ é potência de um elemento fixo quando n é primo ou o quadrado de um primo. Na seção 3.7, definimos quando um natural n é pseudoprimo com respeito a uma base e calculamos a probabilidade disto ocorrer quando n é composto. O custo das operações aritméticas usuais é analisado na seção 3.8, onde é também apresentado um algoritmo eficiente para o cálculo da exponenciação em $\mathbb{Z}_n$. Finalmente, na seção 3.9, introduzimos a pérola deste capítulo: o algoritmo randomizado de Rabin para decidir primalidade. A seção 3.10 fecha o capítulo com uma exposição do algoritmo RSA para criptografia, pretendendo justificar o enorme interesse que há em torno de algoritmos eficientes para encontrar números primos grandes.

3.1 Aritmética modular

Dado um número natural n, denotamos por $[n]$ o conjunto dos possíveis restos de uma divisão por n, isto é, $[n] = \{0, 1, 2, \ldots, n-1\}$. Claramente, $[n]$ não é fechado com relação às operações usuais de adição e multiplicação: quando efetuamos uma destas operações em dois elementos de $[n]$, o resultado pode não estar em $[n]$. Este problema pode ser contornado definindo-se novas operações de adição e multiplicação neste conjunto: o resultado destas novas operações será

o resto da divisão por n do resultado obtido quando estas operações são realizadas normalmente em $\mathbb{Z}_n$. Formalmente, para $a, b \in [n]$, definimos

$$\begin{aligned} a \oplus b &= \text{res}(a+b, n) \\ a \otimes b &= \text{res}(ab, n), \end{aligned}$$

onde $\text{res}(x, n)$ denota o resto da divisão do inteiro x por n. A *aritmética modular* estabelece outro enfoque para a definição e a aplicação destas operações, dando origem a demonstrações elegantes e eliminando vários problemas técnicos.

Para um inteiro a, considere o seguinte subconjunto de $\mathbb{Z}$:

$$\overline{a} = \{b \in \mathbb{Z} : \text{res}(a, n) = \text{res}(b, n)\}.$$

Quando $b \in \overline{a}$, temos que $\overline{a} = \overline{b}$, pois a e b deixam o mesmo resto quando divididos por n. Portanto, dois conjuntos pertencentes a $\{\overline{a} : a \in \mathbb{Z}\}$ são iguais ou disjuntos. Isto é, $\{\overline{a} : a \in \mathbb{Z}\}$ é uma partição de $\mathbb{Z}$ que será denotada por $\mathbb{Z}_n$. Se r for o resto da divisão de a por n, então $\overline{a} = \overline{r}$. Conseqüentemente,

$$\mathbb{Z}_n = \{\overline{0}, \overline{1}, \overline{2}, \ldots, \overline{n-1}\}.$$

Para $a \neq b \in [n]$, temos que $\overline{a} \neq \overline{b}$. Logo, $\mathbb{Z}_n$ possui exatamente n elementos. Mais ainda, a função de $[n]$ em $\mathbb{Z}_n$ que leva a em $\overline{a}$ é uma bijeção entre estes conjuntos. Isto é, os elementos de $[n]$ e $\mathbb{Z}_n$ possuem uma identificação natural.

A adição de dois elementos $\overline{a}, \overline{b} \in \mathbb{Z}_n$, denotada por $\overline{a} + \overline{b}$ e resultando em $\overline{a+b}$, é bem definida. Isto é, se $\overline{a} = \overline{a'}$ e $\overline{b} = \overline{b'}$, temos $\overline{a+b} = \overline{a'+b'}$, pois

$$(a+b) - (a'+b') = (a-a') + (b-b')$$

é divisível por n — uma vez que $a - a'$ e $b - b'$ o são. A adição em $\mathbb{Z}_n$ possui as propriedades usuais:

Proposição 3.1.1. *A operação de adição em $\mathbb{Z}_n$ é associativa, comutativa, possui elemento neutro e tem inverso aditivo.*

Demonstração. Observe que $\overline{0}$ é o elemento neutro de $\mathbb{Z}_n$, pois $\overline{a}+\overline{0} = \overline{a+0} = \overline{a}$, para qualquer inteiro a. Como $\overline{a} + \overline{-a} = \overline{a+(-a)} = \overline{0}$, temos que $\overline{-a}$ é o inverso aditivo de $\overline{a}$. Por definição, para inteiros a, b e c, temos:

$$\overline{a}+(\overline{b}+\overline{c})=\overline{a}+\overline{b+c}=\overline{a+(b+c)}=\overline{(a+b)+c}=\overline{a+b}+\overline{c}=(\overline{a}+\overline{b})+\overline{c},$$

provando a associatividade. (A terceira igualdade segue da associatividade para a adição nos inteiros.) A comutatividade é mostrada de maneira similar. □

A operação de multiplicação em $\mathbb{Z}_n$ é definida de forma análoga, isto é, quando $\overline{a}, \overline{b} \in \mathbb{Z}_n$, o produto de $\overline{a}$ com $\overline{b}$, denotado por $\overline{a}\overline{b}$, resulta em $\overline{ab}$. Esta operação também está bem definida, ou seja, quando $\overline{a} = \overline{a'}$ e $\overline{b} = \overline{b'}$,

$$ab - a'b' = ab - ab' + ab' - a'b' = a(b-b') + b'(a-a')$$

é divisível por n porque $b-b'$ e $a-a'$ o são. Logo, $\overline{ab} = \overline{a'b'}$. Também de maneira análoga à adição, temos que:

Proposição 3.1.2. *A operação de multiplicação em $\mathbb{Z}_n$ é associativa, comutativa e possui elemento neutro.*

Note que, quando a e b pertencem ao conjunto $[n]$,

$$\overline{a} + \overline{b} = \overline{a \oplus b} \quad \text{e} \quad \overline{a}\overline{b} = \overline{a \otimes b}.$$

Isto é, as operações definidas em $\mathbb{Z}_n$ coincidem com as operações definidas para $[n]$ (quando fazemos a identificação natural entre estes conjuntos).

As proposições 3.1.1 e 3.1.2 estabelecem propriedades naturais para as operações de adição e multiplicação em $\mathbb{Z}_n$. Contudo, o inesperado pode ocorrer quando operamos neste conjunto. Por exemplo, quando $n = 14$, temos que

$$\overline{2} \cdot \overline{7} = \overline{14} = \overline{0}.$$

Isto é, o produto de dois elementos diferentes de $\overline{0}$ pode ser igual a $\overline{0}$. (Lembre-se que isto também ocorre com a multiplicação de matrizes.)

Dizemos que $\overline{a}$ é *divisor de zero* em $\mathbb{Z}_n$ quando $\overline{a} \neq \overline{0}$ e existe $\overline{b}$ em $\mathbb{Z}_n$ tal que $\overline{b} \neq \overline{0}$ e $\overline{a}\overline{b} = \overline{0}$.

Para descrevermos os divisores de zero em $\mathbb{Z}_n$, precisamos de uma definição. Sejam a e b inteiros, com a ou b diferente de 0. O *maior divisor comum* de a e b, denotado por (a, b), é o maior número natural que divide simultaneamente a e b. Quando $(a, b) = 1$, dizemos que a e b são *relativamente primos* (ou *primos entre si*).

Se a é um inteiro satisfazendo $d = (a, n)$, então, por definição, existem inteiros a' e n' tais que $a = da'$ e $n = dn'$. Portanto,

$$\overline{a}\overline{n'} = \overline{an'} = \overline{a'dn'} = \overline{a'n} = \overline{0}.$$

Conseqüentemente, $\overline{a}$ é divisor de zero quando $\overline{a} \neq \overline{0}$ e $d > 1$.

Dizemos que um elemento $\overline{a}$ de $\mathbb{Z}_n$ é *inversível* quando existe $\overline{b}$ em $\mathbb{Z}_n$ tal que $\overline{a}\overline{b} = \overline{1}$. (Isto é, $\overline{a}$ possui inverso multiplicativo e pode-se "dividir por $\overline{a}$" em $\mathbb{Z}_n$.) Um elemento $\overline{a}$ não pode ser simultaneamente inversível e divisor de zero em $\mathbb{Z}_n$. De fato, caso $\overline{a}$ seja inversível em $\mathbb{Z}_n$, existe elemento $\overline{b}$ de $\mathbb{Z}_n$ tal que $\overline{a}\overline{b} = \overline{1}$. Portanto, quando $\overline{ac} = \overline{0}$ temos:

$$\overline{0} = \overline{b}\overline{0} = \overline{b}(\overline{ac}) = (\overline{b}\overline{a})\overline{c} = \overline{1}\overline{c} = \overline{c}$$

e $\overline{a}$ não é divisor de zero em $\mathbb{Z}_n$.

Na próxima seção, mostraremos que todo elemento de $\mathbb{Z}_n$ que é diferente de $\overline{0}$ ou é um divisor de zero ou é inversível.

Caso $\overline{a}$ possua um inverso $\overline{b}$ em $\mathbb{Z}_n$, este inverso é único. Suponha que $\overline{b'}$ seja também um inverso de $\overline{a}$ em $\mathbb{Z}_n$. Por definição, $\overline{a}\overline{b} = \overline{1}$. Multiplicando ambos os lados desta identidade por $\overline{b'}$, temos $(\overline{a}\overline{b})\overline{b'} = \overline{b'}$. Utilizando associatividade e comutatividade da multiplicação em $\mathbb{Z}_n$, reescrevemos esta identidade como $\overline{b}(\overline{a}\overline{b'}) = \overline{b'}$. Conseqüentemente $\overline{b} = \overline{b'}$ e daí $\overline{b}$ é único.

3.2 Maior divisor comum

Sejam a e b inteiros tais que $a \neq 0$ ou $b \neq 0$. Relembraremos o algoritmo de Euclides para encontrar o maior divisor comum de a e b, que foi denotado por (a, b). Como

$$(a, b) = (b, a) \quad \text{e} \quad (a, b) = (|a|, |b|),$$

não perdemos generalidade ao assumirmos que $a \geq b \geq 0$. Se $b = 0$, então $(a, b) = a$. Portanto, vamos supor que $b \neq 0$. Iremos construir recursivamente duas seqüências, uma de restos e outra de quocientes, a saber: $r_0, r_1, \ldots, r_k, r_{k+1}$ e $q_1, \ldots, q_k$. Faça $r_0 = a, r_1 = b$ e, enquanto $r_i \neq 0$, q_i e r_{i+1} são respectivamente o quociente e o resto da divisão de r_{i-1} por r_i. Isto é,

$$r_{i-1} = q_i r_i + r_{i+1}, \text{ com } 0 \leq r_{i+1} < r_i. \tag{3.1}$$

Note que $r_{k+1} = 0$. Vamos mostrar que $r_k = (a, b)$. De (3.1), quando um inteiro divide dois elementos consecutivos na seqüência de restos, divide também o elemento anterior e o posterior a estes. Portanto, todo inteiro que divide dois elementos consecutivos da seqüência de restos divide todos os elementos desta seqüência. Como (a, b) divide dois elementos consecutivos na seqüência de restos, r_0 e r_1, então (a, b) divide r_k. De (3.1), para $i = k$, concluímos que r_k divide r_{k-1}. Conseqüentemente, r_k divide dois elementos consecutivos da seqüência de restos: r_{k-1} e r_k. Portanto, r_k divide todos os elementos da seqüência de restos, em particular a e b. Por definição do maior divisor comum, $r_k \leq (a, b)$. Logo, $r_k = (a, b)$, pois (a, b) divide r_k.

Queremos encontrar um majorante para k, isto é, um limite superior para o número de divisões realizadas pelo algoritmo de Euclides para o cálculo do maior divisor comum. De (3.1), temos que

$$r_0 \geq r_1 > r_2 > r_3 > \cdots > r_k > 0. \tag{3.2}$$

De (3.1) e (3.2), pois, segue-se que

$$q_i \geq 1 \quad \text{para todo} \quad i \in \{1, 2, 3, \ldots, k\}. \tag{3.3}$$

Substituindo (3.3) em (3.1), obtemos que, quando $i \in \{1, 2, 3, \ldots, k\}$,

$$r_{i-1} = q_i r_i + r_{i+1} \geq r_i + r_{i+1} > 2r_{i+1},$$

onde a última desigualdade segue de (3.2). Portanto, a seqüência

$$r_0, r_2, r_4, r_6, \ldots, r_{2l}$$

formada por todos os restos não-nulos tendo como índice um inteiro par satisfaz a seguinte propriedade: ao percorrermos a seqüência da

direita para a esquerda, um elemento será sempre maior que o dobro de seu antecessor. Logo

$$a = r_0 > 2r_2 > 2^2 r_4 > 2^3 r_6 > \cdots > 2^{l-1} r_{2(l-1)} > 2^l r_{2l}.$$

Em particular,

$$\log a > l + \log r_{2l} \geq l.$$

(Ao longo deste capítulo, utilizamos $\log a$ para denotar o logaritmo de a na base 2.) Como $2l \in \{k-1, k\}$, obtemos o seguinte limite superior para o número k de divisões realizadas pelo algoritmo de Euclides para o cálculo do maior divisor comum:

$$k \leq 1 + 2l \leq 1 + 2\log a.$$

Se, em (3.1), a divisão fosse realizada de forma que o resto r_{i+1} satisfaça

$$-\frac{|r_i|}{2} < r_{i+1} \leq \frac{|r_i|}{2},$$

necessitaríamos no máximo $\log a$ divisões para encontrar o maior divisor comum, tornando o algoritmo de Euclides mais eficiente. (Neste caso, os restos poderiam assumir também valores negativos.)

Dizemos que um inteiro x é *combinação* dos inteiros y e z quando existem inteiros μ e ν tais que $x = \mu y + \nu z$. Observe que x é combinação de y e z se e somente se $|x|$ é combinação de $|y|$ e $|z|$.

Teorema 3.2.1. *Se a e b são inteiros tais que $a \neq 0$ ou $b \neq 0$, então (a, b) é combinação de a e b.*

Demonstração. Sem perda de generalidade, podemos assumir que $a \geq b \geq 0$. Estabeleceremos um resultado mais forte: (a, b) é combinação de quaisquer dois elementos consecutivos da seqüência de restos. Utilizando recursão, este resultado segue dos seguintes fatos:

(i) (a, b) é combinação dos dois últimos elementos da seqüência de restos; e

(ii) para quaisquer três elementos consecutivos da seqüência dos restos, se (a, b) é combinação dos dois últimos, então também é combinação dos dois primeiros.

Comprovamos **(i)** facilmente, pois $(a, b) = r_k = \mu r_k + \nu r_{k+1}$, onde (μ, ν) é igual a $(1, 0)$. Para mostrar **(ii)**, suponha que r_{i-1}, r_i, r_{i+1} sejam os três elementos consecutivos da seqüência de restos, para algum natural i. Por hipótese, (a, b) é combinação de r_i e r_{i+1}, isto é, existem inteiros μ e ν tais que $(a, b) = \mu r_i + \nu r_{i+1}$. De (3.1), temos que $r_{i+1} = r_{i-1} - q_i r_i$. Portanto,

$$(a, b) = \mu r_i + \nu r_{i+1} = \mu r_i + \nu(r_{i-1} - q_i r_i) = \nu r_{i-1} + (\mu - \nu q_i) r_i$$

e (a, b) é combinação de r_{i-1} e r_i. Conseqüentemente, **(ii)** segue. □

Observe que na demonstração do resultado anterior está implícito um algoritmo para encontrar tal combinação linear. Este algoritmo, conhecido como *algoritmo euclidiano estendido*, possui k etapas e, em cada uma delas, é realizada uma subtração e uma multiplicação.

Seja n um número natural. Se $\overline{a}$ é um elemento de $\mathbb{Z}_n$ e $\overline{a} \neq \overline{0}$, então:

(i) $(a, n) \neq 1$ e $\overline{a}$ é um divisor de zero de $\mathbb{Z}_n$; ou

(ii) $(a, n) = 1$ e $\overline{a}$ é inversível em $\mathbb{Z}_n$.

Já estabelecemos **(i)** na seção anterior. Vamos mostrar **(ii)**. Pelo teorema 3.2.1, existem inteiros μ e ν tais que $1 = (a, n) = \mu a + \nu n$. Portanto,

$$\overline{1} = \overline{\mu a + \nu n} = \overline{\mu}\,\overline{a} + \overline{\nu}\,\overline{n} = \overline{\mu}\,\overline{a} + \overline{\nu}\,\overline{0} = \overline{\mu}\,\overline{a}$$

e daí $\overline{a}$ é inversível.

O conjunto de todos os elementos inversíveis de $\mathbb{Z}_n$ será denotado por $\mathbb{Z}_n^*$. Este conjunto é fechado com relação à operação de multiplicação e terá papel central na compreensão dos algoritmos para decidir a primalidade de um número. A cardinalidade de $\mathbb{Z}_n^*$ será denotada por $\phi(n)$. (Esta função é conhecida como a função ϕ de Euler.) Note que

$$\phi(n) = |\{a \in [n] : (a, n) = 1\}|.$$

Em particular, quando p é um número primo e m um inteiro positivo, $\phi(p^m) = p^m - p^{m-1}$. Mais ainda, quando $m = 1$, $\mathbb{Z}_p^* = \mathbb{Z}_p - \{\overline{0}\}$.

3.3 Teorema Fundamental da Aritmética

Nesta seção, recapitulamos o clássico Teorema Fundamental da Aritmética, um dos primeiros fatos matemáticos apresentados aos estudantes no ensino fundamental.

Um inteiro maior que 1 é dito *primo* quando não é o produto de dois inteiros positivos menores. O resultado seguinte é o núcleo da demonstração do Teorema Fundamental da Aritmética.

Lema 3.3.1. *Se um primo divide um produto de inteiros, então divide um de seus fatores.*

Demonstração. Seja p um número primo. Suponhamos que p divida o produto ab de dois números $a, b \in \mathbb{Z}$. Se p divide a, então o resultado segue. Assumamos, portanto, que p não divide a. Observe que $(a, p) = 1$, pois (a, p) é um divisor próprio de p. Pelo teorema 3.2.1, existem inteiros μ e ν tais que

$$1 = \mu a + \nu p.$$

Multiplicando-se esta igualdade por b, obtemos

$$b = \mu ab + \nu pb. \tag{3.4}$$

Como p divide ab e pb, temos que p divide o lado direito de (3.4). Portanto, p divide b e o lema vale para o produto de dois inteiros.

Suponhamos, agora, que p divide o produto de n inteiros, digamos $a_1, a_2, \ldots, a_n$. Pelo que acabamos de estabelecer,

(i) p divide a_1; ou

(ii) p divide o produto dos outros $n-1$ inteiros, que são $a_2, \ldots, a_n$.

Podemos repetir este processo e, ao final, encontramos um a_i que é divisível por p. □

Estamos prontos para demonstrar o Teorema Fundamental da Aritmética. Vamos assumir que o produto dos elementos pertencentes ao conjunto vazio é 1 e que o produto dos elementos pertencentes a um conjunto unitário é o seu elemento.

Teorema 3.3.2. *Todo inteiro positivo decompõe-se de maneira única, a menos da ordem dos fatores, como o produto de números primos.*

Demonstração. Seja n um inteiro positivo. Como o resultado vale quando n é 1 ou primo, necessitamos demonstrá-lo apenas quando n é composto. Primeiro, mostraremos que n decompõe-se como o produto de números primos. Escolha uma seqüência $a_1, a_2, \ldots, a_k$ de inteiros maiores que 1, tal que

$$n = a_1 a_2 \cdots a_k$$

e o tamanho da seqüência seja o maior possível. (Note que as seqüências $a_1, a_2, \ldots, a_k$ cujo produto é igual a n têm comprimento limitado por $\log n$ porque $n = a_1 a_2 \ldots a_k \geq 2^k$. Portanto, faz sentido escolhermos uma de tamanho máximo.) Observe que, para todo i, a_i é primo; do contrário a_i seria o produto de dois inteiros positivos menores e, ao substituirmos, na seqüência, a_i por aqueles dois inteiros, obteríamos uma seqüência de tamanho maior, o que é uma contradição.

Agora, estabeleceremos a unicidade da decomposição. Sejam $a_1, a_2, \ldots, a_k$ e $b_1, b_2, \ldots, b_l$ seqüências de números primos tais que

$$n = a_1 a_2 \cdots a_k = b_1 b_2 \cdots b_l.$$

Sem perda de generalidade, podemos supor que $k \leq l$. Pelo lema 3.3.1, a_1 divide b_i, para algum i. Como b_i é primo, temos que $a_1 = b_i$. Podemos reordenar os elementos na seqüência $b_1, b_2, \ldots, b_l$ e supor que $i = 1$. Logo,

$$\frac{n}{a_1} = a_2 \cdots a_k = b_2 \cdots b_l.$$

Repetindo este processo, podemos reordenar os elementos da segunda seqüência, de forma que $a_2 = b_2, \ldots, a_k = b_k$. Portanto,

$$\frac{n}{a_1 a_2 \cdots a_k} = 1 = b_{k+1} \cdots b_l.$$

Como, necessariamente, $k = l$, essas decomposições de n são idênticas, a menos da ordem dos fatores. □

3.4 O Pequeno Teorema de Fermat

Seja n um inteiro positivo. Para um elemento $\overline{a}$ de $\mathbb{Z}_n^*$, considere todas as suas potências:

$$\overline{1} = \overline{a}^0, \overline{a}^1, \overline{a}^2, \overline{a}^3, \ldots, \overline{a}^k, \ldots.$$

Como todas estas potências pertencem ao conjunto finito $\mathbb{Z}_n^*$, existem inteiros i e j tais que $i < j$ e $\overline{a}^i = \overline{a}^j$. Escolha i e j de forma que j seja o menor possível. Se $\overline{b}$ é o inverso multiplicativo de $\overline{a}$, então

$$\overline{1} = \overline{1}^i = (\overline{a}\overline{b})^i = \overline{a}^i\overline{b}^i = \overline{a}^j\overline{b}^i = \overline{a}^{j-i}(\overline{a}\overline{b})^i = \overline{a}^{j-i}.$$

Pela escolha de i e j, temos que $i = 0$. Dizemos que j é a *ordem* de $\overline{a}$ em $\mathbb{Z}_n^*$. Note que

$$\overline{a}^1, \overline{a}^2, \overline{a}^3, \ldots, \overline{a}^j = \overline{1}$$

são todas as potências de $\overline{a}$ em $\mathbb{Z}_n$. A seguir apresentamos o Teorema de Lagrange.

Teorema 3.4.1. *Seja $\overline{a}$ um elemento de $\mathbb{Z}_n^*$ de ordem j. Se $\overline{a}^k = \overline{1}$, então j divide k.*

Demonstração. Pelo algoritmo da divisão, existem inteiros q e r tais que $k = qj + r$ e $0 \leq r < j$. Conseqüentemente,

$$\overline{1} = \overline{a}^k = \overline{a}^{qj+r} = (\overline{a}^j)^q\overline{a}^r = \overline{1}^q\overline{a}^r = \overline{a}^r.$$

Como j é o menor inteiro positivo tal que $\overline{a}^j = \overline{1}$, tem-se que $r = 0$. Logo j divide k. □

Lema 3.4.2. *Seja $\overline{a}$ um elemento de $\mathbb{Z}_n^*$ de ordem j. Se $i \in [j]$, então a ordem de $\overline{a}^i$ é igual a $\frac{j}{(i,j)}$.*

Demonstração. Existem inteiros i' e j' tais que $i = (i,j)i'$ e $j = (i,j)j'$. Note que

$$(\overline{a}^i)^{j'} = \overline{a}^{ij'} = \overline{a}^{(i,j)i'j'} = \overline{a}^{i'j} = (\overline{a}^j)^{i'} = \overline{1}^{i'} = \overline{1}.$$

Se k é a ordem de $\overline{a}^i$, então, pelo teorema 3.4.1, k divide $j' = \frac{j}{(i,j)}$.

Por definição, $\overline{1} = (\overline{a}^i)^k = \overline{a}^{ik}$. Pelo teorema 3.4.1, j divide ik. Portanto, j' divide $i'k$. Como $(i', j') = 1$, tem-se que j' divide k. Mas, pelo parágrafo anterior, k divide j' e daí $k = j'$, como queríamos mostrar. □

O próximo resultado foi descoberto por Euler e descreve as possíveis ordens dos elementos de $\mathbb{Z}_n^*$.

Teorema 3.4.3. *A ordem de um elemento $\overline{a}$ de $\mathbb{Z}_n^*$ divide $\phi(n)$.*

Demonstração. A função $f : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ dada por $f(X) = \overline{a}X$ é injetiva porque $\overline{a}$ é inversível. Como $\mathbb{Z}_n^*$ é finito, f também é sobrejetiva. Isto é, f induz uma permutação dos elementos de $\mathbb{Z}_n^*$. Portanto,

$$\prod_{X\in\mathbb{Z}_n^*} X = \prod_{X\in\mathbb{Z}_n^*} f(X) = \prod_{X\in\mathbb{Z}_n^*} (\overline{a}X) = \prod_{X\in\mathbb{Z}_n^*} \overline{a} \prod_{X\in\mathbb{Z}_n^*} X = \overline{a}^{\phi(n)} \prod_{X\in\mathbb{Z}_n^*} X.$$

Logo, $\overline{a}^{\phi(n)} = 1$. Pelo teorema 3.4.1, a ordem de $\overline{a}$ divide $\phi(n)$. □

A instância particular do teorema anterior em que n é primo foi obtida por Fermat. Neste caso $\mathbb{Z}_n^* = \{\overline{a} : a \in [n]^*\}$ e $\phi(n) = n - 1$, onde, para um inteiro positivo m, $[m]^* = \{a \in [m] : a \neq 0\} = \{1, \ldots, m-1\}$. Conseqüentemente,

Teorema 3.4.4. *Se $a \in [n]^*$ e n é primo, então $\overline{a}^{n-1} = \overline{1}$.*

Este resultado inspira um possível algoritmo para decidir a primalidade de n. Repita algumas vezes o seguinte procedimento: escolha aleatoriamente $a \in [n]^*$ e calcule o resto da divisão de a^{n-1} por n. Se algum destes restos não for igual a 1, então, pelo resultado anterior, n é composto. Senão, será que podemos afirmar que n é primo a menos de uma probabilidade muito baixa? A resposta infelizmente é não! Existem números compostos n que falham neste teste para muito poucos $a \in [n]^*$. Neste caso, a probabilidade do resto ser 1 é muito alta, mesmo n sendo composto. Nosso objetivo será mostrar que uma pequena variante deste algoritmo funciona como desejado. Para tanto, necessitamos estudar raízes de polinômios com coeficientes em $\mathbb{Z}_n$, quando n é primo. (Surpreendentemente, o próximo resultado não é verdadeiro quando n é composto.)

Lema 3.4.5. *Se n é primo, então o número de raízes de um polinômio $p(X)$ com coeficientes em $\mathbb{Z}_n$ é no máximo igual ao grau de $p(X)$.*

Demostração. Escolha a seqüência de elementos $\overline{a_1}, \ldots, \overline{a_m}$ de $\mathbb{Z}_n$ com o maior comprimento possível tal que

$$p(X) = (X - \overline{a_1}) \cdots (X - \overline{a_m}) q(X)$$

para algum polinômio $q(X)$ com coeficientes em $\mathbb{Z}_n$. Note que $\overline{a_1}, \ldots, \overline{a_m}$ são raízes de $p(X)$. Se estas são todas as raízes de $p(X)$ então o resultado segue, pois o grau de $p(X)$ é igual a m mais o grau de $q(X)$. Podemos supor que $p(X)$ possui raiz $\overline{a}$ tal que $\overline{a} \neq \overline{a_i}$ para todo i. Logo,

$$\overline{0} = (\overline{a} - \overline{a_1}) \cdots (\overline{a} - \overline{a_m}) q(\overline{a}).$$

Mas $\overline{a} - \overline{a_i} \neq \overline{0}$, para todo i, e daí $q(\overline{a}) = \overline{0}$ porque $\mathbb{Z}_n$ não tem divisores de zero quando n é primo. Dividindo $q(X)$ por $X - \overline{a}$ encontramos um polinômio $\widehat{q}(X)$ com coeficientes em $\mathbb{Z}_n$ tal que $q(X) = \widehat{q}(X)(X - \overline{a}) + q(\overline{a}) = \widehat{q}(X)(X - \overline{a})$. Portanto,

$$p(X) = (X - \overline{a_1}) \cdots (X - \overline{a_m})(X - \overline{a}) \widehat{q}(X)$$

e $\overline{a_1}, \ldots, \overline{a_m}, \overline{a}$ contraria a escolha de $\overline{a_1}, \ldots, \overline{a_m}$. □

Descreveremos um algoritmo para decidir, com probabilidade de erro pequena, a primalidade de um número natural n. Utilizaremos, para isto, a seguinte variante do teorema 3.4.4.

Teorema 3.4.6. *Suponha que $n - 1 = 2^r m$, onde r é um inteiro não-negativo e m é um inteiro ímpar. Se $a \in [n]^*$ e n é primo, então:*

(i) $\overline{a}^m = \overline{1}$*; ou*

(ii) $\overline{a}^{2^i m} = \overline{-1}$*, para algum inteiro i tal que $0 \leq i \leq r - 1$.*

Demonstração. Suponha que **(i)** não ocorre. Escolha o maior inteiro i tal que $0 \leq i \leq r$ e $\overline{a}^{2^i m} \neq \overline{1}$. Pelo teorema 3.4.4, $i < r$. Portanto, $\overline{a}^{2^{i+1} m} = \overline{1}$. Logo $\overline{a}^{2^i m}$ é uma raiz do polinômio $p(X) = X^2 - \overline{1}$. Pelo lema 3.4.5, $p(X)$ possui duas raízes que necessariamente são $\overline{1}$ e $\overline{-1}$. Pela escolha de i, $\overline{a}^{2^i m} = \overline{-1}$. Temos **(ii)**. □

Na seção 3.7, mostraremos que a probabilidade de $a \in [n]^*$ satisfazer **(i)** ou **(ii)** do teorema anterior, no caso em que n é ímpar e composto, é inferior a $\frac{1}{4}$.

3.5 Teorema Chinês do Resto

O próximo resultado é conhecido como o Teorema Chinês do Resto e, surpreendentemente, era utilizado para determinar o tamanho de tropas militares.

Teorema 3.5.1. *Sejam $m_1, m_2, \ldots, m_k$ inteiros tais que $(m_i, m_j) = 1$ para qualquer 2-subconjunto $\{i, j\}$ de $\{1, 2, \ldots, k\}$. Se $0 \leq r_i < m_i$, para todo $i \in \{1, 2 \ldots, k\}$, então existe um único $a \in [m_1 m_2 \ldots m_k]$ tal que $r_i = \text{res}(a, m_i)$, para todo $i \in \{1, 2 \ldots, k\}$.*

Ressaltamos que, embutido na demonstração deste teorema, encontra-se um algoritmo para efetivamente encontrar o valor de a.

Demonstração. Inicialmente trataremos o caso em que $k = 2$. Observe que

$$\{a \in [m_1 m_2] : r_1 = \text{res}(a, m_1)\} = \{qm_1 + r_1 : q \in [m_2]\}.$$

Desejamos encontrar todos os elementos a pertencentes a este conjunto tais que

$$r_2 = \text{res}(a, m_2).$$

Isto é, desejamos encontrar todos os $q \in [m_2]$ tais que

$$\overline{r_2} = \overline{qm_1 + r_1} \text{ em } \mathbb{Z}_{m_2}.$$

Esta identidade pode ser reescrita como

$$\overline{qm_1} = \overline{r_2 - r_1} \text{ em } \mathbb{Z}_{m_2}.$$

Como $(m_1, m_2) = 1$, temos que m_1 possui um inverso multiplicativo em $\mathbb{Z}_{m_2}$, digamos $\overline{m}$. (Este inverso pode ser calculado utilizando-se o algoritmo que está implícito na demonstração do teorema 3.2.1.) Logo,

$$\overline{q} = \overline{m(r_2 - r_1)} \text{ em } \mathbb{Z}_{m_2}.$$

Conseqüentemente, $\overline{q}$ existe e é único. O mesmo ocorre com q porque $q \in [m_2]$. Logo o resultado vale quando $k = 2$.

Vamos construir recursivamente uma seqüência $a_1, a_2, \ldots, a_k$ tal que, para todo $l \in \{1, 2, \ldots, k\}$,

$$a_l \in \left[\prod_{i=1}^{l} m_i\right] \quad \text{e} \quad r_i = \text{res}(a_l, m_i), i = 1, 2, \ldots, l. \tag{3.5}$$

Mais ainda, mostraremos que esta seqüência é única. Suponha que a_j já foi determinado. (Note que $a_1 = r_1$.) Descreveremos, agora, como obter a_{j+1} a partir de a_j. Como m_i é relativamente primo com m_{j+1}, para todo $i \in \{1, \ldots, j\}$, então, pelo Teorema Fundamental da Aritmética, $\prod_{i=1}^{j} m_i = m_1 \cdots m_j$ é relativamente primo com m_{j+1}. Aplicando o resultado do parágrafo anterior, existe um único elemento a_{j+1} de $[m_1 \cdots m_j m_{j+1}]$ tal que

$$a_j = \text{res}(a_{j+1}, m_1 \cdots m_j) \quad \text{e} \quad r_{j+1} = \text{res}(a_{j+1}, m_{j+1}). \tag{3.6}$$

Por (3.5) para $l = j$ e (3.6), concluímos que (3.5) vale também para $l = j + 1$. Isto é, a_{j+1} satisfaz as propriedades desejadas. Observe que a_{j+1} é o único elemento de $[m_1 \cdots m_j m_{j+1}]$ satisfazendo (3.6) e, portanto, será único, já que a_j o é. □

Usaremos uma outra abordagem, que não é algorítmica, para estabelecer o Teorema Chinês do Resto. Sejam $m_1, m_2 \ldots, m_k$ inteiros positivos tais que $(m_i, m_j) = 1$ para qualquer 2-subconjunto $\{i, j\}$ de $\{1, 2, \ldots, k\}$. Denote o produto destes inteiros por n, isto é, $n = m_1 \cdots m_k$. Considere a função $\Psi : \mathbb{Z}_n \to \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_k}$ dada por $\Psi(\overline{a}) = (\overline{a}, \overline{a}, \ldots, \overline{a})$, para um inteiro a. (Apesar de $\overline{a}$ representar conjuntos diferentes nesta identidade, nossa notação não gerará confusão, já que está implícito qual conjunto estamos considerando em cada ocorrência de $\overline{a}$. Na primeira, $\overline{a}$ denota o conjunto de inteiros que têm o mesmo resto que a quando divididos por n; na segunda quando divididos por m_1; na terceira quando divididos por m_2; e na última quando divididos por m_k.)

Mostraremos simultaneamente que Ψ está bem definida e é injetiva. Estas propriedades seguem das seguintes equivalências para os inteiros a e a':

(i) $\overline{a} = \overline{a'}$ em $\mathbb{Z}_n$;

(ii) n divide $a - a'$;

(iii) para todo $i \in \{1, 2, \ldots, k\}$, m_i divide $a - a'$;

(iv) para todo $i \in \{1, 2, \ldots, k\}$, $\overline{a} = \overline{a'}$ em $\mathbb{Z}_{m_i}$;

(v) $\Psi(\overline{a}) = \Psi(\overline{a'})$.

(Para estabelecer que **(ii)** e **(iii)** são equivalentes é necessário utilizar o Teorema Fundamental da Aritmética.) Como

$$|\mathbb{Z}_n| = |\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_k}|$$

e Ψ é injetiva, temos que:

Proposição 3.5.2. *A função Ψ é bijetiva.*

Como conseqüência deste resultado obtemos o Teorema Chinês do Resto.

Utilizaremos a função Ψ para obter uma propriedade fundamental da função ϕ de Euler: multiplicatividade. Isto é,

$$\phi(n) = \phi(m_1)\phi(m_2)\cdots\phi(m_k). \tag{3.7}$$

Como $\phi(m) = |\mathbb{Z}_m^*|$, para um inteiro m positivo, a equação (3.7) segue de:

$$|\mathbb{Z}_n^*| = |\mathbb{Z}_{m_1}^* \times \mathbb{Z}_{m_2}^* \times \cdots \times \mathbb{Z}_{m_k}^*|. \tag{3.8}$$

Observe que (3.8) é uma conseqüência imediata da proposição 3.5.2 juntamente com o próximo resultado:

Lema 3.5.3. $\overline{a} \in \mathbb{Z}_n^*$ *se somente se* $\Psi(\overline{a}) \in \mathbb{Z}_{m_1}^* \times \mathbb{Z}_{m_2}^* \times \cdots \times \mathbb{Z}_{m_k}^*$.

Demonstração. Observe que as seguintes afirmações são equivalentes:

(i) $\overline{a} \in \mathbb{Z}_n^*$;

(ii) $(a, n) = 1$;

(iii) para todo $i \in \{1, 2, \ldots, k\}$, $(a, m_i) = 1$;

(iv) para todo $i \in \{1, 2, \ldots, k\}$, $\overline{a} \in \mathbb{Z}_{m_i}^*$.

A equivalência entre **(ii)** e **(iii)** é estabelecida pelo Teorema Fundamental da Aritmética. □

3.6 Geradores para $\mathbb{Z}_n^*$

Seja n um número natural. Um elemento $\overline{a}$ de $\mathbb{Z}_n^*$ é dito um *gerador* para $\mathbb{Z}_n^*$ quando todo elemento de $\mathbb{Z}_n^*$ é uma potência de $\overline{a}$. Neste caso,

$$\mathbb{Z}_n^* = \{\overline{a}^j : j \in \mathbb{N}\}.$$

Se a ordem de $\overline{a}$ é d, então,

$$\mathbb{Z}_n^* = \{\overline{1}, \overline{a}, \overline{a}^2, \ldots, \overline{a}^{d-1}\}.$$

Em particular, $d = \phi(n)$. Nesta seção, mostraremos que, quando n é primo ou o quadrado de um primo, então $\mathbb{Z}_n^*$ possui um gerador. Primeiro contaremos o número de elementos que possuem determinada ordem em $\mathbb{Z}_n^*$, quando n é primo.

Lema 3.6.1. *Suponha que n é primo. Se d é um inteiro positivo, então $\mathbb{Z}_n^*$ possui 0 ou $\phi(d)$ elementos de ordem d.*

Demonstração. Considere o polinômio com coeficientes em $\mathbb{Z}_n$:

$$p(X) = X^d - \overline{1}.$$

Observe que todo elemento de $\mathbb{Z}_n^*$ que tem ordem d é raiz de $p(X)$. Portanto, para determinarmos o número de elementos de $\mathbb{Z}_n^*$ que possuem ordem d, vamos listar todas as raízes de $p(X)$ e decidir quais são aquelas que possuem ordem d.

Se $\mathbb{Z}_n^*$ não tem elemento de ordem d, então o resultado segue. Assuma que $\mathbb{Z}_n^*$ tenha algum elemento de ordem d. Seja $\overline{a}$ um destes elementos. Na seção 3, mostramos que

$$\overline{a}, \overline{a}^2, \ldots, \overline{a}^{d-1}, \overline{a}^d = \overline{1}$$

são dois a dois distintos. Vamos mostrar que cada um destes elementos é raiz de $p(X)$. De fato,

$$p(\overline{a}^j) = (\overline{a}^j)^d - \overline{1} = (\overline{a}^d)^j - \overline{1} = (\overline{1})^j - \overline{1} = \overline{1} - \overline{1} = \overline{0}.$$

Pelo lema 3.4.5, $p(X)$ possui no máximo d raízes em $\mathbb{Z}_n$. Portanto,

$$\overline{1}, \overline{a}, \overline{a}^2, \ldots, \overline{a}^{d-1}$$

são todas as raízes de $p(X)$. Pelo lema 3.4.2, para $i \in [d]$, $\overline{a}^i$ tem ordem d se e somente se $(i,d) = 1$. Conseqüentemente, $p(X)$ possui exatamente $\phi(d)$ raízes com ordem d. Isto é, $\mathbb{Z}_n^*$ possui $\phi(d)$ elementos com ordem d. □

Utilizaremos a notação $a|b$ com o seguinte significado: a e b são inteiros positivos e a divide b.

Lema 3.6.2. *Suponha que n é primo. Se d é um inteiro positivo que divide $n-1$, então $\mathbb{Z}_n^*$ possui $\phi(d)$ elementos de ordem d.*

Demonstração. Para cada inteiro positivo d, seja Λ_d o conjunto de elementos de $\mathbb{Z}_n^*$ que possuem ordem d. Desejamos estabelecer que $|\Lambda_d| = \phi(d)$. Observe que

$$\{\Lambda_d : d \in \mathbb{Z} \quad \text{e} \quad d > 0\}$$

é uma partição de $\mathbb{Z}_n^*$. Pelo teorema 3.4.3, $\Lambda_d = \varnothing$ quando d não divide $\phi(n) = n-1$. Conseqüentemente,

$$\{\Lambda_d : d|n-1\}$$

é uma partição de $\mathbb{Z}_n^*$. Portanto,

$$n - 1 = \sum_{d|n-1} |\Lambda_d|.$$

Pelo lema 3.6.1, $|\Lambda_d| \in \{0, \phi(d)\}$. Em particular,

$$|\Lambda_d| \leq \phi(d), \text{ quando } d|n-1. \tag{3.9}$$

Portanto,

$$n - 1 = \sum_{d|n-1} |\Lambda_d| \leq \sum_{d|n-1} \phi(d) = n - 1, \tag{3.10}$$

onde a última igualdade segue de (3.11), que será estabelecida a seguir. Temos, dessa forma, a igualdade em (3.10). Portanto, a igualdade tem que ocorrer em (3.9), para todo $d|n-1$. Isto é, $|\Lambda_d| = \phi(d)$, para todo $d|n-1$. □

Na demonstração do lema anterior, necessitamos da seguinte identidade:

$$\sum_{d|m} \phi(d) = m. \tag{3.11}$$

Para $d|m$, seja $\Gamma_d = \{a \in [m] : (a, m) = d\}$. Observe que $\{\Gamma_d : d|m\}$ é uma partição de $[m]$. Conseqüentemente,

$$\sum_{d|m} |\Gamma_d| = m. \tag{3.12}$$

Pelo Teorema Fundamental da Aritmética,

$$\Gamma_d = \left\{ad : a \in \left[\frac{m}{d}\right] \text{ e } \left(a, \frac{m}{d}\right) = 1\right\}$$

e daí

$$|\Gamma_d| = \phi\left(\frac{m}{d}\right).$$

Substituindo a última identidade em (3.12), obtemos

$$\sum_{d|m} \phi\left(\frac{m}{d}\right) = m. \tag{3.13}$$

Quando d percorre todos os divisores positivos de m, então $\frac{m}{d}$ também percorre todos os divisores positivos de m. Logo,

$$\sum_{d|m} \phi\left(\frac{m}{d}\right) = \sum_{d|m} \phi(d). \tag{3.14}$$

Note que (3.11) é uma conseqüência de (3.13) e (3.14).

Uma conseqüência imediata do resultado anterior será a próxima proposição, fundamental para estabelecermos a probabilidade de erro do algoritmo para decidir primalidade.

Proposição 3.6.3. *Se n é primo, então $\mathbb{Z}_n^*$ possui um gerador.*

Demonstração. Um elemento de $\mathbb{Z}_n^*$ é gerador quando sua ordem é $\phi(n) = n-1$. Pelo lema 3.6.2, $\mathbb{Z}_n^*$ possui $\phi(n-1)$ elementos de ordem $n-1$. O resultado segue pois $\phi(n-1) \neq 0$. □

Proposição 3.6.4. *Se n é primo, então $\mathbb{Z}_{n^2}^*$ possui um gerador.*

Demonstração. Um elemento de $\mathbb{Z}_{n^2}^*$ é gerador quando sua ordem é $\phi(n^2) = n^2 - n = n(n-1)$. Pela proposição 3.6.3, existe $a \in [n]$ tal que $\overline{a}$ gera $\mathbb{Z}_n^*$. Seja d a ordem de $\overline{a}$ em $\mathbb{Z}_{n^2}^*$. Por definição, em $\mathbb{Z}_{n^2}^*$,

$$\overline{a}^d = \overline{1}.$$

Em outras palavras, n^2 divide $a^d - 1$. Conseqüentemente, n divide $a^d - 1$. Ou seja, em $\mathbb{Z}_n^*$,

$$\overline{a}^d = \overline{1}.$$

Pelo teorema 3.4.1, a ordem de $\overline{a}$ em $\mathbb{Z}_n^*$ divide d. Isto é, $n-1$ divide d. Pelo teorema 3.4.3, d divide $\phi(n^2) = n(n-1)$. Como n é primo, $d \in \{n-1, n(n-1)\}$. Se $d = n(n-1)$, então o resultado segue. Podemos assumir que $d = n-1$. Em particular,

$$\overline{a^{n-1}} = \overline{1} \tag{3.15}$$

em $\mathbb{Z}_{n^2}^*$. De maneira análoga, temos que a ordem de $\overline{n+a}$ em $\mathbb{Z}_{n^2}^*$ é igual a $n-1$ ou $n(n-1)$. Se a ordem for $n(n-1)$, então o resultado segue. Podemos assumir que a ordem é $n-1$ e daí

$$\overline{(n+a)^{n-1}} = \overline{1} \tag{3.16}$$

em $\mathbb{Z}_{n^2}^*$. Pelo binômio de Newton, existe inteiro c, que depende de a e n, tal que

$$(n+a)^{n-1} = cn^2 + (n-1)na^{n-2} + a^{n-1} = (c+a^{n-2})n^2 - na^{n-2} + a^{n-1}.$$

(Coloque em evidência n^2 nos termos do binômio cujo expoente de n é pelo menos 2.) Em $\mathbb{Z}_{n^2}^*$, esta igualdade torna-se

$$\overline{(n+a)^{n-1}} = \overline{(c+a^{n-2})n^2 - na^{n-2} + a^{n-1}} = \overline{-na^{n-2}} + \overline{a^{n-1}}.$$

Substituindo (3.15) e (3.16) na identidade acima, temos

$$\overline{1} = \overline{-na^{n-2}} + \overline{1}.$$

Portanto, em $\mathbb{Z}_{n^2}^*$, $\overline{-na^{n-2}} = \overline{0}$. Isto é, n^2 divide na^{n-2}. Como n é primo, n divide a; uma contradição já que $\overline{a}$ é um gerador para $\mathbb{Z}_n^*$. □

3.7 Pseudoprimos

Seja n um inteiro positivo ímpar. Existem inteiros positivos r e m tais que m é ímpar e $n-1 = 2^r m$. Para $a \in [n]^*$, dizemos que n é *pseudoprimo* com respeito à *base* a quando:

(i) $\overline{a}^m = \overline{1}$; ou

(ii) $\overline{a}^{2^i m} = \overline{-1}$, para algum inteiro i tal que $0 \leq i \leq r-1$.

Pelo teorema 3.4.6, quando n é primo, **(i)** ou **(ii)** é sempre verdadeira. Em particular, n é pseudoprimo[2] com respeito a todo elemento de $[n]^*$.

No restante desta seção, assumiremos que n não é primo. Mostraremos que n não é um pseudoprimo com respeito à maioria dos elementos pertencentes a $[n]^*$. Na verdade, vamos obter um limite superior para a probabilidade P_n de n ser pseudoprimo com respeito a um elemento de $[n]^*$ escolhido aleatoriamente. Observe que:

$$P_n = \frac{|\{a \in [n]^* : \ n \text{ é pseudoprimo com respeito a } a\}|}{n-1}.$$

Rabin [45] obteve o seguinte resultado:

Teorema 3.7.1. *Seja n um inteiro positivo ímpar. Se n não é primo, então a probabilidade P_n de n ser pseudoprimo com respeito a um elemento de $[n]^*$ escolhido aleatoriamente não excede $\frac{1}{4}$.*

Nesta seção, nosso objetivo é demonstrar este teorema. Consideraremos primeiro o caso em que n não é "livre de quadrados".

Proposição 3.7.2. *Se existe primo q tal que $q^2|n$, então*

$$P_n \leq \frac{1}{q+1} \leq \frac{1}{4}.$$

Demonstração. Quando n é pseudoprimo com respeito a a, temos que $\overline{a}^{n-1} = \overline{1}$ em $\mathbb{Z}_n$. Neste caso, $\overline{a} \in \mathbb{Z}_n^*$. Portanto,

$$P_n \leq \frac{|\{\overline{a} \in \mathbb{Z}_n^* : \overline{a}^{n-1} = \overline{1} \text{ em } \mathbb{Z}_n\}|}{n-1}. \tag{3.17}$$

[2] Soa estranho dizer que um número primo é pseudoprimo com respeito a um outro número, mas esta abordagem facilitará nossa discussão futura.

Podemos reescrever a equação 3.17 como:

$$P_n \leq \frac{|\{a \in [n] : (a,n) = 1 \text{ e } n|a^{n-1} - 1\}|}{n-1}. \tag{3.18}$$

Como $q^2|n$, obtemos que $\{a \in [n] : (a,n) = 1 \text{ e } n|a^{n-1} - 1\} \subseteq \{a \in [n] : (a,q^2) = 1 \text{ e } q^2|a^{n-1} - 1\}$. Conseqüentemente,

$$P_n \leq \frac{|\{a \in [n] : (a,q^2) = 1 \text{ e } q^2|a^{n-1} - 1\}|}{n-1}. \tag{3.19}$$

Como cada elemento $\overline{b}$ de $\mathbb{Z}_{q^2}$ intersecta $[n]$ em $\frac{n}{q^2}$ elementos, concluímos que

$$|\{a \in [n] : (a,q^2) = 1 \text{ e } q^2|a^{n-1} - 1\}| = \frac{n}{q^2}|\{a \in [q^2] : (a,q^2) = 1 \text{ e } q^2|a^{n-1} - 1\}|.$$

Substituindo esta igualdade na equação (3.19), obtemos:

$$P_n \leq \frac{n}{q^2(n-1)}|\{a \in [q^2] : (a,q^2) = 1 \text{ e } q^2|a^{n-1} - 1\}|.$$

Esta desigualdade pode ser escrita como

$$P_n \leq \frac{n}{q^2(n-1)}|\{\overline{a} \in \mathbb{Z}^*_{q^2} : \overline{a}^{n-1} = \overline{1} \text{ em } \mathbb{Z}_{q^2}\}|. \tag{3.20}$$

Pela proposição 3.6.4, existe inteiro a tal que $\overline{a}$ é um gerador para $\mathbb{Z}^*_{q^2}$. Portanto, em $\mathbb{Z}^*_{q^2}$, $\overline{a}$ possui ordem $\phi(q^2) = q(q-1)$. Sabemos que

$$\mathbb{Z}^*_{q^2} = \{\overline{a}^i : i \in [q(q-1)]\}.$$

Portanto, temos de determinar todos os expoentes i pertencentes a $[q(q-1)]$ tais que, em $\mathbb{Z}_{q^2}$,

$$\overline{a}^{i(n-1)} = (\overline{a}^i)^{n-1} = \overline{1}. \tag{3.21}$$

Pelo teorema 3.4.1, $q(q-1)|i(n-1)$. Como $q^2|n$, temos que $(q, n-1) = 1$. Portanto, $q|i$. Como apenas $q-1$ expoentes i em $[q(q-1)]$ são múltiplos de q, existem no máximo $q-1$ expoentes i em $[q(q-1)]$ que podem satisfazer (3.21). Isto é,

$$|\{\overline{a} \in \mathbb{Z}^*_{q^2} : \overline{a}^{n-1} = \overline{1} \text{ em } \mathbb{Z}_{q^2}\}| \leq q-1.$$

Substituindo esta desigualdade em (3.20), obtemos

$$P_n \leq \frac{n(q-1)}{q^2(n-1)} = \frac{n(q-1)}{q^2n-q^2} \leq \frac{n(q-1)}{q^2n-n} = \frac{n(q-1)}{n(q^2-1)} = \frac{1}{q+1}.$$

O resultado segue pois q é ímpar, já que n é ímpar, e daí $q+1 \geq 3$. □

Na proposição 3.7.2, mostramos que $P_n \leq \frac{1}{4}$, quando n é não é "livre de quadrados". Portanto, a partir deste momento, iremos assumir que n é "livre de quadrados". Pelo Teorema Fundamental da Aritmética, existem primos distintos $q_1, q_2, \ldots, q_k$ tais que

$$n = q_1 q_2 \cdots q_k.$$

Observe que $k \geq 2$ pois n não é primo. Para $i \in \{1, 2, \ldots, k\}$, como q_i é ímpar, existem inteiros positivos r_i e m_i tais que m_i é ímpar e $q_i - 1 = 2^{r_i} m_i$.

Mostraremos dois lemas auxiliares que estabelecem limites superiores para o número de $a \in [n]^*$ satisfazendo respectivamente **(i)** e **(ii)**.

Lema 3.7.3. *Vale a seguinte igualdade:*

$$|\{a \in [n]^* : \overline{a}^m = \overline{1} \text{ em } \mathbb{Z}_n\}| = (m, m_1)(m, m_2) \cdots (m, m_k).$$

Lema 3.7.4. *Para $0 \leq j \leq r-1$, $|\{a \in [n]^* : \overline{a}^{2^j m} = \overline{-1} \text{ em } \mathbb{Z}_n\}|$ é igual a 0, se $\min\{r_1, r_2, \ldots, r_k\} \leq j$, ou igual a $2^{jk}(m, m_1)(m, m_2) \cdots (m, m_k)$, caso contrário.*

Uma etapa comum à demonstração destes dois lemas é estabelecida no resultado enunciado a seguir.

Lema 3.7.5. *Sejam u, v e w inteiros tais que $u > 0$ e $v \in \{-1, 1\}$. Se*

$$\kappa_v^u(w) := \{a \in [w]^* : \overline{a}^u = \overline{v} \text{ em } \mathbb{Z}_w\},$$

então

$$|\kappa_v^u(n)| = |\kappa_v^u(q_1)||\kappa_v^u(q_2)| \cdots |\kappa_v^u(q_k)|.$$

Utilizando a notação definida no lema anterior, os lemas 3.7.3 e 3.7.4 fornecem limites superiores para respectivamente $|\kappa_1^m(n)|$ e $|\kappa_{-1}^{2^j m}(n)|$.

Demonstração do lema 3.7.5. Observe que as seguintes afirmações são equivalentes para $a \in [n]^*$:

(a) $\overline{a}^u = \overline{v}$ em $\mathbb{Z}_n$;

(b) $n|a^u - v$;

(c) para todo $i \in \{1, 2, \ldots, k\}$, $q_i|a^u - v$;

(d) para todo $i \in \{1, 2, \ldots, k\}$, $\overline{a}^u = \overline{v}$ em $\mathbb{Z}_{q_i}$.

O Teorema Fundamental da Aritmética estabelece a equivalência entre **(b)** e **(c)**.

Na seção 4, mostramos que a função $\Psi : \mathbb{Z}_n \to \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$ dada por $\Psi(\overline{a}) = (\overline{a}, \overline{a}, \ldots, \overline{a})$, para um inteiro a é uma bijeção. Pelo parágrafo anterior, Ψ leva o conjunto $\kappa_v^u(n)$ em $\kappa_v^u(q_1) \times \kappa_v^u(q_2) \times \cdots \times \kappa_v^u(q_k)$. Portanto,

$$|\kappa_v^u(n)| = |\kappa_v^u(q_1) \times \kappa_v^u(q_2) \times \cdots \times \kappa_v^u(q_k)|$$

e o resultado segue. □

Demonstração do lema 3.7.3. Pelo lema 3.7.5 será suficiente estabelecer que, para cada $i \in \{1, 2 \ldots, k\}$,

$$|\kappa_1^m(q_i)| = (m, m_i). \tag{3.22}$$

Pela proposição 3.6.3, $\mathbb{Z}_{q_i}^*$ possui um gerador $\overline{a}$ (cuja ordem é $q_i - 1 = 2^{r_i} m_i$). Portanto, qualquer elemento de $\mathbb{Z}_{q_i}^*$ é uma potência de $\overline{a}$. Isto é,

$$\mathbb{Z}_{q_i}^* = \{\overline{a}, \overline{a}^2, \ldots, \overline{a}^{q_i - 1}\}.$$

Necessitamos determinar todos os expoentes j tais que $\overline{a}^j \in \kappa_1^m(q_i)$. Isto é,

$$\overline{a}^{jm} = (\overline{a}^j)^m = \overline{1} \text{ em } \mathbb{Z}_{q_i}.$$

Pelo teorema 3.4.1, $2^{r_i} m_i | jm$. Se $h = \frac{m_i}{(m, m_i)}$, então $2^{r_i} h | j$, pois m é ímpar. Como existem (m, m_i) múltiplos de $2^{r_i} h$ em $[2^{r_i} m_i]$, temos que (3.22) segue. □

Demonstração do lema 3.7.4. Pelo lema 3.7.5 será suficiente estabelecer que, para cada $i \in \{1, 2 \ldots, k\}$,

$$|\kappa_{-1}^{2^j m}(q_i)| \leq \begin{cases} 2^j(m, m_i) & \text{quando } r_i > j \\ 0 & \text{quando } r_i \leq j \end{cases} \tag{3.23}$$

Pela proposição 3.6.3, $\mathbb{Z}_{q_i}^*$ possui um gerador $\overline{a}$ (cuja ordem é $q_i - 1 = 2^{r_i} m_i$). Portanto, qualquer elemento de $\mathbb{Z}_{q_i}^*$ é uma potência de $\overline{a}$. Isto é,

$$\mathbb{Z}_{q_i}^* = \{\overline{a}, \overline{a}^2, \ldots, \overline{a}^{q_i - 1}\}.$$

Necessitamos determinar todos os expoentes k tais que $\overline{a}^k \in \kappa_{-1}^{2^j m}(q_i)$. Isto é,

$$\overline{a}^{2^j mk} = (\overline{a}^k)^{2^j m} = \overline{-1} \text{ em } \mathbb{Z}_{q_i}.$$

Pelo teorema 3.4.1,

$$2^{r_i} m_i | 2^{j+1} mk \quad \text{e} \quad 2^{r_i} m_i \not| 2^j mk. \tag{3.24}$$

Em particular, $m_i | mk$, pois m_i é ímpar. Se $r_i \leq j$, então não existe k satisfazendo (3.24) e daí $\kappa_{-1}^{2^j m}(q_i) = \varnothing$. Neste caso, (3.23) é verificada. Vamos supor que $j < r_i$. Podemos, então, reescrever (3.24) como:

$$2^{r_i - j - 1} m_i | mk \quad \text{e} \quad 2^{r_i - j} m_i \not| mk. \tag{3.25}$$

Se $h = \frac{m_i}{(m, m_i)}$, então

$$2^{r_i - j - 1} h | k \quad \text{e} \quad 2^{r_i - j} h \not| k. \tag{3.26}$$

Conseqüentemente, k tem que ser um múltiplo ímpar de $2^{r_i - j - 1} h$. Note que existem $2^{j+1}(m, m_i)$ múltiplos de $2^{r_i - j - 1} h$ em $[2^{r_i} m_i]$, dos quais metade são múltiplos ímpares. Portanto, (3.23) segue. □

Demonstração do teorema 3.7.1. Pela proposição 3.7.2, podemos assumir que n é livre de quadrados. Pelos lemas 3.7.3 e 3.7.4, n não é pseudoprimo com respeito a exatamente

$$(m, m_1)(m, m_2) \cdots (m, m_k)(1 + 1 + 2^k + \cdots + 2^{k(s-1)})$$

elementos de $[n]^*$, onde $s = \min\{r, r_1, r_2, \ldots, r_k\}$. Portanto,

$$P_n = \frac{(m, m_1)(m, m_2) \cdots (m, m_k)(1 + 1 + 2^k + \cdots + 2^{k(s-1)})}{n - 1}.$$

Como $(q_1-1)(q_2-1)\cdots(q_k-1) < q_1q_2\cdots q_k - 1 = n-1$, temos que

$$P_n < \frac{(m,m_1)(m,m_2)\cdots(m,m_k)(1+1+2^k+\cdots+2^{k(s-1)})}{2^{r_1}m_1 2^{r_2}m_2\cdots 2^{r_k}m_k}.$$

Podemos reescrever esta desigualdade como

$$P_n < \frac{(m,m_1)(m,m_2)\cdots(m,m_k)}{m_1m_2\cdots m_k}\,\frac{(1+1+2^k+\cdots+2^{k(s-1)})}{2^{r_1+r_2+\cdots+r_k}}. \tag{3.27}$$

Utilizando a expressão da soma de uma progressão geométrica e também o fato de que $s \geq 1$, temos que

$$\begin{aligned}
1+1+2^k+\cdots+2^{k(s-1)} &= 1+\frac{2^{ks}-1}{2^k-1} \\
&= 1+\frac{2^{ks}}{2^k-1}-\frac{1}{2^k-1} \\
&= \frac{2^{ks}}{2^k-1}-\frac{2^k-2}{2^k-1} \\
&= 2^{ks}\left(\frac{1}{2^k-1}+\frac{2^k-2}{2^{ks}(2^k-1)}\right) \\
&\leq 2^{ks}\left(\frac{1}{2^k-1}+\frac{2^k-2}{2^k(2^k-1)}\right) \\
&= 2^{ks}\left(\frac{2^k+2^k-2}{2^k(2^k-1)}\right) \\
&= 2^{ks}\left(\frac{2(2^k-1)}{2^k(2^k-1)}\right) \\
&= 2^{ks}\left(\frac{1}{2^{k-1}}\right).
\end{aligned}$$

Substituindo este limite superior para $1+1+2^k+\cdots+2^{k(s-1)}$ em (3.27), obtemos

$$P_n < \frac{(m,m_1)(m,m_2)\cdots(m,m_k)}{m_1m_2\cdots m_k}\,\frac{2^{ks}}{2^{r_1+r_2+\cdots+r_k}}\,\frac{1}{2^{k-1}}. \tag{3.28}$$

Como, para todo $i \in \{1, 2, \ldots, k\}$, $(m, m_i) \leq m_i$ e $2^s \leq 2^{r_i}$, temos que:

$$\frac{(m, m_1)(m, m_2) \cdots (m, m_k)}{m_1 m_2 \cdots m_k} \leq 1 \text{ e } \frac{2^{ks}}{2^{r_1 + r_2 + \cdots + r_k}} \leq 1.$$

Conseqüentemente,

$$P_n < \frac{1}{2^{k-1}}$$

e o resultado segue quando $k \geq 3$. Podemos assumir que $k = 2$. Neste caso, (3.28) pode ser escrita como:

$$P_n < \frac{(m, m_1)(m, m_2)}{m_1 m_2} \frac{2^{2s-1}}{2^{r_1 + r_2}}. \tag{3.29}$$

Note que o resultado também segue neste caso, a menos que $r_1 = r_2 = s$, $(m, m_1) = m_1$ e $(m, m_2) = m_2$. Vamos supor que este é o caso. Como $s \leq r$, temos que:

$$q_1 - 1 | n - 1 \quad \text{e} \quad q_2 - 1 | n - 1. \tag{3.30}$$

Observe que $n - 1 = q_1 q_2 - 1 = (q_1 - 1)q_2 + q_2 - 1$. Por (3.30), $q_1 - 1 | q_2 - 1$. De maneira análoga, $q_2 - 1 | q_1 - 1$. Portanto, $q_1 = q_2$. Com esta contradição concluímos a demonstração do teorema. □

3.8 A exponenciação é rápida em $\mathbb{Z}_n$

Com o objetivo de estimar a complexidade da exponenciação em $\mathbb{Z}_n$, iniciamos esta seção analisando o custo das operações aritméticas usuais. Por simplicidade, consideraremos que os operandos são inteiros representados em binário. Como se sabe, o número de dígitos de n, quando apresentado como um numeral escrito na base 2, é $\lfloor \log n \rfloor + 1$.

Proposição 3.8.1. *Sejam a e b inteiros positivos, cada um com no máximo k dígitos em sua representação binária. As operações de adição, subtração e comparação entre a e b têm custo $O(k)$; a multiplicação e a divisão têm custo $O(k^2)$.*

Demonstração. No algoritmo usual utilizado para a adição, determinam-se os dígitos do resultado, por ordem, iniciando-se pelo localizado na direita da representação na base 2. Vamos supor que os $i-1$ primeiros dígitos foram determinados. Descreveremos como calcular o i-ésimo. O seu valor será 0 ou 1 dependendo de existir respectivamente um número par ou ímpar de dígitos iguais a 1 nas posições da memória que guardam: o i-ésimo dígito de a; o i-ésimo dígito de b; e o "vai um". (A posição "vai um" tem valor inicial 0 e é continuamente atualizada da seguinte forma: recebe valor 0 se existe no máximo um dígito igual a 1 nas três posições de memória que acabamos de descrever; caso contrário, recebe valor 1.) Independentemente de como o leitor deseje calcular quantas "tarefas básicas" foram realizadas para o cálculo deste dígito, este número será limitado, digamos, por l. Portanto, o número de "tarefas básicas" realizadas para o cálculo da adição de a com b é limitado por lk. Sendo assim, este algoritmo tem custo $O(k)$. (Caso o "vai um" final tenha valor 1, o resultado da adição terá $k+1$ dígitos e o $(k+1)$-ésimo dígito terá valor 1 — o que, evidentemente, não altera a complexidade assintótica $O(k)$ da operação.)

Para descrever o algoritmo da subtração, precisamos comparar a com b. Inicialmente, completamos as representações na base 2 de a e b com 0's à esquerda, de forma que ambos fiquem com k "dígitos". (Quando a e b são armazenados em k *bits* de memória, esta tarefa foi executada.) Vamos assumir que percorremos, iniciando pela esquerda, os $i-1$-ésimos primeiros dígitos das representações de a e b na base 2 sem chegarmos a uma conclusão. Na etapa seguinte, verificamos se os i-ésimos dígitos de a e b são iguais ou diferentes. Se forem diferentes, então um deles será 0 e o outro será 1. O número que tiver o i-ésimo dígito igual a 1 será o maior. Senão, quando $i = k$, a e b serão iguais, e, quando $i < k$, continuamos com o processo, pois ainda não fomos capazes de chegar a uma conclusão. Mais uma vez, fica claro que este algoritmo tem custo $O(k)$.

Para subtrair, necessitamos comparar a com b. Esta fase do algoritmo tem custo $O(k)$. Caso b seja maior ou igual a a, o resultado de $a-b$ será um inteiro não-positivo. Portanto, calculamos $b-a$ e alteramos o sinal. Vamos descrever como fazer a subtração apenas no caso em que a é maior que b. O algoritmo é o mesmo da adição exceto na alteração do valor encontrado em "vai um". O "vai um"

ficará igual a 1 se e somente se existem mais dígitos iguais a 1 nas posições que armazenam o i-ésimo dígito de b e o "vai um" que na posição que guarda o i-ésimo dígito de a. Claramente o custo da segunda fase deste algoritmo também é $O(k)$. Portanto, a subtração tem custo $O(k)$.

Reduziremos a multiplicação à adição de até k números, cada um com no máximo $2k$ dígitos na base 2. (Mais ainda, o resultado da adição de qualquer subconjunto destes números possui no máximo $2k$ dígitos.) Portanto, realizamos no máximo $k-1$ adições, cada uma com custo $O(k)$. O custo final do algoritmo da multiplicação será $O(k^2)$. Para encontrar o conjunto de números a serem adicionados, percorremos b da direita para a esquerda. Caso o i-ésimo dígito encontrado seja igual 1, coloca-se na família um número obtido de a ampliando, à direita, a sua representação com $i-1$ dígitos iguais a 0. Caso o i-ésimo dígito seja 0, não acrescenta-se número ao conjunto[3].

A divisão pode ser reduzida apenas à comparação e subtração de números, cada um com até k dígitos. Como são realizadas no máximo k de cada uma destas operações, o custo do algoritmo será $O(k^2)$, já que o custo individual de cada operação é $O(k)$. □

Do resultado anterior, temos como estimar o custo de realizar operações em $\mathbb{Z}_n$. Isto será estabelecido na próxima proposição. Salientamos que o custo para realizar a multiplicação pode ser melhorado desde que um algoritmo mais eficiente para realizar a multiplicação e divisão nos inteiros seja utilizado[4].

Proposição 3.8.2. *Seja n um inteiro positivo. Se a e b pertencem a $[n]$, então o custo de calcular $a \oplus b$ e $a \otimes b$ é respectivamente $O(\log n)$ e $O(\log^2 n)$.*

Demonstração. Seja k o número de dígitos de n, isto é, $k = \lfloor \log n \rfloor + 1$. Como $a \oplus b$ é igual a $a+b$, quando $a+b < n$, ou $(a+b)-n$, quando

[3] O custo de encontrar este conjunto de números pode ser desprezado, pois a adição pode ser feita utilizando-se apenas a representação de a na base 2. Outrossim, o custo de encontrar cada um destes números é $O(k)$ e, como encontramos no máximo k deles, o custo final para encontrá-los a todos é $O(k^2)$ — o que não altera o custo assintótico do algoritmo da multiplicação.

[4] Veja, também, para uma discussão sobre algoritmos mais eficientes para as operações com inteiros gigantes, o livro de Crandall e Pomerance [14]. Tanto a multiplicação quanto a divisão podem ser realizadas em $O(k(\log k)^{O(1)})$.

$a+b \geq n$, realizamos, no caso que requer o maior número de operações, uma adição, uma comparação e uma subtração de inteiros positivos com no máximo $k+1$ dígitos. Pela proposição 3.8.1, o custo de realizar cada uma destas operações é $O(k) = O(\log n)$. Conseqüentemente, o custo de realizar todas as três também é $O(\log n)$.

Observe que $a \otimes b$ é igual ao resto da divisão de ab por n, que são inteiros positivos com no máximo $2k$ dígitos. Pela proposição 3.8.1, o custo de realizar cada uma destas duas operações é $O(\log^2 n)$. Portanto, o custo da multiplicação em $[n]$ é $O(\log^2 n)$. □

Sejam m e n inteiros positivos. Neste parágrafo, descreveremos um algoritmo para calcular $\overline{a}^m$ em $\mathbb{Z}_n$, onde $a \in [n]$.

No computador, a representação de m é na base 2. Logo existem $r_0, r_1, \ldots, r_{k-1}, r_k$ pertencentes ao conjunto $\{0,1\}$, com $r_k \neq 0$, isto é, $r_k = 1$, tais que

$$m = (r_k r_{k-1} \cdots r_1 r_0)_2.$$

Ou seja

$$m = r_k 2^k + r_{k-1} 2^{k-1} + \cdots + r_1 2^1 + r_0 2^0.$$

Portanto,

$$\begin{aligned} \overline{a}^m &= \overline{a}^{r_k 2^k + r_{k-1} 2^{k-1} + \cdots + r_1 2^1 + r_0 2^0} \\ &= \overline{a}^{r_k 2^k} \overline{a}^{r_{k-1} 2^{k-1}} \cdots \overline{a}^{r_1 2^1} \overline{a}^{r_0 2^0} \\ &= \left(\overline{a}^{2^k}\right)^{r_k} \left(\overline{a}^{2^{k-1}}\right)^{r_{k-1}} \cdots \left(\overline{a}^{2^1}\right)^{r_1} \left(\overline{a}^{2^0}\right)^{r_0} \end{aligned}$$

Como $\left(\overline{a}^{2^i}\right)^{r_i}$ é igual a $\overline{a}^{2^i}$, quando $r_i \neq 0$ (isto é, $r_i = 1$), ou $\overline{1}$, quando $r_i = 0$, temos que

$$\overline{a}^m = \prod_{i \in [k+1] : r_i \neq 0} \overline{a}^{2^i}. \tag{3.31}$$

Portanto, necessitamos calcular

$$\overline{a}^{2^0}, \overline{a}^{2^1}, \overline{a}^{2^2}, \ldots, \overline{a}^{2^{k-1}}, \overline{a}^{2^k} \text{ em } \mathbb{Z}_n. \tag{3.32}$$

Como cada elemento desta seqüência é igual ao quadrado do elemento que o antecede, podemos encontrar todos os elementos de (3.32) realizando k multiplicações em $\mathbb{Z}_n$. (Não é necessário calcular o primeiro elemento, já que $\overline{a}^{2^0} = \overline{a}^1 = \overline{a}$.) Por (3.31), necessitamos realizar no máximo mais k multiplicações em $\mathbb{Z}_n$ para calcular $\overline{a}^m$.

Teorema 3.8.3. *Sejam m e n inteiros positivos. Se $a \in [n]$, então pode-se calcular $\overline{a}^m$ em $\mathbb{Z}_n$ realizando no máximo $2\lfloor \log m \rfloor$ multiplicações em $\mathbb{Z}_n$. O custo de calcular $\overline{a}^m$ em $\mathbb{Z}_n$ é $O(\log m \log^2 n)$.*

Demonstração. Pelo parágrafo anterior, realizamos no máximo $2k$ multiplicações em $\mathbb{Z}_n$ para calcular $\overline{a}^m$ em $\mathbb{Z}_n$. Como $k+1$ é o número de dígitos de m na base 2, temos que

$$k + 1 = \lfloor \log m \rfloor + 1$$

e daí $k = \lfloor \log m \rfloor$. A primeira parte do resultado segue. Para concluir a demonstração do teorema, necessitamos apenas estimar o custo de calcular $\overline{a}^m$ em $\mathbb{Z}_n$. Pela proposição 3.8.2, cada multiplicação em $\mathbb{Z}_n$ tem custo $O(\log^2 n)$. Portanto, o custo de realizar no máximo $2\lfloor \log m \rfloor$ é $O(\log m \log^2 n)$. □

Salientamos que o custo da exponenciação é polinomial apenas em $\mathbb{Z}_n$, já que mantemos o controle do número de dígitos na representação na base 2 de todos os elementos da seqüência descrita em (3.32), bem como de todos os produtos parciais obtidos quando computamos o produto descrito em (3.31). (O resultado de uma multiplicação em $\mathbb{Z}_n$ é sempre representado por um dos possíveis restos da divisão por n, isto é, tem a forma $\overline{b}$, com b pertence a $[n]$.) Caso optemos por realizar exponenciação nos inteiros, o seu custo seria exponencial porque o número de dígitos iria duplicar a cada quadrado computado. Em particular, caso a e m sejam inteiros positivos, o número de dígitos de a^m é aproximadamente

$$\log a^m = m \log a = 2^{\log m} \log a.$$

Logo o número de dígitos de a^m é exponencial no tamanho da entrada do problema, que é a e m. (Os números a e m ocupam respectivamente $\lfloor \log a \rfloor + 1$ e $\lfloor \log m \rfloor + 1$ posições de memória.)

Teorema 3.8.4. *Seja n um inteiro positivo. Se $a \in [n]^*$ e n é ímpar, então é possível decidir quando n é pseudoprimo com respeito a a realizando no máximo $2\lfloor \log n \rfloor$ multiplicações em $\mathbb{Z}_n$. O custo desta decisão é $O(\log^3 n)$.*

Demonstração. Existem inteiros positivos r e m tais que m é ímpar e $n - 1 = 2^r m$. Lembre-se que n é *pseudoprimo com respeito* a a quando:

(i) $\overline{a}^m = \overline{1}$; ou

(ii) $\overline{a}^{2^i m} = \overline{-1}$, para algum inteiro i tal que $0 \leq i \leq r-1$.

Portanto, para decidirmos quando n é pseudoprimo com respeito a a, necessitamos calcular os elementos da seqüência

$$\overline{a}^m, \overline{a}^{2^1 m}, \overline{a}^{2^2 m}, \ldots, \overline{a}^{2^{r-2} m}, \overline{a}^{2^{r-1} m} \text{ em } \mathbb{Z}_n. \tag{3.33}$$

Ao percorrermos esta seqüência da esquerda para a direita, ao elevarmos o quadrado de um elemento, obtemos o seu consecutivo, pois

$$\left(\overline{a}^{2^i m}\right)^2 = \overline{a}^{2 \cdot 2^i m} = \overline{a}^{2^{i+1} m}.$$

A partir de $\overline{a}^m$, necessitamos realizar r multiplicações em $\mathbb{Z}_n$ para encontrar toda a seqüência. Pelo teorema 3.8.3, $\overline{a}^m$ pode ser computado com no máximo $2\lfloor \log m \rfloor$ multiplicações em $\mathbb{Z}_n$. Conseqüentemente, são necessárias

$$r + 2\lfloor \log m \rfloor \leq 2\lfloor r + \log m \rfloor = 2\lfloor \log 2^r m \rfloor = 2\lfloor \log(n-1) \rfloor.$$

Pela proposição 3.8.2, o custo de realizar todos estes produtos em $\mathbb{Z}_n$ será $O(\log^3 n)$. O resultado segue, já que, pela proposição 3.8.1, cada uma das r comparações envolvidas nesta decisão tem custo $O(\log n)$. □

3.9 Quase decidindo primalidade em tempo polinomial

Seja n um número positivo ímpar. Nesta seção, vamos apresentar um algoritmo, desenvolvido por Rabin [45], que decide se n é `PRIMO` ou `COMPOSTO`. Quando a saída do algoritmo for n é `COMPOSTO`, então realmente n é composto. Mas, se o algoritmo afirmar que n é `PRIMO`, eventualmente pode ocorrer de n não o ser. Isto é, o algoritmo pode falhar e detectar primalidade onde ela não exista[5].

[5] Em outras palavras, trata-se de um algoritmo de Monte Carlo baseado-no-não para o problema de se determinar se n é primo.

Entrada:
    n: um inteiro ímpar.
Saída:
    PRIMO ou COMPOSTO, decidindo a primalidade de n.

primalidade(n):
    escolha, aleatoriamente, $a_1, a_2, \ldots, a_{5k} \in [n]^*$
    para i de 1 a $5k$, faça:
        se n *não é* pseudoprimo com relação a a_i, retorne COMPOSTO
    retorne PRIMO

Figura 3.1: Algoritmo de Rabin.

Tememos que, após diversas seções voltadas a puro ferramental matemático, o leitor tenha se esquecido que este texto destina-se à discussão de algoritmos e técnicas randomizadas e esteja se perguntando se não teria valido mais a pena — para dominar o problema da primalidade que dele exigiu tamanha preparação — ter estudado de uma vez o algoritmo AKS, que a decide deterministicamente. Mais uma vez, porém, ressaltamos que a matemática necessária para compreender o AKS é *extremamente* mais complexa. Ademais, como de costume, a probabilidade de erro do algoritmo de Rabin pode ser escolhida tão pequena quanto se queira. Pode ser escolhida, inclusive, como menor do que a probabilidade de que haja uma falha no hardware do computador, de forma que se pode conseguir que a resposta com ele obtida seja, para todos os efeitos, *tão boa quanto* a do AKS!

A figura 3.1 apresenta o pseudo-código para o algoritmo de Rabin para decidir primalidade.

Como já temos todo o material necessário, a análise do algoritmo de Rabin é deveras descomplicada. Pelo teorema 3.4.6, caso a saída do algoritmo de Rabin seja COMPOSTO, então n é realmente composto[6]. Por outro lado, a chance de que um n composto seja pseudoprimo para uma base qualquer considerada em uma iteração do laço do algoritmo é inferior a $\frac{1}{4}$, como nos garante o teorema 3.7.1. Dessa

[6] Raciocínio alternativo: entradas em que n seja primo nunca levarão o algoritmo a responder COMPOSTO.

forma, a probabilidade de que o algoritmo responda *erroneamente* `PRIMO` será inferior a

$$\left(\frac{1}{4}\right)^{5k} = \frac{1}{2^{10k}} = \left(\frac{1}{2^{10}}\right)^{k} = \left(\frac{1}{1024}\right)^{k} < \left(\frac{1}{1000}\right)^{k} = \left(\frac{1}{10^{3}}\right)^{k} = \frac{1}{10^{3k}}.$$

Portanto, a probabilidade global de erro do algoritmo de Rabin é

$$\varepsilon < \frac{1}{10^{3k}}.$$

Para que esta probabilidade seja feita tão pequena quando se queira, é suficiente escolhermos valores maiores[7] para k.

Pelo teorema 3.8.4, necessitamos realizar no máximo

$$10k \log n$$

multiplicações em $\mathbb{Z}_n$ para executar o algoritmo de Rabin. Mais ainda, o custo deste algoritmo será $O(k \log^3 n)$. Isto é, será polinomial quando k for fixo. Caso sejam utilizados algoritmos mais rápidos para executar as operações de multiplicação e divisão, o custo do algoritmo de Rabin cai para $O(k \log^2 n (\log \log n)^{O(1)})$.

3.10 A importância de encontrar números primos grandes: o RSA

Nesta última seção, apresentamos o algoritmo RSA para criptografia, em que a capacidade de se encontrar primos grandes tem papel fundamental.

O *alfabeto* (conjunto de caracteres) utilizado é o mesmo para todos os usuários do sistema através do qual serão transmitidas as mensagens criptografadas pelo algoritmo (pode conter, por exemplo, todos os símbolos disponíveis em um teclado de computador). Como o alfabeto utilizado em nossa escrita, o alfabeto utilizado pelo RSA é *ordenado*. Assumamos que o alfabeto em questão possua L letras. Estão fixos dois inteiros r e s, com $r < s$, para serem utilizados por todos usuários.

[7] Para $k = 10$, a probabilidade de erro do algoritmo de Rabin já pode ser considerada, para todos os efeitos, nula!

Um usuário qualquer, que deseja *receber* mensagens criptografadas pelo RSA, precisa escolher dois números primos grandes p e q de forma que

$$L^r < pq < L^s. \tag{3.34}$$

Seja $n = pq$. Observe que $\phi(n) = (p-1)(q-1)$. O usuário escolhe um elemento $\overline{e}$ inversível em $\mathbb{Z}_{(p-1)(q-1)}$ e calcula o seu inverso[8] $\overline{d}$ em $\mathbb{Z}_{(p-1)(q-1)}$. Isto é,

$$\overline{ed} = \overline{e}\overline{d} = \overline{1} \text{ em } \mathbb{Z}_{(p-1)(q-1)}. \tag{3.35}$$

Podemos assumir que d e e são inteiros em $[(p-1)(q-1)]$. O usuário torna públicas as "chaves" n, e (para codificação) e mantém secreta a "chave" d (para decodificação). Os primos p e q podem ser destruídos.

Para *encaminhar* uma mensagem para um usuário que disponibilizou suas chaves públicas n, e, utilizamos o seguinte algoritmo: divide-se a mensagem a ser cifrada em blocos com r letras (caso o último bloco fique com menos de r letras, completa-se as posições vazias com a última letra do alfabeto — que é, em geral, um espaço em branco). Cada bloco B com r letras pode ser visto como um número inteiro não-negativo b com até r dígitos escrito na base L (e tendo como dígitos as letras do alfabeto). Observe que $b < L^r$ e, sendo assim, $b \in [n]$. Calcula-se $b' = \text{res}(b^e, n)$. Como $n < L^s$, o inteiro não-negativo b' tem no máximo s dígitos quando escrito na base L. Em particular, pode ser transformado em um bloco B' de s letras na base L. A mensagem a ser encaminhada é obtida substituindo-se cada bloco B pelo bloco B' correspondente.

Para *decifrar* uma mensagem que chega, o usuário a quebra em blocos de s letras, cada um dos quais associado a um inteiro $b' < L^s$. O algoritmo utilizado para cifrar estabelece $b' = \text{res}(b^e, n)$. Pelo teorema 3.4.3,

$$\overline{b'}^d = \left(\overline{b}^e\right)^d = \overline{b}^{de} = \overline{b}^{\lambda\phi(n)+1} = \left(\overline{b}^{\phi(n)}\right)^\lambda \overline{b} = \overline{b} \text{ em } \mathbb{Z}_n$$

(como $\overline{d}$ é o inverso de $\overline{e}$ em $\mathbb{Z}_{(p-1)(q-1)}$, temos que $de = \lambda\phi(n) + 1$, para algum λ). Portanto, com o conhecimento de d, obtemos b a partir de b', e a mensagem original é recuperada.

[8] Na seção 3.2 caracterizamos os elementos inversíveis de $\mathbb{Z}_{(p-1)(q-1)}$ e apresentamos implicitamente um algoritmo com custo polinomial para o cálculo de seu inverso.

É possível mostrar que, caso alguém possua um algoritmo para sempre obter b a partir de b' — isto é, quebrando o sistema criptográfico e lendo efetivamente as mensagens — , então este alguém consegue fatorar n rapidamente. Assume-se, no entanto, que esta tarefa é impossível do ponto de vista computacional.

3.11 Exercícios

1. Determine todos os divisores de zero em $\mathbb{Z}_{30}$.

2. Existem números inteiros a e b tais que $1 = 1514a + 1357b$?

3. Em $\mathbb{Z}_{1514}$, $\overline{1357}$ é inversível? Em caso afirmativo, determine seu inverso.

4. Em $\mathbb{Z}_n^*$, qual a ordem de $\overline{a}^{45}$ quando a ordem de $\overline{a}$ é 132?

5. Qual o resto da divisão de 2^{10123} por 17?

6. Encontre todas as raízes do polinômio $p(X) = X^2 - \overline{1}$ em $\mathbb{Z}_{32}$. Resolva o mesmo problema quando $\mathbb{Z}_{32}$ é substituído por $\mathbb{Z}_{2^n}$.

7. Encontre o menor inteiro positivo que deixa restos 5, 2 e 3 quando dividido respectivamente por 9, 10 e 11.

8. Qual o valor de $\phi(1518)$?

9. Encontre um gerador para $\mathbb{Z}_{25}^*$. Determine a ordem de todos os elementos de $\mathbb{Z}_{25}^*$.

10. Sejam p e n inteiros positivos. Quando p é primo, mostre que $\mathbb{Z}_{p^n}^*$ possui um gerador.

11. Mostre que $\overline{a}^{560} = \overline{1}$ para todo $\overline{a} \in \mathbb{Z}_{561}^*$.

12. O inteiro 341 é pseudoprimo com respeito a 2?

13. Calcule $\overline{7}^{917}$ em $\mathbb{Z}_{1234}$.

3.12 Notas bibliográficas

Uma boa descrição do AKS pode ser encontrada no artigo de Granville [27]. Caso deseje entender, com todos os detalhes, as razões pelas quais o algoritmo funciona, o leitor encontrará no livro de Coutinho [13] uma excelente alternativa em língua portuguesa.

Para uma discussão mais detalhada sobre o sistema de criptografia com chave pública conhecido como RSA, outro livro de Coutinho [12] constitui excelente escolha em língua pátria. Pode-se também recorrer ao livro de Koblitz [34], que é escrito em inglês (e poderíamos tentar adaptar, aqui, a piada de Ribenboim sobre estudar criptografia em uma linguagem cifrada! — veja a página 104 de [47]).

Por fim, para o leitor interessado em saber tudo sobre números primos, indicamos o excelente livro de Crandall e Pomerance [14], que incorpora os mais recentes avanços — incluindo o AKS — e traz uma abordagem detalhada dos aspectos algorítmicos envolvidos.

Capítulo 4

Geometria Computacional

Geometria computacional é a área da computação que estuda a complexidade de algoritmos, problemas e estruturas de dados de natureza geométrica. O problema mais famoso é talvez o fecho convexo, que consiste em determinar o menor polígono convexo que contém um dado conjunto de pontos.

Diversas razões justificam o fato de os algoritmos randomizados terem ganho importância central no estudo da geometria computacional. Primeiro, algoritmos randomizados são, como já o observamos nos capítulos anteriores, freqüentemente mais simples e eficientes na prática do que seus equivalentes determinísticos. Segundo, algoritmos randomizados para problemas no plano geralmente se estendem para espaços d-dimensionais, com pequenas modificações. Finalmente, os algoritmos determinísticos mais eficientes para vários problemas foram obtidos a partir das de-randomizações de algoritmos randomizados. Por exemplo, o único algoritmo ótimo conhecido para computação do fecho convexo em dimensões ímpares maiores que 3 é a de-randomização de um algoritmo randomizado incremental.

Devido à natureza contínua dos problemas, o modelo computacional mais utilizado é o *real RAM*, onde operações algébricas com números reais podem ser realizadas em tempo constante. Os algo-

ritmos são normalmente descritos de modo geométrico, sem entrar em detalhes de implementação de funções como cálculo de ângulos e distâncias. Assume-se que qualquer computação envolvendo um número constante de primitivas geométricas pode ser realizada em tempo $O(1)$.

Também assume-se que a dimensão d do espaço Euclidiano é uma constante. Portanto, um algoritmo cuja complexidade seja $O(2^d n)$ tem sua complexidade escrita como $O(n)$. Dependências exponenciais na dimensão do espaço são comuns. Sendo assim, a maior parte dos algoritmos não é eficiente para dimensões muito altas.

Outra prática comum em geometria computacional é assumir que a entrada do problema está em *posição geral*. Não tentamos definir formalmente posição geral, mas a idéia é desconsiderar propriedades que se perdem com uma perturbação infinitesimal da entrada. Por exemplo, caso a entrada seja um conjunto de pontos, podemos assumir que não há três pontos colineares, ou que não há quatro pontos cocirculares. Caso a entrada seja um conjunto de retas, podemos assumir que não há retas verticais, horizontais, ou duas retas paralelas. Na maioria dos casos, assumir posição geral não altera a complexidade do problema e simplifica a explicação e a análise dos algoritmos.

Na seção 4.1, apresentamos um algoritmo randomizado incremental para o problema de programação linear com um número constante de variáveis. Na seção 4.2, introduzimos o conceito de função hash randomizada, que utilizamos no algoritmo para obtenção do par de pontos mais próximos descrito na seção 4.3. O leitor encontrará, na seção 4.4, diversos exercícios algorítmicos usando as técnicas introduzidas neste capítulo. E, como de costume, incluímos notas bibliográficas e comentários mais avançados na seção 4.5.

4.1 Programação linear

Um *problema de programação linear* com d variáveis consiste em determinar o vetor d-dimensional X que maximiza a função linear $f = CX$ e que satisfaz um sistema de desigualdades lineares $AX \leq B$.

Reescrevendo o problema sem usar notação matricial temos:

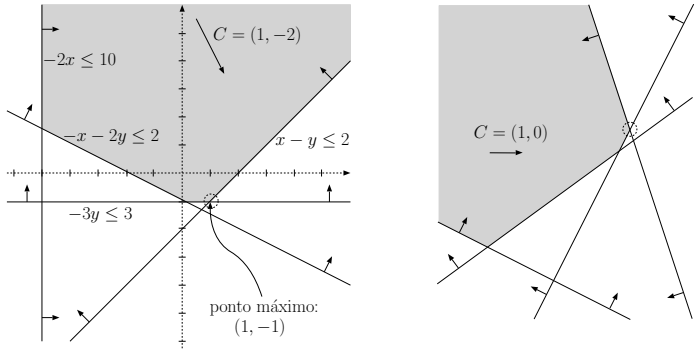
$$\begin{aligned} \text{Maximizar:} \quad & f = c_1x_1 + c_2x_2 + \cdots + c_dx_d \\ \text{Satisfazendo:} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d \leq b_1 \\ & \vdots \\ & a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d \leq b_n. \end{aligned}$$

Geometricamente, cada desigualdade linear representa um semi-espaço d-dimensional. A interseção dos n semi-espaços é chamada de *região viável*. Maximizar a função linear $f = CX$ satisfazendo as desigualdades $AX \leq B$ consiste em determinar um ponto extremo na direção C, que esteja contido na região viável. Um problema de programação linear com duas variáveis está ilustrado na figura 4.1(a).

Concentramos nossa explicação no caso de apenas duas variáveis ($d = 2$), mencionando brevemente como estender o algoritmo para um número arbitrário de variáveis. Sem perda de generalidade, assumimos que $C = (1, 0)$, pois os demais casos podem ser reduzidos ao caso $C = (1, 0)$ rodando o espaço (figura 4.1(b)). Assim, estamos interessados em obter o ponto mais à direita da interseção de um conjunto de semiplanos S.

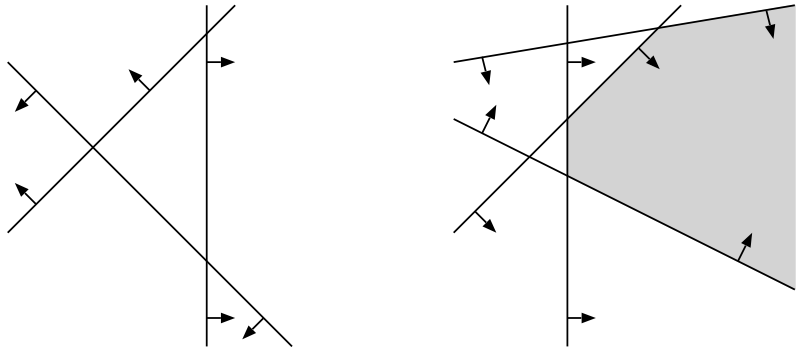
Existem dois casos especiais em que o problema não possui solução. O primeiro caso ocorre quando a região viável é vazia (figura 4.2(a)) e é chamado de *problema inviável*. O segundo caso ocorre quando a região viável não é limitada do lado direito (figura 4.2(b)) e é chamado de *problema ilimitado*. Dizemos que dois semiplanos $s_1, s_2 \in S$ *limitam o problema* se a região formada por $s_1 \cap s_2$ é limitada do lado direito.

O caso de o problema ser ilimitado é o primeiro que tratamos. Desejamos obter um algoritmo que ou diga que o problema é ilimitado ou encontre dois semiplanos que limitem o problema. Dado um semiplano s, definimos $N(s)$ como o vetor unitário normal à borda de s e que aponta para o interior de s. Definimos $N'(s)$ como o ângulo entre $N(s)$ e $(-1, 0)$, com $-\pi < N'(s) \leq \pi$. Seja s_1 o semiplano com $N'(s) \geq 0$ que minimiza $N'(s)$ e s_2 o semiplano com $N'(s) < 0$ que maximiza $N'(s)$. É possível provar que s_1 e s_2 limitam o problema se $N'(s_1) - N'(s_2) < \pi$. O problema é ilimitado se s_1 ou s_2 não existir, ou se $N'(s_1) - N'(s_2) > \pi$. O caso $N'(s_1) - N'(s_2) = \pi$ é mais delicado, mas não ocorre quando os semiplanos estão em posição geral.



(a) (b)

Figura 4.1: (a) Interpretação geométrica de um problema de programação linear com duas variáveis. (b) O mesmo problema após rotação.



(a) (b)

Figura 4.2: (a) Problema de programação linear inviável. (b) Problema de programação linear ilimitado.

Usando o procedimento descrito no parágrafo anterior, não só podemos nos concentrar apenas em problemas limitados, como também sabemos como obter duas restrições que limitam o problema, caso elas existam. Esta é a condição inicial do nosso algoritmo incremental. A partir daí, acrescentamos as restrições uma a uma, atualizando o ponto extremo satisfazendo as restrições. De modo geral, um *algoritmo randomizado incremental* primeiro resolve o problema para um subconjunto pequeno dos elementos da entrada e, a cada passo, acrescenta um novo elemento da entrada escolhido aleatoriamente, atualizando a solução. Quando todos os elementos da entrada tiverem sido acrescentados, tem-se a solução do problema.

Sejam $s_1, \ldots, s_n$ os n semiplanos da entrada do problema, onde s_1, s_2 são um par de semiplanos que limitam o problema. Definimos $S_i = \{s_1, \ldots, s_i\}$ e p_i como o ponto mais à direita da interseção de semiplanos de S_i. Iniciamos o algoritmo determinando o ponto p_2. O ponto p_2 pode ser determinado resolvendo o sistema linear formado pelas retas que definem a borda de s_1 e s_2. Para calcularmos p_n, que é a solução do nosso problema, o algoritmo procede calculando p_i incrementalmente, para i de 3 até n. Existem dois casos que podem ocorrer: ou bem $p_{i-1} \in s_i$, ou $p_{i-1} \notin s_i$. Analisamos estes dois casos separadamente.

Note que a região viável definida por S_i está contida na região viável definida por S_{i-1}, pois a região viável de S_i é a interseção de s_i com a região viável de S_{i-1}. Conseqüentemente, se $p_{i-1} \in s_i$, então $p_i = p_{i-1}$. Neste caso, p_i pode ser computado em tempo $O(1)$.

Caso $p_{i-1} \notin s_i$, precisamos examinar os semiplanos de S_i para determinar o valor de p_i. Fica como exercício provar que, se o problema é viável, então p_i está na borda de s_i. No caso de duas variáveis, é fácil determinar o valor de p_i, em tempo $O(i)$, examinando a interseção de cada semiplano de S_{i-1} com a reta formada pela borda de s_i. É possível que, neste procedimento, não encontremos nenhum ponto viável. Se isso ocorrer, podemos afirmar que o problema é inviável. Vale notar que, no caso de d variáveis, como a solução se encontra no subespaço s_i, é necessário resolver recursivamente um problema de programação linear com $d - 1$ variáveis.

Assim, para acrescentarmos um semiplano a um problema com i semiplanos, a complexidade de tempo é $O(i)$ no pior caso. Para acrescentarmos n semiplanos, um a um, começando com 2 semiplanos, a

Entrada:
    v: Vetor com n elementos a serem permutados aleatoriamente.
Saída:
    O vetor v permutado aleatoriamente.
Observações:
    rand(n): Número aleatório distribuído uniformemente de 0 a $n-1$.

```
permutaçãoAleatória(v, n):
    para i decrescendo de n − 1 até 1
        troca v[i] com v[rand(i)]
```

Figura 4.3: Algoritmo que permuta aleatoriamente um vetor.

complexidade de tempo é

$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2).$$

Desejamos reduzir o valor esperado da complexidade de tempo. Para isto, permutamos aleatoriamente os semiplanos de s_3 a s_n. O algoritmo para permutar aleatoriamente um vetor de n elementos em tempo $O(n)$ está descrito na figura 4.3. O pseudo-código do algoritmo de programação linear encontra-se na figura 4.4.

Podemos escrever o valor esperado da complexidade de tempo como

$$\mathbf{E}[T(n)] = \sum_{i=3}^{n} (q_i O(i) + (1 - q_i) O(1)),$$

onde q_i é a probabilidade de $p_{i-1} \notin s_i$. Para determinarmos $\mathbf{E}[T(n)]$, precisamos calcular um limite superior para o valor de q_i. Este limite superior deve depender apenas da permutação aleatória dos semiplanos, e não da entrada do problema.

A técnica *análise de trás para frente* é freqüentemente utilizada para analisar algoritmos incrementais randomizados. O algoritmo incremental descrito parte de uma solução inicial e segue acrescentando um semiplano por vez. Podemos imaginar esse processo de trás para frente, partindo da solução do problema e removendo um semiplano

```
Entrada:
    S: Conjunto de n semiplanos.
Saída:
    p: Ponto extremo direito na interseção dos semiplanos de S.

progLin(S):
    determinar 2 semiplanos s_1, s_2 que limitem o problema
    se s_1, s_2 não existem:
        retorne "problema ilimitado"
    p ← interseção das bordas de s_1 e s_2
    fazer s_3, ..., s_n uma permutação aleatória de S \ {s_1, s_2}.

    para i de 3 até n:
        se p ∉ s_i:
            p ← ponto extremo direito na borda de s_i e
                contido na interseção de s_1, ..., s_{i-1}
            se p não existe:
                retorne "problema inviável"
    retorne p
```

Figura 4.4: Algoritmo que resolve o problema de programação linear.

por vez, até chegar na solução inicial. A cada passo, removemos um semiplano aleatório dentre $i-2$ semiplanos, pois 2 semiplanos fazem parte da solução inicial. O valor de q_i é a probabilidade de removermos um dos 2 semiplanos que definem p_i. Então concluímos que $q_i \leq 2/(i-2) = O(1/i)$. Assim, temos

$$\mathbf{E}[T(n)] = \sum_{i=3}^{n} (O(1/i)O(i) + O(1)O(1)) = \sum_{i=3}^{n} O(1) = O(n).$$

4.2 Funções hash

Funções *hash*, também chamadas de funções *de dispersão*, não são um tópico particularmente geométrico. Entretanto, elas encontram diversas aplicações dentro de geometria computacional, assim como nas áreas mais diversas da computação, de linguagens de programação a

teoria da complexidade. Dados um parâmetro inteiro m e um conjunto de números inteiros C, uma *função hash* é uma função h que mapeia os elementos de C em números inteiros de 0 a $m-1$. Chamamos os elementos de C de *chaves*. Desejamos escolher aleatoriamente uma função hash de modo que, se x e y são duas chaves distintas, então $h(x) \neq h(y)$ com alta probabilidade (no caso, pelo menos $1 - 1/m$).

Por exemplo, considere o conjunto de chaves $C = \{7, 92, 1092\}$ e o parâmetro $m = 3$. Uma excelente função hash seria $h(x) = x \bmod 3$, pois $h(7) = 1$, $h(92) = 2$, $h(1092) = 0$, de modo que $h(x) \neq h(y)$ sempre que $x \neq y$. Porém, a função $h(x) = x \bmod 3$ funciona extremamente mal para certos conjuntos de chaves, como por exemplo, $C = \{3, 33, 129\}$. Neste caso, temos $h(x) = 0$ para todo $x \in C$. A utilização de números aleatórios permite construir funções hash que funcionem bem, no caso esperado, para qualquer conjunto de chaves C, sem nem mesmo necessitarmos examinar o conjunto C.

Para construirmos uma função hash, primeiro determinamos um número primo p tal que $p > x$ para todo $x \in C$. A obtenção de tal número primo não é tarefa trivial, porém p pode ser obtido eficientemente usando as técnicas descritas no capítulo 3, juntamente com o postulado de Bertrand, que diz que, para todo $x > 3$, existe um número primo p tal que $x \leq p \leq 2x - 2$. Após calcularmos p, obtemos dois números inteiros aleatórios $a \in [1, p-1]$ e $b \in [0, p-1]$. Definimos a função hash como

$$h(x) = \text{res}(\text{res}(ax + b, p), m),$$

onde $\text{res}(x, y)$ representa, como na seção 3.1, o resto da divisão de x por y. Para essa função hash, temos o seguinte resultado:

Teorema 4.2.1. *Dadas duas chaves distintas $x, y \in C$, a probabilidade de $h(x) = h(y)$ é no máximo $1/m$. A probabilidade é obtida em função dos valores aleatórios a e b, independente do valor de x e y.*

Demonstração. Nesta demonstração usamos notação e propriedades estabelecidas no capítulo 3. Definimos $H(x) = \overline{a}\ \overline{x} + \overline{b}$ em $\mathbb{Z}_p$. Considere duas chaves $\overline{c_1}, \overline{c_2} \in \mathbb{Z}_p$. Temos que

$$H(c_1) = \overline{a}\ \overline{c_1} + \overline{b} \quad \text{e} \quad H(c_2) = \overline{a}\ \overline{c_2} + \overline{b}.$$

Primeiro, mostramos que $H(c_1) = H(c_2)$ se e só se $\overline{c_1} = \overline{c_2}$. Partindo de $H(c_1) = H(c_2)$, por definição temos que $\overline{a}\ \overline{c_1} + \overline{b} = \overline{a}\ \overline{c_2} + \overline{b}$. Pela existência de inverso aditivo em $\mathbb{Z}_p$, temos que $\overline{a}\ \overline{c_1} = \overline{a}\ \overline{c_2}$. Como $a \neq 0$ e p é primo, $\overline{a}$ possui inverso multiplicativo em $\mathbb{Z}_p$. Então concluímos que $\overline{c_1} = \overline{c_2}$.

O próximo passo é mostrarmos que podemos determinar $\overline{a}$ e $\overline{b}$ em função de $\overline{c_1}$, $\overline{c_2}$, $H(c_1)$ e $H(c_2)$. Subtraindo $\overline{a}\ \overline{c_1}$ de $H(c_1)$ temos que

$$\overline{b} = H(c_1) - \overline{a}\ \overline{c_1}.$$

Subtraindo $H(c_1) - H(c_2)$ temos

$$H(c_1) - H(c_2) = \overline{a}\ (\overline{c_1 - c_2}).$$

Como $\overline{c_1} \neq \overline{c_2}$, então $\overline{c_1 - c_2}$ possui inverso multiplicativo em $\mathbb{Z}_p$. Denotamos o inverso multiplicativo de $\overline{c_1 - c_2}$ por $(\overline{c_1 - c_2})^{-1}$. Segue que

$$\overline{a} = (\overline{c_1 - c_2})^{-1}(H(c_1) - H(c_2)).$$

Note que existem $p(p-1)$ atribuições possíveis para o par $(\overline{a}, \overline{b})$, e existem também $p(p-1)$ pares $(H(c_1), H(c_2))$ com $H(c_1) \neq H(c_2)$. Sendo assim, existe uma correspondência 1 para 1 entre $(\overline{a}, \overline{b})$ e $(H(c_1), H(c_2))$. Como $(\overline{a}, \overline{b})$ é escolhido aleatória e uniformemente em $(\mathbb{Z}_p^*, \mathbb{Z}_p)$, o par $(H(c_1), H(c_2))$ também é escolhido aleatória e uniformemente dentre pares de valores distintos em $\mathbb{Z}_p$.

A partir daqui, consideramos $H(c_1)$ e $H(c_2)$ como inteiros de 0 a $p-1$, e não mais como elementos de $\mathbb{Z}_p$. Temos que $h(c_1) = \text{res}(H(c_1), m)$ e $h(c_2) = \text{res}(H(c_2), m)$. Queremos determinar a probabilidade $\mathbf{Pr}[h(c_1) = h(c_2)]$ e sabemos que $H(c_1) \neq H(c_2)$.

Se fixarmos o valor de $H(c_1)$, existem no máximo $\lceil p/m \rceil - 1$ valores possíveis de $H(c_2)$ com $H(c_2) \neq H(c_1)$ e $h(c_1) = h(c_2)$. Como há no total $p-1$ valores possíveis para $H(c_2)$, então

$$\mathbf{Pr}[h(c_1) = h(c_2)] \leq \frac{\lceil p/m \rceil - 1}{p-1} \leq \frac{(p-1)/m}{p-1} \leq \frac{1}{m}.$$

□

Corolário 4.2.2. *Considere uma chave $x \in C$, e um conjunto $C' \subseteq C$ com $|C'| \leq m$. O valor esperado do número de chaves $y \in C'$ com $h(x) = h(y)$ é menor que 2, ou seja, $\boldsymbol{E}[|\{y \in C' : h(x) = h(y)\}|] < 2$.*

Demonstração. O valor esperado do número de chaves $y \in C'$ com $h(x) = h(y)$ é a soma das probabilidades de $h(x) = h(y)$ para as chaves $y \in C'$. Caso $x \in C'$, usando o teorema 4.2.1,

$$\mathbf{E}[|\{y \in C' : h(x) = h(y)\}|] \leq 1 + \sum_{i=1}^{|C'|-1} \frac{1}{m} = 1 + \frac{|C'| - 1}{m} < 2.$$

Caso $x \notin C'$, e usando ainda o teorema 4.2.1,

$$\mathbf{E}[|\{y \in C' : h(x) = h(y)\}|] \leq \sum_{i=1}^{|C'|} \frac{1}{m} = \frac{|C'|}{m} \leq 1.$$

□

4.3 Par de pontos mais próximos

Dado um conjunto P contendo n pontos de um espaço Euclidiano d-dimensional, é natural perguntar qual o *par de pontos mais próximos*, isto é, quais são os dois pontos distintos $p, q \in P$ que minimizam a distância $|p - q|$. Um algoritmo trivial para este problema tem complexidade de tempo $O(n^2)$, simplesmente calculando as distâncias entre todos os $n(n-1)/2$ pares de pontos e escolhendo o par que determina a distância mínima. Entretanto, existem algoritmos mais eficientes. Por bastante tempo, o problema foi considerado completamente solucionado, pois há diversos algoritmos determinísticos com tempo $O(n \log n)$ e existe um limite inferior de $\Omega(n \log n)$ para o problema, mesmo no caso unidimensional. O limite inferior se refere apenas a algoritmos determinísticos e restritos a comparar resultados de operações algébricas.

O *piso* de um número real x é denotado por $\lfloor x \rfloor$ e é definido como o maior inteiro i tal que $i \leq x$. O piso não é uma operação algébrica. Surpreendentemente, usando randomização e computação de pisos, é possível resolver o problema em tempo $O(n)$. Usando pisos, mas sem usar randomização, é possível resolver o problema em tempo $O(n \log \log n)$. Descrevemos um algoritmo randomizado que resolve o problema em tempo $O(n)$. O algoritmo pode ser facilmente

generalizado para espaços d-dimensionais, mas nos restringimos ao caso bidimensional.

O algoritmo usa randomização de duas maneiras. Primeiro, usa randomização explicitamente na escolha do ponto a ser amostrado a cada iteração. Segundo, usa randomização implicitamente através da utilização de funções hash (vide seção 4.2). Iniciamos com algumas definições que facilitam a descrição do algoritmo. O *vizinho mais próximo* de um ponto p (com respeito ao conjunto P), denotado por $vmp(p)$, é o ponto $p' \in P \setminus \{p\}$ que minimiza $|p - p'|$. Definimos $f(p) = |p - vmp(p)|$, ou seja, $f(p)$ é a distância do ponto p ao seu vizinho mais próximo.

A idéia do algoritmo é, a cada iteração, escolher um ponto p aleatoriamente e remover de P todo ponto q com $f(q) \geq f(p)$. Este procedimento é repetido até todos os pontos de P serem removidos. Note que $f(p)$ limita superiormente a distância entre o par de pontos mais próximos. Conseqüentemente, os pontos removidos não podem ser um dos pontos do par de pontos mais próximos, a não ser que p seja um dos pontos do par de pontos mais próximos. Quando todos os pontos foram removidos, sabe-se que p e seu vizinho mais próximo são de fato o par de pontos mais próximos. O pseudo-código do algoritmo encontra-se na figura 4.5.

Entretanto, calcular $f(q)$ leva tempo $O(n)$, pois determinar o vizinho mais próximo de um ponto q exige examinar todos os demais pontos de P. Assim, a primeira chamada da função removerPontos leva tempo $\Theta(n^2)$ ao calcular $f(q)$ para todo $q \in P$. Como a nossa meta é determinar o par de pontos mais próximos em tempo $O(n)$, precisamos acelerar a função removerPontos. Temos que remover do conjunto P todos os pontos q com $f(q) \geq f(p)$, sem calcularmos $f(q)$ individualmente para cada ponto $q \in P$.

Podemos, antes de iniciar o algoritmo, transladar e escalar os pontos sem alterar o par de pontos mais próximos. Portanto, assumimos, sem perda de generalidade, que os pontos de P estão contidos no *quadrado unitário*, ou seja, $P \subset [0, 1]^2$. Imagine um quadriculado dividindo o quadrado unitário em *células* de diâmetro $f(p)/2$ (e lado $f(p)/2\sqrt{2}$). Chamamos a célula que contém um ponto q de $c(q)$, e definimos a *adjacência* de uma célula $c(q)$ como a região formada por $c(q)$ e as no máximo 8 células no seu entorno (figura 4.6). Note que, se $f(q) \geq f(p)$, então a adjacência de $c(q)$ contém somente o ponto q.

```
Entrada:
    P: Conjunto de n pontos.
Saída:
    p, p': Par de pontos mais próximos de P.

pontosMaisPróximos(P):
    enquanto P ≠ ∅:
        p ← ponto de P escolhido aleatoriamente
        p' ← vmp(p)
        P ← removerPontos(P,f(p))
    retorne p, p'

removerPontos(P,fp):
        para cada q ∈ P:
            se f(q) ≥ fp:
                P ← P \ {q}
```

Figura 4.5: Primeira versão do algoritmo que determina o par de pontos mais próximos.

Por outro lado, a adjacência de $c(q)$ conter somente o ponto q não significa que $f(q) \geq f(p)$, mas sim que $f(q) \geq f(p)/2\sqrt{2}$. Para removermos eficientemente todo ponto q tal que a adjacência de $c(q)$ contém somente q, usamos uma função hash e a função piso.

Definimos uma função hash h com parâmetro $m = n$ e conjunto de chaves sendo o conjunto de todas as células do quadriculado, representadas por números inteiros como definido a seguir. A função piso é importante porque, para representar a célula de um ponto $q = (q_x, q_y)$ como um número inteiro, fazemos $c(q) = \lfloor q_x/(f(p)/2\sqrt{2}) \rfloor + k \lfloor q_y/(f(p)/2\sqrt{2}) \rfloor$, onde $k = 1 + \lfloor 1/(f(p)/2\sqrt{2}) \rfloor$.

Criamos então um vetor v com n posições inicializadas com um conjunto vazio. O vetor v é definido como um vetor de coleções de conjuntos de pontos. Associamos cada ponto $q \in P$ a um dos conjuntos em $v[h(c(q))]$. Desejamos que dois pontos pertençam ao mesmo conjunto se e só se pertencerem à mesma célula, mas duas células distintas podem ter o mesmo valor da função hash. Esta é razão de associarmos $v[h(c(q))]$ a uma *coleção* de conjuntos, sendo um

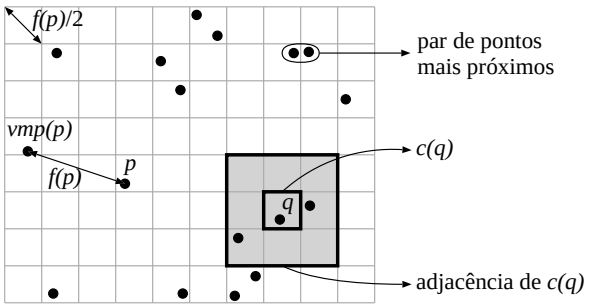


Figura 4.6: Divisão do plano por um quadriculado, com adjacência da célula do ponto q em cinza.

conjunto (de pontos) para cada célula distinta. Deste modo, a posição $v[h(c(q))]$ armazena as células cuja função hash tem valor $h(c(q))$, e um dos conjuntos de $v[h(c(q))]$ contém os pontos da célula $c(q)$.

Usando o vetor v, podemos remover todo ponto q tal que a adjacência de $c(q)$ contém somente q. Para isso, examinamos um elemento do conjunto de pontos — de modo a determinar a célula correspondente — e o número de pontos no conjunto — de modo a determinar se a célula contém algum ponto que não seja q —, conforme o pseudo-código da figura 4.7.

Infelizmente, a função removerPontos remove pontos q com $f(p)/2\sqrt{2} \leq f(q) \leq f(p)$. Deste modo, nosso algoritmo não encontra necessariamente o par de pontos mais próximos, mas sim uma *aproximação* do par de pontos mais próximos. Mais especificamente, o algoritmo encontra um par de pontos p, p' cuja distância $|p - p'|$ é no máximo $2\sqrt{2}$ vezes a distância entre o par de pontos mais próximos. Antes de mostrarmos como obter uma solução exata para o problema, a partir da solução aproximada, analisamos a complexidade de tempo do algoritmo.

Primeiro, analisamos a complexidade da função removerPontos. Para mostrar que a função leva tempo $O(n)$, precisamos mostrar que a condição se do loop mais interno é avaliada $O(n)$ vezes. O número de conjuntos $C \in v[h(x)]$ é igual ao número de células y com

```
removerPontos(P,fp):
    P' ← {}
    v[0...(n−1)] ← {}
    k ← 1 + ⌊1/(fp/2√2)⌋
    seja c(q) = ⌊q_x/(fp/2√2)⌋ + k ⌊q_y/(fp/2√2)⌋
    para cada q ∈ P:
        para cada conjunto C ∈ v[h(c(q))]
            se c(C[0]) = c(q)
                C = C ∪ {q}
        se q não foi adicionado a nenhum conjunto C
            v[h(c(q))] = v[h(c(q))] ∪ {{q}}
    para cada q ∈ P:
        remover ← 1
        para cada célula x na adjacência de c(q):
            para cada conjunto C ∈ v[h(x)]
                se c(C[0]) = x e (c(C[0]) ≠ c(q) ou |C| > 1)
                    remover ← 0
        se remover ≠ 1:
            P' ← P' ∪ {q}
    retorne P'
```

Figura 4.7: Função que remove todos os pontos que não possuem nenhum outro ponto na adjacência de sua célula em tempo $O(n)$.

$h(y) = h(x)$. Pelo corolário 4.2.2, o valor esperado do número de células y com $h(y) = h(x)$ é no máximo 2. O número de células na adjacência de uma célula qualquer é no máximo 9. Portanto, a condição se do loop mais interno é avaliada no máximo 18 vezes por ponto de P e, conseqüentemente, a função removerPontos leva tempo esperado $O(n)$.

Desejamos analisar a complexidade do algoritmo da figura 4.5 usando a função removerPontos. Note que, a cada iteração, o conjunto P pode ser ordenado segundo o valor de $f(p)$ para cada $p \in P$. Escolhemos um ponto p aleatoriamente. Portanto, no caso esperado, pelo menos metade dos pontos $q \in P$ tem $f(q) \geq f(p)$, sendo então removidos. O algoritmo termina quando não há mais pontos em P.

Sendo assim, limitamos o valor esperado da complexidade de tempo usando a linearidade da esperança:

$$\mathbf{E}[T(n)] \leq \sum_{i=0}^{\infty} O(n/2^i) = O(n).$$

Voltamos agora ao problema do algoritmo analisado não determinar o par de pontos mais próximos, mas sim uma aproximação do par de pontos mais próximos. Para obtermos o par de pontos mais próximos a partir da aproximação, usamos um quadriculado e função hash novamente. Seja a a distância entre o par de pontos retornada pelo algoritmo aproximado. Criamos um quadriculado com células de lado a e usamos a função hash para organizar os pontos em um vetor, do mesmo modo que feito anteriormente. Para cada ponto q, determinamos o ponto mais próximo de q na adjacência da célula que contém q, se houver. Note que, como a é um limite superior para a distância entre o par de pontos mais próximos, e as células têm lado a, sabemos que o par de pontos mais próximos encontra-se em células adjacentes.

Para analisarmos a complexidade do algoritmo, precisamos saber o número máximo de pontos que podem pertencer a uma mesma célula do quadriculado. Sabemos que não há dois pontos mais próximos que $a/2\sqrt{2}$, já que a é uma aproximação de fator $2\sqrt{2}$ da distância do par de pontos mais próximos. Dividimos uma célula do quadriculado em 16 sub-células de diâmetro $a/2\sqrt{2}$ (figura 4.8). Como não há dois pontos mais próximos que $a/2\sqrt{2}$, cada sub-célula pode ter no máximo 1 ponto, e uma célula pode ter no máximo 16 pontos. Usando o corolário 4.2.2 de maneira análoga à anterior, mostramos que esta rotina leva tempo $O(n)$ — e obtemos o par de pontos mais próximos em tempo $O(n)$.

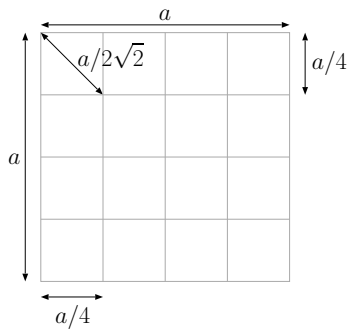


Figura 4.8: Divisão de uma célula em 16 sub-células com no máximo um ponto por sub-célula.

4.4 Exercícios

1. Escreva algoritmos randomizados incrementais, com complexidade de tempo $O(n)$, para os seguintes problemas:

   (a) Dado um conjunto de n pontos P, e um ponto $p \in P$, determinar o menor círculo C que contém todos os pontos de P, tal que p pertence à borda de C.

   (b) Dado um conjunto de n pontos P, determinar o menor círculo C que contém todos os pontos de P.

2. Dados dois conjuntos de pontos P_1, P_2, separados por uma reta vertical, chamamos de *ponte* a reta r que contém dois pontos $p_1 \in P_1$ e $p_2 \in P_2$, de modo que todos os demais pontos de $P_1 \cup P_2$ estão abaixo de r. Escreva um algoritmo randomizado incremental que determina a ponte em tempo $O(|P_1| + |P_2|)$.

3. Dado um conjunto de n chaves inteiras C, uma função hash h é dita *perfeita* quando $h(x) \neq h(y)$ para todo par de chaves distintas x, y. Neste exercício, você deve escrever e analisar um algoritmo que, dado um conjunto C, obtenha uma função hash perfeita $h : C \to \{0, \ldots, m-1\}$. A função h deve poder ser

avaliada em tempo $O(1)$. Forneça inicialmente um algoritmo de Monte Carlo e, em seguida, converta este algoritmo em um algoritmo de Las Vegas. Considere que o parâmetro m é:

(a) $m = O(n^2)$;

(b) $m = O(n)$. (*Dica*: use dois níveis de funções hash.)

4. Escreva um algoritmo que, dados um conjunto de pontos P e um número real r, liste todos os pares de pontos p_1, p_2 tal que $|p_1 - p_2| \leq r$. O seu algoritmo deve ter complexidade de tempo $O(n + k)$, onde k é o número de pontos listados.

5. Modifique o algoritmo que determina o par de pontos mais próximos para funcionar em espaços d-dimensionais, onde d é constante.

4.5 Notas bibliográficas

Diversos livros estudam algoritmos randomizados em geometria computacional. De Berg, van Kreveld, Overmars e Schwarzkopf [15] fazem uma excelente introdução ao estudo de geometria computacional, cobrindo diversos algoritmos randomizados, entre eles o algoritmo de programação linear apresentado aqui. Motwani e Raghavan [41] analisam um grande número de algoritmos randomizados de programação linear, geometria computacional e funções hash. Mulmuley [42] introduz problemas fundamentais de geometria computacional usando algoritmos randomizados. Em língua portuguesa, Figueiredo e Carvalho [22] e também Rezende e Stolfi [46] escreveram ótimos textos introdutórios de geometria computacional.

O algoritmo de programação linear apresentado aqui foi descoberto por Seidel [49]. Caso consideremos a dimensão d como uma variável assintótica, a complexidade de tempo é $O(d!\, n)$. Outros algoritmos randomizados são mais eficientes em função de d, tendo complexidades como $O(d^2 n + e^{\sqrt{d \ln d}})$ [37].

Funções hash vêm sendo estudadas na ciência da computação desde a década de 50 e costumam ser apresentadas em livros introdutórios de algoritmos [11, 32]. Knuth [33] cobre funções hash, porém considerando a distribuição probabilística das chaves. Funções hash

que garantem resultados probabilísticos independentemente da distribuição das chaves foram introduzidas por Carter e Wegman [4].

O algoritmo clássico para o par de pontos mais próximos leva tempo $O(n \log n)$ e é baseado no paradigma de divisão e conquista [43]. A primeira solução randomizada foi descoberta por Rabin [44] e leva tempo $O(n)$. Um algoritmo determinístico que usa a função piso e leva tempo $O(n \log \log n)$ foi descoberto por Fortune e Hopcroft [23]. Khuller e Matias [31] descobriram o algoritmo descrito aqui.

Grande parte dos algoritmos randomizados para geometria computacional possuem versões de-randomizadas com complexidades similares. Chazelle e Friedman [6] introduziram diversas técnicas para de-randomizar algoritmos geométricos. Matoušek [36] compilou diversos resultados de de-randomização em geometria computacional. O algoritmo randomizado incremental para o fecho convexo em tempo ótimo $O(n^{\lfloor (d-1)/2 \rfloor} + n \log n)$ foi descoberto por Clarkson e Shor [7] e de-randomizado por Chazelle [5].

Capítulo 5

O Método Probabilístico

Algoritmos randomizados, além de poderem oferecer bom custo-benefício na solução de problemas combinatórios, introduziram uma poderosa ferramenta para provas de existência: o chamado *método probabilístico*. A idéia é utilizarmos algoritmos randomizados e as probabilidades a eles associadas para provar (com certeza!) a existência de determinada propriedade ou objeto. A seção 5.1 discute o método da probabilidade positiva e o método da esperança para provas de existência, exemplificando-os com problemas de teoria dos grafos.

Em alguns casos, é também possível *de-randomizar* algoritmos randomizados, obtendo algoritmos determinísticos cuja corretude é provada pelo método probabilístico. A seção 5.2 apresenta o método das esperanças condicionais para a obtenção de algoritmos determinísticos a partir de algoritmos randomizados.

5.1 Provas de existência

Imagine-se num país desconhecido sobre cuja moeda você nada sabe. Há um baú contendo moedas locais. Você desconhece totalmente quais sejam os valores das moedas existentes naquele país. Ora, se

um nativo lhe informa que é nula a probabilidade de que uma moeda retirada do baú ao acaso seja uma moeda de, digamos, 3 dinheiros, isso não lhe agrega muita informação quanto aos tipos de moedas emitidas naquele país. Sequer poderia você concluir que *não existem* moedas locais de 3 dinheiros, pois é bem possível que *aquele baú, em particular*, não tenha sido abastecido com moeda alguma daquele valor. Por outro lado, se o nativo lhe informasse ser *positiva* a probabilidade de que uma moeda retirada ao acaso fosse uma moeda de 3 dinheiros, ele estaria lhe informando, não há dúvida, que *existe pelo menos uma* moeda de três dinheiros no sistema monetário daquele país! Note bem que não importa qual seja aquela probabilidade, contanto seja estritamente maior do que zero — *existe* a tal moeda!

Suponha agora que você, naquele mesmo país desconhecido, se interesse pelos salários pagos aos professores de matemática locais. Você toma conhecimento de um estudo que revela que a média salarial da classe dos professores de matemática[1] é de 500 dinheiros. Ora, este dado traz consigo duas informações extras: *existe* algum professor de matemática ganhando 500 dinheiros ou menos; e *existe* algum professor de matemática ganhando 500 dinheiros ou mais.

Esses dois exemplos ilustram as estratégias de prova mais simples baseadas no método probabilístico. Nas notas bibliográficas, indicamos onde encontrar o lema Local de Lovász e o método do segundo momento, que são ferramentas de prova mais sofisticadas.

5.1.1 O método da probabilidade positiva

Seja K_{40} o grafo completo[2] de 40 vértices. Deseja-se saber se é possível 2-colorir suas $\binom{40}{2} = 780$ arestas[3] de forma que não haja nenhuma clique monocromática[4] de tamanho maior ou igual a 8.

[1] O mesmo que o valor esperado, ou esperança, da variável aleatória S definida como o salário de um professor de matemática sorteado aleatória e uniformemente do universo de professores de matemática daquele país.

[2] Um grafo é *completo* se cada um de seus vértices é adjacente a todos os demais vértices do grafo.

[3] Uma *d-coloração* em arestas é uma função do conjunto de arestas de um grafo em um conjunto qualquer de d elementos, ditos "cores".

[4] Uma *clique* de um grafo é um subgrafo completo. Uma clique é monocromática, no contexto da coloração de arestas, se todas as suas arestas possuem a mesma cor.

Entrada:
    G: um grafo.
Saída:
    Uma 2-coloração de G.

coloração(G):
    para cada aresta a do grafo G:
        jogue uma moeda
        se cara, pinte a de azul; se coroa, pinte a de amarelo
    retorne a coloração assim obtida

Figura 5.1: Algoritmo randomizado para 2-colorir um grafo.

Seja o espaço amostral constituído de todas as 2^{780} 2-colorações possíveis para o K_{40}. Se mostrarmos que, ao selecionarmos randomicamente um elemento daquele espaço amostral, é estritamente positiva a probabilidade de que aquele elemento possua a característica desejada, isto é, trate-se de uma 2-coloração em arestas que não contém cliques monocromáticas de tamanho maior ou igual a 8, então teremos provado que tal coloração *existe*.

Voltemos nossa atenção, agora, para o algoritmo randomizado da figura 5.1.

Qual a probabilidade da resposta obtida por uma chamada a coloração(K_{40}) atender ao critério de não conter cliques monocromáticas grandes?

Notemos, em primeiro lugar, que não possuir cliques monocromáticas de tamanho maior ou igual a 8 é exatamente o mesmo que não possuir cliques monocromáticas de tamanho *igual* a 8 (convidamos o leitor a conferir, mentalmente, esta equivalência). Ora, chamemos C_i, $i = 1, \ldots, \binom{40}{8}$, às cliques de tamanho 8 de nosso K_{40} e seja M_i o evento em que as arestas de C_i são todas azuis ou todas amarelas na coloração retornada pelo nosso algoritmo. É facil ver que

$$\mathbf{Pr}[M_i] = \frac{1}{2^{\binom{8}{2}-1}} = 2^{-27}.$$

Estamos interessados em

$$\mathbf{Pr}\left[\bigcap_{i=1}^{\binom{40}{8}} \overline{M_i}\right] = 1 - \mathbf{Pr}\left[\bigcup_{i=1}^{\binom{40}{8}} M_i\right].$$

Pelo limite da união, temos

$$\mathbf{Pr}\left[\bigcup_{i=1}^{\binom{40}{8}} M_i\right] \leq \sum_{i=1}^{\binom{40}{8}} \mathbf{Pr}[M_i] = \frac{\binom{40}{8}}{2^{27}} < 0{,}573.$$

Chegamos, portanto, a

$$\mathbf{Pr}\left[\bigcap_{i=1}^{\binom{40}{8}} \overline{M_i}\right] > 1 - 0{,}573 = 0{,}427 > 0.$$

Conclui-se, assim, que existe uma tal coloração.

O mesmo raciocínio nos permite provar a existência de 2-colorações para o K_{5817} onde não há cliques monocromáticas de tamanho 20 (a probabilidade utilizada na prova pelo método probabilístico é maior que 0,00233, portanto ainda estritamente positiva). Nada poderíamos concluir sobre a existência de 2-colorações para o K_{5818} sem cliques monocromáticas de tamanho 20.

> *Define-se um espaço probabilístico e prova-se positiva a probabilidade de um objeto selecionado aleatoriamente possuir determinada propriedade — conclui-se que existe um objeto com a propriedade em questão.*

5.1.2 O método da esperança

O problema do corte máximo de um grafo é NP-difícil. Um corte é uma partição dos vértices de um grafo em dois conjuntos, onde o número de arestas que liga um vértice de um dos conjuntos a um vértice do outro conjunto constitui o *tamanho* do corte.

Sendo difícil obter o tamanho do corte máximo, pode-se desejar saber se há algum corte grande, ou seja, contendo um número mínimo de arestas, ainda que não seja este número o maior possível.

Entrada:
    G: um grafo.
Saída:
    Um corte de G.

corte(G):
    crie dois conjuntos A e B inicialmente vazios
    para cada vértice v do grafo, jogue uma moeda
        se cara, coloque v em A; se coroa, coloque v em B
    retorne o corte assim obtido

Figura 5.2: Algoritmo randomizado para obter um corte.

Usando o método probabilístico, consegue-se provar que há sempre um corte de tamanho maior ou igual a $m/2$, isto é, à metade do número de arestas do grafo.

Seja o algoritmo randomizado da figura 5.2 para encontrar um corte em um grafo.

Estamos interessados no valor esperado da variável aleatória $C(A, B)$ definida como o tamanho do corte dessa forma obtido.

Criemos um indicador de Bernoulli X_i, para cada aresta a_i do grafo, definido como

$$X_i = \begin{cases} 1, & \text{se } a_i \text{ conecta } A \text{ a } B; \\ 0, & \text{caso contrário.} \end{cases}$$

A esperança de cada X_i é igual a probabilidade de a_i ligar A a B, que é a probabilidade de que os extremos de a_i não tenham sido colocados no mesmo conjunto, isto é, $1/2$.

É fácil ver que

$$\mathbf{E}\left[C(A,B)\right] = \mathbf{E}\left[\sum_{i=1}^{m} X_i\right] = \sum_{i=1}^{m} \mathbf{E}[X_i] = \frac{m}{2}.$$

Portanto, existe corte de tamanho maior ou igual a $m/2$.

Define-se um espaço probabilístico e obtém-se a esperança $\boldsymbol{E}[X] = \mu$ *de uma variável aleatória* X *definida para aquele espaço — conclui-se que são positivas* $\boldsymbol{Pr}[X \leq \mu]$ *e* $\boldsymbol{Pr}[X \geq \mu]$ *e, portanto, existe elemento naquele espaço para o qual* X *tem valor menor ou igual a* μ *e existe elemento naquele espaço para o qual* X *tem valor maior ou igual a* μ*.*

5.2 De-randomização

Na prova de existência de cortes com pelo menos metade do número de arestas de um grafo, e embora não tenhamos atentado para este fato, estivemos diante de um algoritmo de Las Vegas para *localizar* um corte com pelo menos aquele número de arestas. O pseudo-código de tal algoritmo encontra-se na figura 5.3.

Cada iteração do laço principal do algoritmo da figura 5.3 encontra um corte de tamanho maior ou igual a $m/2$ com probabilidade p. O número de iterações até que seja encontrado o primeiro corte de tamanho maior ou igual a $m/2$ é, portanto, uma variável geométrica cuja esperança é 1/p. Calculando p, chegamos facilmente ao tempo esperado de execução de nosso algoritmo de Las Vegas.

Queremos calcular $p = \mathbf{Pr}[C(A,B) \geq \frac{m}{2}]$.

Sabemos que

$$\begin{aligned}\frac{m}{2} &= \mathbf{E}[C(A,B)] \\ &= \sum_{i=0}^{m/2-1} i\ \mathbf{Pr}[C(A,B)=i] + \sum_{i=m/2}^{m} i\ \mathbf{Pr}[C(A,B)=i].\end{aligned}$$

A primeira parcela acima é menor ou igual a

$$(m/2-1)\sum_{i=0}^{m/2-1} \mathbf{Pr}[C(A,B)=i] = (m/2-1)(1-p)$$

e a segunda parcela é menor ou igual a

$$m\sum_{i=m/2}^{m} \mathbf{Pr}[C(A,B)=i] = mp.$$

Entrada:
    G: um grafo.
Saída:
    Um corte de G com pelo menos metade de suas arestas.

corteGrandeLasVegas(G):
    repita:
        crie dois conjuntos A e B inicialmente vazios
        para cada vértice v do grafo, jogue uma moeda
            se cara, coloque v em A; se coroa, coloque v em B
        se o tamanho do corte é maior ou igual a $m/2$, retorne-o
    até que um corte seja retornado

Figura 5.3: Algoritmo randomizado para obter um corte grande.

Portanto,

$$\frac{m}{2} \leq \left(\frac{m}{2} - 1\right)(1-p) + mp,$$

implicando

$$p \geq \frac{1}{1 + \frac{m}{2}}$$

e um valor esperado menor ou igual a $1 + m/2$ para o número de iterações em uma execução do algoritmo.

Veremos, agora, uma maneira de *de-randomizar* o algoritmo para encontrar cortes grandes de forma determinística.

5.2.1 O método das esperanças condicionais

A idéia central do método das esperanças condicionais é quase genial de tão simples: se há um algoritmo que, fazendo escolhas randômicas a cada passo, resulta num valor esperado μ para uma certa variável aleatória X, podemos retirar a aleatoriedade do algoritmo, *ajudando-o*, por assim dizer, a tomar as decisões que *garantam* que o valor atribuído a X não será "pior" (menor ou maior, dependendo do que se deseje) do que μ.

Seja o algoritmo da figura 5.2. O corte retornado, quando os vértices são, um a um, colocados aleatoriamente em A ou B tem *tamanho esperado* $m/2$ — podendo, evidentemente, ser maior ou menor que $m/2$ numa iteração em particular.

Seja, agora, $E_k[X] = \mathbf{E}[X \mid Y_1, Y_2, \ldots, Y_k]$ a esperança de X condicionada ao fato de que os primeiros k vértices *já foram*, àquele momento, posicionados em A ou B (a variável aleatória Y_i assume o valor simbólico "A" ou "B" de acordo com o conjunto em que v_i foi posicionado). Note que $\mathbf{E}[X] = E_0[X] = m/2$.

Pensemos no momento em que o $(k+1)$-ésimo vértice v_{k+1} será colocado em A ou B, segundo o lançamento da moeda. Ora, se pudéssemos abandonar a moeda e escolher *deterministicamente* o conjunto em que v_{k+1} será posicionado, de forma a garantir $E_{k+1}[X] \geq E_k[X]$, então teríamos, por indução em k, um método determinístico que posicionaria todos os n vértices do grafo em A ou B de forma a garantir que o tamanho do corte retornado no final, ou $E_n[X]$, seria maior ou igual a $E_0[X] = m/2$.

Em outras palavras, queremos provar — por indução no número de vértices já posicionados em passos anteriores — que é possível, a cada passo, escolher deterministicamente o conjunto em que cada vértice será posicionado, de forma a obter um corte de tamanho maior ou igual ao valor esperado $m/2$ para o tamanho do corte que seria retornado pelo algoritmo randomizado. A prova por indução tem a seguinte estrutura:

base: $\mathbf{E}[X \mid Y_1 = \text{"A"}] = E_0[X]$, por simetria (o primeiro vértice considerado pode ser colocado em A ou B indistintamente).

passo indutivo: $E_{k+1}[X] \geq E_k[X]$, o que será garantido ao se escolher deterministicamente a posição de v_{k+1}.

conclusão: o corte retornado pelo algoritmo de-randomizado tem tamanho $E_n[X] \geq E_0[X] = m/2$.

A estratégia ditada pelo método das esperanças condicionais nos garantirá o passo indutivo exposto acima, permitindo-nos, portanto, de-randomizar o algoritmo original. Mas *como* consegui-lo? Como fazer todas as escolhas de forma garantidamente *não-pior* do que o faria uma sucessão de lançamentos da moeda?

Entrada:
    G: um grafo.
Saída:
    Um corte de G com pelo menos metade de suas arestas.

corteGrande(G):
    sejam A e B dois conjuntos inicialmente vazios
    posicione v_1 arbitrariamente em A
    para cada vértice $v_i, i = 2, \ldots, n$ faça
        se v_i possui menos vizinhos em A do que em B
            coloque v_i em A
        senão coloque v_i em B
            retorne o corte assim obtido

Figura 5.4: Algoritmo de-randomizado para obter um corte grande.

Ora, se o posicionamento de v_{k+1} é definido pelo lançamento de uma moeda, então

$$\begin{aligned} E_k[X] &= \mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “A”})]\cdot\mathbf{Pr}[Y_{k+1}= \text{ “A”}] \\ &\quad + \mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “B”})]\cdot\mathbf{Pr}[Y_{k+1}= \text{ “B”}] \\ &= \mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “A”})]\cdot\frac{1}{2} \\ &\quad + \mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “B”})]\cdot\frac{1}{2} \\ &= \frac{1}{2}\ \{\mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “A”})] \\ &\qquad +\ \mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “B”})]\}\,. \end{aligned}$$

Dessa forma, o *maior* entre $\mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “A”})]$ e $\mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “B”})]$ é necessariamente maior do que $E_k[X]$, e só o que nosso algoritmo de-randomizado precisará fazer para "não fazer pior do que a moeda" é saber avaliar o maior dentre $\mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “A”})]$ e $\mathbf{E}[X|(Y_1,\ldots,Y_k)\cap(Y_{k+1}= \text{ “B”})]$ para escolher o conjunto A ou B para posicionar v_{k+1}. É fácil ver que a escolha do destino de v_{k+1} que maximiza $E_{k+1}[X]$ é aquele

em que v_{k+1} é colocado no conjunto (A ou B) que possua *menos* vizinhos de v_{k+1}, contribuindo assim com o maior número possível de arestas para o corte (isto é, ligando A a B). Desempates podem ser resolvidos arbitrariamente.

Nosso algoritmo fica, portanto, como mostrado na figura 5.4.

Não há, portanto, em momento algum, qualquer experimento aleatório a ditar-lhe os passos — e o corte retornado tem tamanho, como se provou, maior ou igual a $m/2$.

> *Nem sempre é fácil, ou sequer possível, de-randomizar um algoritmo utilizando o método das esperanças condicionais.*

5.3 Exercícios

1. O problema da *satisfatibilidade* (SAT) foi o primeiro problema sabidamente NP-completo, como provado por Cook [8] em 1971. Trata-se de descobrir se uma expressão booleana — envolvendo apenas variáveis, parênteses e os operandos para conjunção (AND), disjunção (OR) e negação (NOT) — pode ser satisfeita, isto é, se existe alguma atribuição de valor às variáveis booleanas envolvidas que faça com que a expressão inteira tenha valor *VERDADEIRO*. Mesmo a versão do problema em que a expressão é dada na forma normal conjuntiva[5] com 3 literais por cláusula (chamada 3-SAT) é ainda NP-completa, como provado por Karp no seu famoso texto dos 21 problemas [29]. Abaixo um exemplo de entrada para o 3-SAT com 4 cláusulas em 5 variáveis:

$$(x_1 \text{ OR } x_2 \text{ OR } \overline{x_5}) \text{ AND}$$
$$(x_2 \text{ OR } \overline{x_3} \text{ OR } \overline{x_4}) \text{ AND}$$
$$(x_3 \text{ OR } x_4 \text{ OR } x_5) \text{ AND}$$
$$(x_1 \text{ OR } \overline{x_4} \text{ OR } x_5).$$

[5] Expressões booleanas na forma normal conjuntiva apresentam-se como uma conjunção de cláusulas, cada qual uma disjunção de literais. Literais são variáveis booleanas ou suas negações. Indicamos por $\overline{x}$ a negação da variável x, isto é, NOT(x).

Mostre, pelo método probabilístico, que, para toda entrada de 3-SAT com n variáveis e m cláusulas, há sempre alguma atribuição booleana $x \in \{0,1\}^n$ que satisfaz pelo menos $7m/8$ cláusulas.

2. O *número de Ramsey* $R(r,s)$ é o menor inteiro para o qual é verdade que, dado um grafo completo G com $R(r,s)$ vértices ou mais e cujas arestas estão particionadas em dois conjuntos (o conjunto das arestas "amarelas" e o das arestas "azuis", por exemplo), G possui um subgrafo completo de r vértices cujas arestas são todas "amarelas" ou um subgrafo completo de s vértices cujas arestas são todas "azuis".

   (a) Mostre que $R(3,3) = 6$.

   (b) Do exposto na seção 5.1.1, o que se consegue inferir sobre $R(8,8)$?

   (c) Ainda não é conhecido o valor de $R(5,5)$. Use o método probabilístico para provar um limite inferior para este número.

3. (a) Prove que, para todo inteiro n, existe uma 2-coloração das arestas do grafo completo K_n tal que o número total de cópias monocromáticas do K_4 é no máximo $\binom{n}{4}2^{-5}$.

   (b) Dê um algoritmo randomizado polinomial (em n) para encontrar uma tal coloração.

   (c) Mostre como construir um algoritmo determinístico baseado no algoritmo do item anterior, e usando o método das esperanças condicionais.

5.4 Notas bibliográficas

A referência obrigatória para o método probabilístico é o livro de Alon e Spencer [2], além dos capítulos sobre o tema encontrados nos livros de Motwani e Raghavan [41] e Mitzenmacher e Upfal [39]. Em português, uma introdução aos números de Ramsey e ao método probabilístico é encontrada no livro de Moreira e Kohayakawa [40].

A NP-completude do problema do corte máximo foi demonstrada por Garey, Johnson e Stockmeyer [25] em 1976. Referimos o leitor aos textos de Festa, Pardalos, Resende e Ribeiro [20] e de Goemans e Williamson [26] para exemplos de heurísticas randomizadas para o problema.

De-randomização é tema de pesquisa muito recente, mas que já conta com bom material publicado. Um bom *survey* é o de Valentine Kabanets [30]. Recomendamos também o capítulo de Peter Bro Miltersen em [38]. Em português, o livro de Fernandes, Miyazawa, Cerioli e Feofiloff [19] dedica um capítulo ao tema.

Bibliografia

[1] M. Agrawal, N. Kayal e N. Saxena. Primes is in P. *Ann. of Math.*, 160(2), 781–793, 2004.

[2] N. Alon e J. Spencer (com apêndice de Paul Erdős). *The Probabilistic Method*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., New York, 1992.

[3] B. Bollobás. *Random Graphs*. Cambridge University Press, Cambridge, 2001.

[4] J. L. Carter e M. N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2), 143–154, 1979.

[5] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10(4), 377–409, 1993.

[6] B. Chazelle e J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3), 229–249, 1990.

[7] K. L. Clarkson e P. W. Shor. Applications of random sampling in computational geometry II. *Discrete Comput. Geom.*, 4(5), 387–421, 1989.

[8] S. A. Cook. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 151–158, Association for Computing Machinery, 1971.

[9] C. Cooper e A. M. Frieze. On the number of Hamilton cycles in a random graph. *J. Graph Theory*, 13(6), 719–735, 1989.

[10] C. Cooper e A. M. Frieze. Hamilton cycles in random graphs and directed graphs. *Random Structures Algorithms*, 16(4), 369–401, 2000.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[12] S. C. Coutinho. *Números Inteiros e Criptografia RSA*. Série de Computação e Matemática, 2. Instituto de Matemática Pura e Aplicada (IMPA), Sociedade Brasileira de Matemática, Rio de Janeiro, 2000.

[13] S. C. Coutinho. *Primalidade em Tempo Polinomial: Uma Introdução ao Algoritmo AKS*. Coleção Iniciação Científica, 2. Sociedade Brasileira de Matemática, Rio de Janeiro, 2004.

[14] R. Crandall e C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer, New York, 2005.

[15] M. de Berg, M. van Kreveld, M. Overmars e O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, segunda edição, 2000.

[16] B. C. Dean. A simple expected running time analysis for randomized "divide and conquer" algorithms. *Discrete Appl. Math.*, 154(1), 1–5, 2006.

[17] J. Edmonds e K. Pruhs. Balanced allocations of cake. *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, 623–634, IEEE Computer Society, 2006.

[18] P. Erdős e A. Rényi. On random graphs. *Publ. Math. Debrecen*, 6, 290–297, 1959.

[19] C. G. Fernandes, F. K. Miyazawa, M. R. Cerioli e P. Feofiloff. *Uma Introdução Sucinta a Algoritmos de Aproximação*. Publicações Matemáticas do IMPA. 23° Colóquio Brasileiro de Matemática. IMPA, Rio de Janeiro, 2001.

[20] P. Festa, P. Pardalos, M. Resende e C. Ribeiro. Randomized heuristics for the max-cut problem. *Optim. Methods Softw.*, 17(6), 1033–1058, 2002.

[21] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator e N. E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4), 685–699, 1991.

[22] L. H. de Figueiredo e P. C. P. Carvalho. *Introdução à Geometria Computacional*. 18° Colóquio Brasileiro de Matemática, IMPA, Rio de Janeiro, 1991.

[23] S. Fortune e J. E. Hopcroft. A note on Rabin's nearest neighbor algorithm. *Inform. Process. Lett.*, 8(1), 20–23, 1979.

[24] D. Gale e L. S. Shapley. College admissions and the stability of marriage. *Amer. Math. Monthly*, 69(1), 9–15, 1962.

[25] M. R. Garey, D. S. Johnson e L. Stockmeyer. Some simplified NP-complete graph problems. *Theoret. Comput. Sci.*, 1(3), 237–267, 1976.

[26] M. X. Goemans e D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. Assoc. Comput. Mach.*, 42(6), 1115–1145, 1995.

[27] A. Granville. It is easy do determine whether a given integer is prime. *Bull. Amer. Math. Soc.*, 42(1), 3–38, 2005.

[28] D. Gusfield e R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.

[29] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 85–103. Plenum Press, New York, 1972.

[30] V. Kabanets. Derandomization: a brief overview. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 76, 88–103, 2002.

[31] S. Khuller e Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Inform. and Comput.*, 118(1), 34–37, 1995.

[32] J. Kleinberg e E. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[33] D. E. Knuth. *The Art of Computer Programming vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

[34] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer, New York, 1994.

[35] C. A. J. Martinhon. *Algoritmos Randômicos em Otimização Combinatória*. SOBRAPO, Rio de Janeiro, 2002.

[36] J. Matoušek. Derandomization in computational geometry. *J. Algorithms*, 20(3), 545–580, 1996.

[37] J. Matoušek, M. Sharir e E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16(4-5), 498–516, 1996.

[38] P. B. Miltersen. Derandomizing complexity classes. *Handbook of Randomized Computing Vol. II*. S. Rajasekaran, P. M. Pardalos, J. H. Reif e J. D. Rolim (Eds.), Kluwer Academic Publishers, Dordrecht, 2001.

[39] M. Mitzenmacher e E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Cambridge, 2005.

[40] C. G. T. A. Moreira e Y. Kohayakawa. *Tópicos em Combinatória Contemporânea*. Publicações Matemáticas do IMPA. 23° Colóquio Brasileiro de Matemática. IMPA, Rio de Janeiro, 2001.

[41] R. Motwani e P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[42] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, 1994.

[43] F. P. Preparata e M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[44] M. O. Rabin. Probabilistic algorithms. *Algorithms and complexity (Proc. Sympos., Carnegie-Mellon Univ., Pittsburgh, Pa., 1976)*, 21–39, Academic Press, New York, 1976.

[45] M. O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12, 128–138, 1980.

[46] P. J. de Rezende e J. Stolfi. *Fundamentos de Geometria Computacional*. IX Escola de Computação, Recife, PE, 1994.

[47] P. Ribenboim. *The Little Book of Big Primes*. Springer, New York, 1991.

[48] R. Rivest, A. Shamir e L. Adleman. A method for obtaining digital signatures and public-key cryptosistems. *Comm. ACM*, 21(2), 120–126, 1978.

[49] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6(5), 423–434, 1991.

[50] P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.*, 41(2), 303–332, 1999.

[51] V. Strumpen e A. Krishnamurthy. A collision model for randomized routing in fat-tree networks. *J. Parallel Distrib. Comput.*, 65(9), 1007–1021, 2005.