Probabilidade com Algoritmos

e vice-versa

Uma introdução à probabilidade discreta e

aos algoritmos probabilísticos

— Jair Donadelli —

última modificação 12/2/2020

[*Rosencrantz and Guildenstern are riding horses down a path - they pause*]
R: Umm, uh...
[*Guildenstern rides away, and Rosencrantz follows. Rosencrantz spots a gold coin on the ground*]
R: Whoa - whoa, whoa.
[*Gets off horse and starts flipping the coin*] **R**: Hmmm. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads. Heads.
[*Guildenstern grabs the coin, checks both sides, then tosses it back to Rosencrantz*]
R: Heads.
[*Guildenstern pulls a coin out of his own pocket and flips it*]
R: Bet? Heads I win?
[*Guildenstern looks at coin and tosses it to Rosencrantz*]
R: Again? Heads.

[...]

R: Heads
G: A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.
R: Heads
G: Consider. One, probability is a factor which operates within natural forces. Two, probability is not operating as a factor. Three, we are now held within um...sub or supernatural forces. Discuss!
R: What?
[...]
R: Heads, getting a bit of a bore, isn't it?

[...]

R: 78 in a row. A new record, I imagine.
G: Is that what you imagine? A new record?
R: Well...
G: No questions? Not a flicker of doubt?
R: I could be wrong.
G: No fear?
R: Fear?
G: Fear!
R: Seventy nine.

[...]

G: I don't suppose either of us was more than a couple of gold pieces up or down. I hope that doesn't sound surprising because its very unsurprisingness is something I am trying to keep hold of. The equanimity of your average tosser of coins depends upon a law, or rather a tendency, or let us say a probability, or at any rate a mathematically calculable chance, which ensures that he will not upset himself by losing too much nor upset his opponent by winning too often. This made for a kind of harmony and a kind of confidence. It related the fortuitous and the ordained into a reassuring union which we recognized as nature. The sun came up about as often as it went down, in the long run, and a coin showed heads about as often as it showed tails.

Tom Stoppard, *Rosenkrantz and Guildenstern are dead* (1996).

2 | ALGORITMOS ALEATORIZADOS

Uma definição precisa de algoritmo é importante para entendermos os processos computacionais, conhecer seus limites e estabelecer sua eficiência na resolução de problemas, discutiremos mais dos aspectos formais no capítulo 5. Neste capítulo ficaremos com uma discussão informal, apresentaremos exemplos clássicos de algoritmos que usam bits aleatórios em suas computações e como limitamos a chance de erro e seu tempo de execução. Veremos alguns algoritmos da Teoria do Números e apresentamos uma introdução a computação em corpos finitos, temas que serão úteis em outros capítulos. Na última seção apresentamos um problema computacional cuja solução probabilística envolve um modelo contínuo.

2.1 ANÁLISE DE ALGORITMOS

Um algoritmo define sem ambiguidade uma sequência de passos para resolver um problema computacional. Um *problema computacional* é caracterizado por um conjunto de *instâncias* (ou entradas), um conjunto de *respostas* e uma *relação* que associa instâncias a respostas. Por exemplo, o problema "Multiplicar dois inteiros positivos" tem como instâncias pares (n, m) de inteiros positivos e como respostas inteiros positivos. A relação que se quer computar é definida pelos pares $((n, m), z)$ de instâncias e respostas tais que $n \cdot m = z$. Notemos que entre instâncias e respostas temos uma relação e não uma função pois é possível que uma instância esteja associada a mais de uma resposta. Por exemplo, se as instância são fórmulas da lógica de proposições e as respostas são valorações das variáveis com verdadeiro ou falso e que tornam a fórmula verdadeira, então a instância x_1 *ou* x_2 tem três respostas possíveis.

Os algoritmos são, geralmente, descritos no que costumamos chamar de *pseudocódigo*, uma mistura de algumas palavras-chave em português com sentenças construídas como em uma linguagem de programação estruturada como a linguagem C.

Um algoritmo executado sobre qualquer instância do problema produz uma resposta que deve estar correta, isto é, o par instância/resposta deve fazer parte da relação do problema. Além da correção do algoritmo, devemos conhecer o comportamento do algoritmo com respeito ao consumo de recursos para resolver o problema.

Alguns recursos para os quais podemos querer estimar o consumo por um algoritmo são *espaço*, *tempo*, *comunicação* e *aleatorização*. Os dois primeiros são os mais comumente estudados, o primeiro está associado a quantidade de "memória" extra usada para resolver uma instância do problema e o segundo é dado em função do número de *instruções elementares* realizadas pelo algoritmo. Além desses, pode ser de interesse a quantidade de unidades de informação transmitidas e recebidas (comunicação, veja o exercício 1.60) e, quando analisamos algoritmos probabilísticos, a *quantidade de bits sorteados* pelo algoritmo (veja um caso na página 56).

A *análise de algoritmos* teórica é uma parte importante da Teoria da Computação e fornece a *correção* de algoritmos e estimativas de *eficiência* quanto ao consumo de recursos.

Tamanho da instância Para expressar o consumo de recursos pelos algoritmos nós levamos em conta o *tamanho da instância*. Em último nível, uma instância do problema é dada por alguma *codificação* da instância com uma cadeia de bits, donde definimos o tamanho de uma instância como o número de bits usados na representação da instância.

Na prática, somos menos formais e adotamos algumas simplificações que dependem muito do problema que esta sendo estudado e da representação usada, em geral o tamanho da instância é um inteiro positivo que descreve a quantidade de componentes da instância. Por exemplo, se o problema é multiplicar dois números inteiros, o tamanho é a quantidade de algarismos desses números (em alguma base com pelo menos dois algarismos). Agora, se o problema é multiplicar duas matrizes de números inteiros, o tamanho é, geralmente, a dimensão da matriz pois supõe-se implicitamente que o tamanho da matriz é muito grande quando comparada ao tamanho dos números; se esse não é o caso então o tamanho dos números deve ser levado em conta. Para pesquisar uma lista de itens e determinar se um item particular está presente nela, o tamanho da instância é o número de itens na lista. No problema de ordenação de uma sequência numérica o tamanho é dado pelo número de elementos da sequência. Em algoritmos sobre grafos, o tamanho é ou o número de vértices, ou o número de arestas, ou ambos. Há justificativas razoáveis para tais simplificações mas, mesmo que façamos escolhas concretas de codificação de instâncias, tentamos manter a discussão abstrata o suficiente para que as estimativas sejam independentes da escolha da codificação.

Tempo de execução Como estimamos o *tempo de execução* de um algoritmo para resolver uma determinada instância? São duas ideias preliminares. Primeiro, como já dissemos, determinamos quanto tempo o algoritmo leva em função do tamanho da instância. A segunda ideia é avaliarmos o quão rapidamente a função que caracteriza o tempo de execução aumenta com o tamanho da entrada expressando a ordem de grandeza do crescimento do tempo de execução. O consumo de tempo dos algoritmos, como medida de sua eficiência, expressa o número de *instruções básicas* executadas pelo algoritmo em função do tamanho da entrada descrita em notação assintótica. Isso nos permite algumas simplificações: podemos (quase sempre) assumir que cada linha consome tempo constante (desde que as operações não dependam do tamanho da entrada), mesmo as operações aritméticas podem (quase sempre) serem assumidas de tempo constante — isso tem de ser feito com cuidado, essa hipótese não pode ser assumida no problema de multiplicação de inteiros, por exemplo, onde as operações aritméticas tem custo proporcional ao tamanho dos operandos. Na seção 2.1.1 veremos as notações que usamos para expressar o tempo de execução de um algoritmo.

É natural esperarmos que instâncias maiores demandem mais recurso dos algoritmos de modo que as estimativas para o consumo são feitas para as classes de instâncias que têm o mesmo tamanho. Isso significa que expressamos o desempenho de algoritmos em função do tamanho da representação das instâncias. Notemos que mesmo entre instâncias de mesmo tamanho a quantidade de recursos usados pode variar de modo que devemos adotar alguma estratégia para resumir o tempo de execução do algoritmo para resolver o problema na classe das instâncias de mesmo tamanho:

tomamos o *pior caso* — aquele em que o algoritmo consome mais recursos — ou o *caso médio* — a média de consumo numa classe.

Em Complexidade Computacional convenciona-se que um algoritmo **eficiente** com respeito ao tempo de execução se o número de instruções executadas é limitado superiormente por uma função polinomial no tamanho da instância, para qualquer instância. Na teoria isso é muito conveniente pois

- a classe das funções polinomiais é fechada para soma, multiplicação e composição de funções, assim a noção de eficiência é preservada por práticas comuns de programação;
- os modelos formais tradicionais de computação são polinomialmente equivalentes, o que torna a escolha do modelo irrelevante para essa definição de eficiência;
- com algum cuidado, as várias representações computacionais de objetos abstratos, como um grafo por exemplo, têm tamanhos polinomialmente relacionados, o que faz a codificação ser irrelevante para essa definição de eficiência.

Na prática isso pode não ser representativo de eficiência pois o polinômio pode ter um grau muito alto.

Consideremos o problema cuja instância é uma lista $a_1, a_2, \cdots, a_n$ de inteiros e um inteiro x, a resposta é "sim" ou "não" e significa a ocorrência ou não, respectivamente, de x na lista. Uma solução para o problema é a busca linear: percorra a lista e verifique se cada elemento dela é o item procurado.

Instância : um vetor $(a_1, \ldots, a_n)$ de inteiros e um inteiro x.
Resposta : *sim* se x ocorre em a e *não* caso contrário.

1 $i \leftarrow 1$;
2 **enquanto** $a_i \neq x$ *e* $i < n$ **faça** $i \leftarrow i + 1$;
3 **se** $a_i = x$ **então responda** sim.
4 **senão responda** não.

Algoritmo 3: busca sequencial.

No melhor caso, o elemento x ocorre na primeira posição da sequência, o algoritmo executa a atribuição na linha 1, o teste na linha 2, a comparação na linha 3 e responde. Essencialmente, um número constante de instruções. Nesse caso escrevemos que o tempo de execução é $O(1)$ e isso será justificado na próxima seção.

No pior caso a busca deve visitar cada elemento uma vez. Isso acontece quando x, o valor que está sendo procurado, não está na lista. O numero de instruções executadas é: 1 atribuição na linha 1, mais $4(n-1)$ nas linhas 2 e 3 pois são feitas duas comparações na linha 2, uma adição e uma atribuição na linha 3 repetidas $n-1$ vezes, mais 1 comparação na linha 4 mais 1 retorno na linha 5, portanto, $4(n-1)+3$ instruções. A função $4n-1$ é linear de modo que dizemos que o seu crescimento é da ordem de n, em notação assintótica (que ainda veremos), escrevemos que a *complexidade de tempo*, ou tempo de execução, de pior caso do algoritmo é $O(n)$.

Notemos que, nesse caso, para a ordem de grandeza basta considerar o numero de comparações que são feitas, o restante das instruções contribui com uma constante multiplicativa e uma constante aditiva. Então, se tivéssemos contado apenas o número de comparações também chegaríamos a conclusão de que a complexidade de tempo de pior caso do algoritmo é $O(n)$.

No melhor caso, a quantidade de comparações é constante, não depende de n e no pior caso cresce linearmente com n. No caso médio, assumimos que o valor procurado está na lista e cada elemento da lista é igualmente provável que seja o valor procurado, o número médio de comparações é i se a busca termina na posição i, portanto, o número médio de comparações é

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

Nesse caso, dizemos que a complexidade de tempo de caso médio do algoritmo é $O(n)$ pois, novamente, temos uma função de crescimento linear em n.

Agora, consideremos o problema de ordenar uma sequência de inteiros. Uma solução é o seguinte algoritmo conhecido como ordenação por inserção:

Instância : um vetor $(a_1,\ldots,a_n)$ de inteiros.
Resposta : uma permutação $(a_{\pi(1)},\ldots,a_{\pi(n)})$ tal que $a_{\pi(i)} \leqslant a_{\pi(j)}$ sempre que $\pi(i) < \pi(j)$.

```
1 para i de 2 até n faça
2     x ← a_i;
3     j ← i − 1;
4     enquanto (a_j > x e j ⩾ 1) faça
5         a_{j+1} ← a_j;
6         j ← j − 1;
7     a_{j+1} ← x.
```

Algoritmo 4: ordenação por inserção.

Notemos que o tempo do algoritmo é determinado pela condição no laço da linha 4 e as linhas 2, 3 e 7 são executadas $n-1$ vezes pelo laço da linha 1. Para i fixo, uma rodada completa do laço executa 2 comparações, 2 atribuições e 2 operações; no pior caso[1] o laço é executado para todo j de i ate 1, depois o laço é falso para a segunda condição ($j = 0$). Logo são $6i$ instruções mais as 2 comparações finais. No pior caso, o custo de ordenação por inserção de uma sequência com n elementos é

$$T(n) = 2 + \sum_{i=2}^{n} 6i = 2 + 6\frac{(n+2)(n-1)}{2} = 3n^2 + 3n + 5$$

portanto $T(n)$ tem ordem de crescimento de n^2. Notemos o seguinte, a ordem de grandeza de $T(n)$ é dada pelo fato de termos dois laços aninhados e dentro desses laços o número de instruções executadas em cada rodada é constante de tal forma que, para a ordem de grandeza, não importa se contamos $j \leftarrow j-1$ como 1 ou 2 instruções, é suficiente que seja constante.

Para estimar o caso médio observamos que o fato determinante para o número de instruções executadas é o "tipo de ordem" dos elementos da sequência e não quais são os elementos em si. Por exemplo, ordenar $(1,2,3,4)$ usa o mesmo número de comparações que $(4,7,8,9)$, assim como ordenar $(1,4,3,5,2)$ e $(11,15,14,20,13)$. Em outras palavras, o mesmo tipo de ordem significa que a mesma permutação π da resposta ordena as duas instâncias. Dito isso, assumimos que as instâncias são formadas por sequências de n inteiros distintos fixos e que qualquer uma das $n!$ permutações são igualmente prováveis.

Fixado i, com $2 \leqslant i \leqslant n$, consideremos o subvetor $(a_1,\ldots,a_i)$. Observemos que, para cada i, no início do laço da linha 1 temos $(a_1,\ldots,a_{i-1})$ ordenado e o laço da linha 4 procura a posição de a_i em $(a_1,\ldots,a_{i-1})$ ordenado. Definimos o posto(a_i) como a posição do elemento a_i no subvetor $(a_1,\ldots,a_i)$ ordenado. Por exemplo, o posto de a_4 em $(3,6,2,5,1,7,4)$ é 3 pois em $(3,6,2,5)$, quando ordenado, o número 5 (que é o a_4) ocupa a 3ª posição.

EXERCÍCIO 2.1. Verifique que o posto de a_i é igualmente provável ser qualquer $j \in \{1,2,\ldots,i\}$.

Dado i, o subvetor $(a_1,\ldots,a_{i-1})$ está ordenado com os mesmos $i-1$ primeiros elementos do vetor original da entrada. O teste no laço é executado $i - \text{posto}(a_i) + 1$ vezes. Assim, o número médio de comparações para o qual o teste do laço vale é

$$\sum_{\text{posto}=1}^{i} \frac{i - \text{posto} + 1}{i} = \frac{i+1}{2}.$$

[1] O pior caso para ordenação por inserção ocorrerá quando a lista de entrada estiver em ordem decrescente.

O número médio de comparações efetuadas pelo algoritmo de ordenação por inserção é

$$\sum_{i=2}^{n} \frac{i+1}{2} = \frac{(n+4)(n-1)}{2}$$

que é da ordem de n^2.

2.1.1 NOTAÇÃO ASSINTÓTICA

As estimativas para a ordem de grandeza da quantidade de recursos necessários por um algoritmo em função do tamanho são consideradas "a menos de uma constante multiplicativa", expressas em notação assintótica. Isso garante, por exemplo, que a estimativa teórica seja representativa para as várias possíveis implementações do algoritmo, as quais dependem da máquina, da linguagem de programação e da habilidade do programador dentre outros fatores. A diferença implicada por esses fatores não altera a ordem de grandeza da função. Além disso, essa consideração permite uma simplificação substancial quando contamos o número de instruções executadas e também simplifica a questão da codificação das instâncias como os números, por exemplo, expressos em qualquer base com pelo menos dois símbolos têm representação de ordem um logarítmica de dígitos.

Abaixo f e g são funções[2] de $\mathbb{R}^{\geqslant 0}$ em $\mathbb{R}$ com g *assintoticamente positiva*, ou seja $g(n) > 0$ para todo n suficientemente grande.

Dizemos que f é **assintoticamente muito menor** que g e escrevemos

$$f(n) = o(g(n)) \text{ quando } n \to \infty \tag{2.1}$$

se, e só se,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Por exemplo,

1. $1 = o(\log(\log(n)))$.
2. $\log(\log(n)) = o(\log(n))$.
3. $\log(n) = o(n^{\varepsilon})$ para todo $\varepsilon > 0$.
4. $n^{\varepsilon} = o(n^{c})$ para quaisquer $0 < \varepsilon < 1 \leqslant c$.
5. $n^{c} = o(n^{\log n})$ para todo $1 \leqslant c$.
6. $n^{\log n} = o(\exp(n))$.
7. $\exp(n) = o(n^{n})$.
8. $n^{n} = o(\exp(\exp(n)))$.

As vezes usamos a notação $f \ll g$ o que nos permite escrever a partir do exemplo acima

$$1 \ll \log(\log(n)) \ll \log(n) \ll n^{\varepsilon} \ll n^{c} \ll n^{\log n} \ll e^{n} \ll n^{n} \ll e^{e^{n}}.$$

Dizemos que f **assintoticamente menor** que g e escrevemos

$$f(n) = O(g(n)) \text{ quando } n \to \infty \tag{2.2}$$

[2] Uma função $f(n)$ que expressa o consumo de um recurso por algum algoritmo é uma função de $\mathbb{N}$ em $\mathbb{N}$, mas aqui vamos tratar a notação assintótica de modo um pouco mais geral que nos permitirá usá-la em outras situações.

se existe $n_0 > 0$ e existe $c > 0$ tais que para todo $n \geqslant n_0$

$$|f(n)| \leqslant cg(n).$$

Em geral, omitimos o $n \to \infty$ em (2.1) e (2.2). Nesses casos dizemos que g é um limitante superior assintótico para f. O símbolo = na definição não é a igualdade no sentido usual, é um abuso da notação em troca de algumas conveniências, no entanto, $n = O(n^2)$ e $n^2 + 2n + 1 = O(n^2)$ mas $n \neq n^2 + 2n + 1$. A rigor, $O(g(n))$ é um conjunto de funções, a saber, aquelas que são limitadas superiormente e assintoticamente por $g(n)$.

Proposição 2.2 *Se $f(n) = o(g(n))$ então $f(n) = O(g(n))$.*

A prova é imediata da definição de limite. A recíproca da proposição 2.2 não vale, como pode ser visto tomando-se $f(n) = g(n) = n^2$.

Proposição 2.3 *Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ então*

1. $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.
2. $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.
3. $a \cdot f_1(n) = O(g_1(n))$ *para toda constante* a.

Demonstração. Vamos provar o item 1. Digamos que $|f_1(n)| \leqslant c_1 g_1(n)$ para todo $n \geqslant n_1$ e que $|f_2(n)| \leqslant c_2 g_2(n)$ para todo $n \geqslant n_2$, onde n_1, n_2, c_1, c_2 são as constantes dadas pela definição de notação O. Então

$$\begin{aligned} |f_1(n) + f_2(n)| \leqslant |f_1(n)| + |f_2(n)| &\leqslant \leqslant c(g_1(n) + g_2(n)) \text{ com } c = \max\{c_1, c_2\} \\ &\leqslant 2c(\max\{g_1(n), g_2(n)\}) \end{aligned}$$

para todo $n \geqslant \max\{n_1, n_2\}$, o que prova a afirmação.

As outras propriedades são deduzidas de modo análogo e são deixadas como exercício. □

É preciso cuidado com o teorema acima pois, por exemplo, não vale que $1^k + 2^k + \cdots + (n-1)^k + n^k = O(\max\{1^k, 2^k, \ldots, (n-1)^k, n^k\}) = O(n^k)$. O problema aqui é que o máximo só pode ser tomado sobre um número de termos que não dependa de n. De fato, $1^k + \cdots + n^k = O(n^{k+1})$ mas $1^k + \cdots + n^k \neq O(n^k)$.

Fica como exercício verificar o seguinte resultado.

Proposição 2.4 *Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f(n) = O(h(n))$.*

Alguns exemplos são dados a seguir.

1. $an^2 + bn + c = O(n^2)$ para toda constante $a > 0$.

   Primeiro, observamos que $|an^2 + bn + c| \leqslant |a|n^2 + |b|n + |c|$ e agora usamos a proposição 2.3 em cada operando das somas, de $an^2 = O(n^2)$, $|b|n = O(n)$ e $|c| = O(1)$ temos $an^2 + |b|n + |c| = O(\max\{n^2, n, 1\}) = O(n^2)$. Analogamente, para todo $k \in \mathbb{N}$

   $$\sum_{i=0}^{k} a_i n^i = O(n^k).$$

2. $n\log(n!) = O(n^2 \log n)$.

   Primeiro, temos $n = O(n)$. Depois, $n! = \prod_{i=1}^{n} i < \prod_{i=1}^{n} n = n^n$. Como log é crescente $\log(n!) < \log(n^n) = n\log(n)$, portanto $\log(n!) = O(n\log(n))$. Pela proposição 2.3 $n\log(n!) = O(n^2 \log n)$.

3. Para toda constante $a > 1$, $\log_a(n) = O(\log(n))$. De fato,

$$\log_a(n) = \frac{1}{\log a} \log n$$

porém $\frac{1}{\log a} = O(1)$ e $\log n = O(\log n)$ e pela proposição 2.3 $\log_a(n) = O(\log(n))$.

Convenções de notação Ao usar notação assintótica, desconsideramos os coeficientes, por exemplo, usamos $O(n^2)$ ao invés de $O(3n^2)$ e $O(1)$ ao invés de $O(1024)$. Notemos que, como classes de funções $O(n^2) = O(3n^2)$ e $O(1) = O(1024)$. Escrevemos no argumento de $O(\cdot)$ somente o termo mais significativo, por exemplo, usamos $O(n^2)$ ao invés de $O(2n^2 + 5n\log n + 4)$. Notemos que da proposição 2.3 vale que $2n^2 + 5n\log n + 4 = O(\max\{n^2, n\log n, 1\}) = O(n^2)$. A ideia é que a função f em $g(n) = O(f(n))$ resuma a ordem de grandeza da função g.

Um algoritmo eficiente com respeito ao tempo de execução é um algoritmo que nas instâncias de tamanho n tem tempo de execução $O(n^k)$ para algum k fixo.

Quando notação assintótica aparece em equações, na forma "expressão 1 = expressão 2" onde "expressão" são expressões algébricas que envolvem notação assintótica, os termos assintóticos em "expressão 1'são quantificados universalmente, enquanto que os termos assintóticos em "expressão 2" são quantificados existencialmente. Por exemplo, em $n^3 + O(n^2) = O(n^3) + n^2 + n$ entendemos como

$$\text{para todo } f(n) = O(n^2), \text{ existe } g(n) = O(n^3), \text{ tal que } n^3 + f(n) = g(n) + n^2 + n$$

para todo n suficientemente grande.

Notação Ω e Θ A notação Ω foi introduzida por Donald Knuth e escrevemos $f(n) = \Omega(g(n))$ se, e somente se, $g(n) = O(f(n))$. Escrevemos $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $g(n) = O(f(n))$.

EXERCÍCIO 2.5. Suponha que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$. Verifica se valem as firmações

(a) $n^{1,5} = O(n^2)$.

(b) $\frac{n^2}{10} = O(n)$.

(c) $n^2 - 100n = O(n^2)$.

(d) $n\log(n) = O(n^2)$.

(e) $n = O(n\log n)$.

(f) $2^n = O(n)$.

(g) $2^n = O(2^{n-1})$.

(h) Se $L > 0$, $f(n) = \Theta(g(n))$.

(i) Se $L = 0$, $f(n) = O(g(n))$ mas $f(n) \neq \Theta(g(n))$.

(j) Se $L = \infty$, $f(n) = \Omega(g(n))$ mas $f(n) \neq \Theta(g(n))$.

(k) Se $a_k > 0$, então $\sum_{i=0}^{k} a_i n^i = \Theta(n^k)$.

O algoritmo de Euclides O seguinte algoritmo é conhecido como Algoritmo de Euclides (circa 300 aC). Ele determina o maior divisor comum de dois inteiros quaisquer. É um algoritmo recursivo baseado no fato de que $\text{mdc}(a, b) = \text{mdc}(b, a \bmod b)$ onde $a \bmod b$ é o resto na divisão de a por b.

Instância : um par de inteiros (a, b).
Resposta : $\text{mdc}(a, b)$.

1 $a \leftarrow |a|$
2 $b \leftarrow |b|$;
3 **se** $b = 0$ **então responda** a.
4 **senão responda** $\text{mdc}(b, a \bmod b)$.

Algoritmo 5: $\text{mdc}(a, b)$.

O algoritmo de Euclides está correto. Notemos que $\mathrm{mdc}(a,b) = \mathrm{mdc}(|a|,|b|)$ e que que $\mathrm{mdc}(a,0) = |a|$, para todo $a \in \mathbb{Z}$, portanto, podemos assumir que $a > b > 0$. Também, notemos que o algoritmo termina pois $0 \leqslant a \bmod b < b$, pelo Teorema da Divisão, logo o valor da variável b decresce estritamente a cada iteração. Para concluir, observamos que se a e $b > 0$ são inteiros então

$$d|a \text{ e } d|b \Leftrightarrow d|b \text{ e } d|a \bmod b$$

donde pode-se concluir (verifique) que se a e $b > 0$ são inteiros então $\mathrm{mdc}(a,b) = \mathrm{mdc}(b, a \bmod b)$.

O algoritmo de Euclides computa $\mathrm{mdc}(a,b)$ em tempo $O(\log(|a|)\log(|b|))$. Consideremos a seguinte sequência dos parâmetros das chamadas recursiva do algoritmo de Euclides, começando com $(a,b) = (r_0, r_1)$

$$\begin{aligned}
(r_1, r_0 \bmod r_1) &= (r_1, r_2) \\
(r_2, r_1 \bmod r_2) &= (r_2, r_3) \\
&\vdots \\
(r_{\ell-2}, r_{\ell-3} \bmod r_{\ell-2}) &= (r_{\ell-2}, r_{\ell-1}) \\
(r_{\ell-1}, r_{\ell-2} \bmod r_{\ell-1}) &= (r_{\ell-1}, r_\ell)
\end{aligned}$$

e $r_{\ell+1} = 0$. O custo em cada linha é o de uma divisão, ou seja, na linha i, $1 \leqslant i < \ell$, o custo é $O(\log(r_i)\log(q_i))$, onde $r_{i-1} = r_i q_i + r_{i+1}$. Agora, o custo total é

$$\sum_{i=1}^{\ell} \log(r_i)\log(q_i) \leqslant \log(b) \sum_{i=1}^{\ell} \log(q_i) = \log(b)\log(q_1 q_2 \cdots q_\ell) \leqslant \log(b)\log(a)$$

pois $a = r_0 \geqslant r_1 q_1 \geqslant r_2 q_2 q_1 \geqslant \cdots \geqslant r_\ell q_\ell \cdots q_2 q_1 \geqslant q_\ell \cdots q_2 q_1$.

Teorema 2.6 *A complexidade do tempo de execução do algoritmo* Euclides(a,b) *é* $O(\log(|a|)\log(|b|))$. □

EXERCÍCIO 2.7 (o pior caso do Algoritmo de Euclides). No que segue, f_k é o k-ésimo número de Fibonacci, definido recursivamente por $f_1 = 1$, $f_2 = 1$ e $f_k = f_{k-1} + f_{k-2}$ para $k \geqslant 3$. Mostre que o algoritmo de Euclides com entradas f_{k+2} e f_{k+1} executa k chamadas recursivas. Prove o seguinte resultado: *se* (a,b) *é o menor par de inteiros positivos que faz a algoritmo de Euclides executar* k *chamadas recursivas então* $(a,b) = (f_{k+1}, f_{k+2})$.

O algoritmo de Euclides estendido O algoritmo de Euclides Estendido determina uma solução (x,y) inteira da equação

$$ax + by = \mathrm{mdc}(a,b) \tag{2.3}$$

para $a,\ b \in \mathbb{Z}$ quaisquer. Por quê (2.3) tem solução?

Esse algoritmo é bastante útil na seguinte situação. Se $\mathrm{mdc}(a,n) = 1$ para inteiros a e $n > 1$, então a equação $ax \equiv c \pmod{n}$ tem solução, ou seja, exitem x e y inteiros tais que $ax + ny = c$ e ao algoritmo estendido os encontra.

No caso $c = 1$ a solução é um **inverso multiplicativo de** a **módulo** n.

Instância : (a,b) par de inteiros não-negativos.
Resposta : uma terna (d,x,y) tal que $d = \mathrm{mdc}(a,b) = ax + by$.

1 **se** $b = 0$ **então responda** $(a,1,0)$.
2 $(d,x,y) \leftarrow$ Euclides_estendido$(b, a \bmod b)$
3 **responda** $(d, y, x - \lfloor a/b \rfloor y)$.

Algoritmo 6: Euclides_estendido(a,b).

Uma execução do algoritmo acima com entradas 86 e 64 resulta nos seguintes valores

a	b	$\lfloor a/b \rfloor$	x	y	d
86	64	1	3	−4	2
64	22	2	−1	3	2
22	20	1	1	−1	2
20	2	10	0	1	2
2	0	—	1	0	2

EXERCÍCIO 2.8 (tempo de execução do algoritmo euclidiano estendido). Verifique que o tempo de execução do algoritmo 6 é $O(\log(|a|)\log(|b|))$.

EXERCÍCIO 2.9 (correção do algoritmo euclidiano estendido). Prove que o algoritmo 6 responde corretamente. Para isso, suponha que (d', x', y') são os valores atribuídos na linha 2

$$d' = bx' + (a \bmod b)y' = bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y').$$

A prova segue por indução.

EXPONENCIAÇÃO MODULAR Computar 2^n no modo tradicional é extremamente custoso quando n é grande; com 100 dígitos isso daria cerca de 10^{100} passos o que é impossível de realizar manualmente sem atalhos. Em geral a^b avaliado por multiplicações repetidas tem tempo de execução $\Omega(b(\log a)^2)$. Um jeito mais esperto é "elevar ao quadrado" repetidas vezes, por exemplo, para calcular 2^{24} podemos começar com $2^3 = 8$, elevá-lo ao quadrado, o que resulta $2^6 = 62$, elevá-lo ao quadrado, o que resulta $2^{12} = 4.096$, e elevá-lo ao quadrado, o que resulta $2^{24} = 16.777.216$. Para calcular 2^{29} com esse método de trás pra frente teríamos $2^{29} = 2 \cdot 2^{28}$, a raiz de 2^{28} é 2^{14} cuja raiz é 2^7 que é $2 \cdot 2^6$ que por sua vez é $2^2 \cdot 2^3$. Em resumo, a^b é avaliado com base na observação de que

$$a^b = \begin{cases} (a^{b/2})^2 & \text{se } b \text{ é par,} \\ a \cdot a^{b-1} & \text{se } b \text{ é ímpar.} \end{cases}$$

O seguinte algoritmo é uma versão iterativa da recursão acima para calcular $a^b \pmod n$. Seja $b_k b_{k-1} \dots b_1 b_0$ a representação binária de b e defina

$$c_i := b_k 2^{i-1} + b_{k-1} 2^{i-2} + \cdots + b_{k-i+1} 2^0$$
$$d_i := a^{c_i} \bmod n$$

para $i \geqslant 1$ com $c_0 = 0$ e $d_0 = 1$. Computamos d_{i+1} a partir de d_i da seguinte forma

$$c_{i+1} = \begin{cases} 2c_i, & \text{se } b_{k-i} = 0 \\ 2c_i + 1, & \text{se } b_{k-i} \neq 0 \end{cases};$$

e

$$d_{i+1} = \begin{cases} {d_i}^2 \bmod n = a^{c_{i+1}} \bmod n = (a^{c_i})^2 \bmod n & \text{se } b_{k-i} = 0 \\ a \cdot {d_i}^2 \bmod n = a^{c_{i+1}} \bmod n = a \cdot (a^{c_i})^2 \bmod n, & \text{se } b_{k-i} \neq 0. \end{cases}$$

Portanto, $c_{k+1} = b_k 2^k + \cdots + b_1 2 + b_0 = b$ e $d_{k+1} = a^b \bmod n$.

Exemplo 2.10. Vamos usar essa estratégia para calcular $2^{24} \bmod 25$. Primeiro, 24 em base 2 fica $b = 11000$.

i	0	1	2	3	4	5
b_{4-i}	1	1	0	0	0	
c_i	0	1	3	6	12	24
d_i	1	2	8	14	21	16

Portanto $2^{24} \equiv 16 \pmod{25}$. ◊

Notemos que em toda iteração os valores de d têm no máximo tantos dígitos quanto n, ou seja, têm $O(\log n)$ dígitos, portanto, a multiplicação e o resto têm tempo de execução $O(\log^2 n)$. O número de iterações é a quantidade de bits na representação de b, logo $O(\log b)$. Isso prova o seguinte resultado.

Teorema 2.11 *O algoritmo 7 abaixo com* $a = O(\log n)$ *determina* $a^b \bmod n$ *em tempo* $O(\log(b)\log^2(n))$.

Instância : inteiros não-negativos a, b e $n > 1$.
Resposta : $a^b \bmod n$.

```
1  c ← 0;
2  d ← 1;
3  Seja b_k b_{k−1} ... b_1 b_0 a representação binária de b;
4  para i de k até 0 faça
5  |  c ← 2 · c;
6  |  d ← d · d mod n;
7  |  se b_i = 1 então
8  |  |  c ← c + 1;
9  |  |  d ← d · a  mod n;
10 responda d.
```

Algoritmo 7: exponenciação modular.

Observação 2.12 (*sobre a complexidade das operações aritméticas*). O custo das operações aritméticas dados acima são os custos dos algoritmos escolares, não os dos mais eficientes. Se $M(n)$ é o custo para multiplicar dois números de até n bits, então temos os seguintes tempos de execução.

Multiplicação	$M(n)$
Divisão	$O(M(n))$ (Newton–Raphson)
MDC	$O(M(n)\log n)$ (Stehlé–Zimmermann)
Exponenciação modular	$O(kM(n))$, k é o tamanho do expoente

Tabela 2.1: custo das operações aritméticas.

O tempo de uma multiplicação do algoritmo escolar é $M(n) = O(n^2)$. Atualmente, o algoritmo mais usado é o algoritmo de Schönhage–Strassen de 1971. A complexidade de tempo de execução é $O(n \log n \log\log n)$ para dois números de n dígitos. Esse algoritmo foi o método de multiplicação mais rápido até 2007, quando o algoritmo de Fürer foi anunciado. Entretanto, o algoritmo de Fürer só alcança uma vantagem para valores astronomicamente grandes e não é usado na prática.

Harvey e Van Der Hoeven (2019) publicaram um algoritmo com complexidade $O(n \log n)$ para a multiplicação de inteiros. Como Schönhage e Strassen conjecturam que $n\log(n)$ é mínimo necessário para a multiplicação esse pode ser o "melhor possível", porém, até o momento esse algoritmo também não é útil na prática pois os autores observam, no início da seção 5, que as estimativas de custo valem para números com pelo menos $2^{4.096}$ bits. O número de átomos estimado no universo é $10^{80} \approx 2^{266}$.

2.2 ALGORITMOS ALEATORIZADOS

Na análise dos algoritmos aleatorizados procuramos determinar um limitante para a probabilidade de erro e limitantes para o consumo de recursos, que neste texto será principalmente tempo e bits aleatórios.

No teste de identidade polinomial (algoritmo 1, página 19) o tempo de execução depende do tempo para o sorteio de a e do tempo para computar $f(a)$. Formalmente, consideramos que os algoritmos sorteiam bits uniforme e independentemente em tempo constante, de modo que o sorteio de a tem tempo $O(\log a)$. Em geral, se $\log_2 |\Omega|$ for polinomial no tamanho da entrada então um sorteio não afeta o tempo de execução de um algoritmo de tempo polinomial e podemos considerar o tempo de um sorteio constante. O tempo para computar $f(a)$ depende da representação de f e será eficiente se for feito em tempo polinomial no tamanho da representação de f. Isso se verifica quando o polinômio é dado explicitamente e vamos assumir esse fato por enquanto. O algoritmo 1 é um algoritmo probabilístico de tempo polinomial que erra com probabilidade limitada. Esse tipo de algoritmo é chamado na literatura de algoritmo de *Monte Carlo*.

O algoritmo gerador de números aleatórios, algoritmo 2, página 32, sempre responde certo, logo não é um algoritmo Monte Carlo. Porém, o tempo de execução pode ser diferente em execuções com a mesma instância. Uma execução do algoritmo com entrada M com uma única rodada do laço tem tempo de execução $O(\log M)$ para sortear os bits e realizar a soma, outra execução com a mesma entrada M pode mais azarada e precisar de 2 rodadas, mas o tempo de execução continua $O(\log M)$. Um outra execução com a mesma entrada M pode ser muito azarada e precisar de M rodadas e o tempo de execução será $O(M \log M)$, que é exponencial no tamanho da entrada! O que nos tranquiliza é que sabemos que isso é muito pouco provável. O que fazemos nesse caso para ter uma estimativa representativa é tomar a média do número de rodadas ponderada pela probabilidade. Se $p = M/2^k$ é a probabilidade com que o algoritmo executa exatamente uma rodada, $(1-p)p$ é a probabilidade com que o algoritmo executa exatamente duas rodadas, $(1-p)^2 p$ para três rodadas e assim por diante. A média é (veja (s.6)).

$$\sum_{k \geqslant 1} k(1-p)^{k-1} p = \frac{1}{p} = \frac{2^k}{M} \leqslant 2 \tag{2.4}$$

rodadas, onde $k = \lfloor \log_2 M \rfloor + 1$. Portanto, o *tempo médio de execução* é $2 \cdot O(\log M)$, ou seja, $O(\log M)$. Esse tipo de algoritmo é chamado de um algoritmo de *Las Vegas*.

Há, ainda, algoritmos que têm probabilidade positiva de errar e têm probabilidade positiva de demorar muito pra terminar e que em algumas referências são chamados de *Atlantic City*.

Um fato importante a ser ressaltado neste momento é que esse tempo médio que calculamos é sobre os sorteios do algoritmo, diferente do que fizemos com os algoritmos de busca sequencial e de ordenação por inserção onde a média foi feita sobre o tempo de execução nas diferentes entradas para o algoritmo. A única fonte de aleatoriedade nos algoritmos probabilísticos são os bits aleatórios que ele usa, não há uma mediada no conjunto de instâncias.

Dado um algoritmo A e uma instância x para A, podemos representar as possíveis computações de A com x com uma árvore binária $T_{A,x}$ (figura 2.1) em que cada ramificação significa que um sorteio foi realizado, uma computação c específica está associada a um caminho da raiz até a folha e ela ocorre com probabilidade $\mathbb{P}(c) = 2^{-|c|}$ onde $|c|$ é o comprimento (número de arestas) no caminho.

Para termos um exemplo simples de uma árvore de execução, vamos modificar ligeiramente o algoritmo 1 para que faça até dois sorteios: se no primeiro $f(a) \neq 0$ então pode parar e responder *não*, senão faça mais um sorteio. Além disso, as instâncias são polinômios de grau no máximo 2.

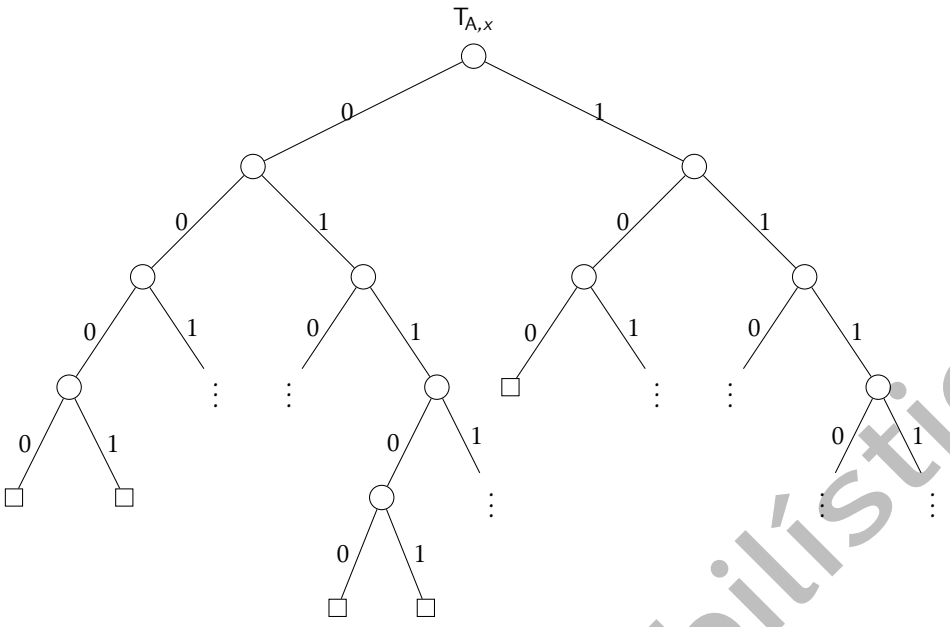


Figura 2.1: árvore de execução de A com instância x.

Instância : d inteiro positivo e f polinômio de grau $\leqslant 2$.
Resposta : *não* se f não é nulo, senão *sim*.

```
1 a ←_R {1,2,3,4};
2 se f(a) ≠ 0 então responda não.
3 senão
4 |   a ←_R {1,2,3,4};
5 |   se f(a) ≠ 0 então responda não.
6 |   senão responda sim.
```

As computações desse algoritmo com entrada $(x-1)(x-3)$ em função dos sorteios são caracterizadas pelas sequências: $(1,1,\textit{não})$, $(1,2,\textit{não})$, $(1,3,\textit{sim})$ e $(1,4,\textit{não})$, $(2,\textit{não})$, $(4,\textit{não})$, $(3,1,\textit{não})$, $(3,2,\textit{não})$, $(3,3,\textit{sim})$ e $(3,4,\textit{não})$, esquematizadas na figura 2.2. Aqui, para simplificar a exposição, consideramos os sorteios dos números ao invés de bits. Dos sorteios independentes decorre que a probabilidade de uma computação é o produto das probabilidades no ramo daquela computação, logo

$$\mathbb{P}[\,\text{erro}\,] = \mathbb{P}((1,3,\textit{sim})) + \mathbb{P}((3,3,\textit{sim})) = \frac{1}{2}\cdot\frac{1}{2} + \frac{1}{2}\cdot\frac{1}{2}.$$

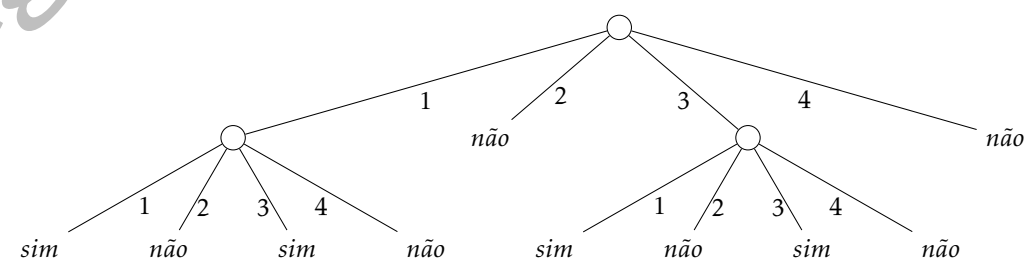


Figura 2.2: árvore de execução do algoritmo acima para $(x-1)(x-3)$.

Definimos o modelo probabilístico discreto $(\Omega_{A,x}, \mathbb{P})$ com o espaço amostral formados pelos caminhos c na árvore de execução e a medida $2^{-|c|}$ em que $c \in \Omega_{A,x}$ é um ramo da árvore e $|c|$ o número de arestas em c.

EXERCÍCIO 2.13. Verifique que para quaisquer dois ramos distintos de $T_{A,x}$ vale que a sequência binária de um ramo não pode ser prefixo da sequência de outro ramo. Prove que $\rho = \sum_c 2^{-|c|} \leqslant 1$. Nessa situação, ρ é a probabilidade com que A com instância x termina a computação.

2.2.1 CORTE-MÍNIMO EM GRAFOS

Um *grafo* G é dado por um par de conjuntos (V, E) em que V é finito, é o conjunto dos *vértices* de G, e $E \subset \binom{V}{2}$. O *grau* de um vértice $x \in V$ em G é a quantidade de arestas de E a que x pertence. As vezes é conveniente representarmos um grafo por um diagrama em que cada vértice é um ponto do plano e dois pontos diferentes são ligados por um segmento de curva se, e somente se, tais pontos representam vértices que compõem uma aresta do grafo. Um exemplo é dado na figura 2.4 abaixo.

Se contamos o número de pares $(v, e) \in V \times E$ tais que $v \in e$ temos, pela definição de grau, que a quantidade de pares é a soma dos graus dos vértices. Por outro lado, cada aresta é composta por dois vértices de modo que a quantidade de pares é $2|E|$. Esse resultado quase sempre é o primeiro teorema nos textos de Teoria dos Grafos: *em todo grafo, a soma dos graus dos vértices é duas vezes o número de arestas do grafo.*

Seja $G = (V, E)$ um grafo sobre o conjunto de vértices V. Podemos assumir sem perda de generalidade que $V = \{1, 2, \ldots, n\}$. Um subconjunto de arestas de G da forma

$$\nabla(A) := \left\{ \{u, v\} \in E(G) : u \in A \text{ e } v \in \overline{A} \right\}$$

é chamado de **corte definido por** A em G.

Exemplo 2.14. A figura 2.3 abaixo mostra um grafo G e um corte de arestas $\nabla(A)$ definido por $A = \{0, 1, 2, 7, 8\}$, o qual é formado pelas arestas (em azul na figura) $\{0, 4\}, \{0, 5\}, \{1, 3\}, \{1, 6\}, \{8, 3\}, \{6, 8\}, \{5, 7\}, \{6, 7\}, \{2, 3\}, \{2, 4\}$ que cruzam

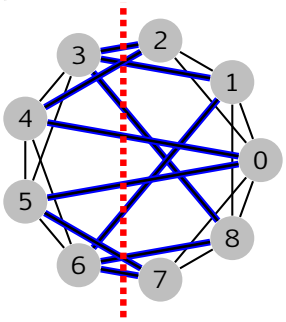


Figura 2.3: o corte definido por $\{0, 1, 2, 7, 8\}$ são as arestas em azul.

a reta vertical pontilhada. ◊

Um **corte mínimo** em G é um corte com

$$\text{mincut}(G) := \min\left\{ \left|\nabla(A)\right| : \varnothing \neq A \subsetneq V \right\}$$

arestas. No grafo do exemplo 2.14, o corte definido por $\{2\}$ é mínimo, assim como o definido por $\{7\}$. O problema em que estamos interessados é enunciado como segue.

Problema computacional do corte mínimo em um grafo (MINCUT):
Instância : um grafo G e um inteiro positivo k.
Resposta : *sim* se $\text{mincut}(G) \leqslant k$, *não* caso contrário.

A seguir, para explicar o algoritmo, precisaremos de uma definição mais geral de grafo. Em um *multigrafo* as arestas formam um multiconjunto no qual entre um par de vértices pode haver mais de uma aresta. Seja M um multigrafo. Para qualquer aresta $e \in E(M)$ em M definimos por *contração da aresta* e a operação que resulta no multigrafo com os extremos de e identificados e as arestas com esses extremos removidos, o multigrafo resultante é denotado por M/e (veja uma ilustração na figura 2.5 abaixo).

A ideia do algoritmo para decidir se $\text{mincut}(G) \leqslant k$ é repetir as operações

1. sortear uniformemente uma aresta,
2. contrair a aresta sorteada,

até que restem 2 vértices no multigrafo. As arestas múltiplas que ligam esses 2 vértices são arestas de um corte no grafo original. Os próximos parágrafos ilustram a ideia do algoritmo que será apresentado em seguida; para facilitar a compreensão mantemos nos rótulos dos vértices todas as identificações realizadas. Considere o grafo G representado pelo diagrama da figura 2.4. A figura 2.5 abaixo representa uma sequência de três contrações de arestas, a aresta que

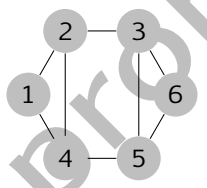


Figura 2.4: exemplo de um grafo.

sofre a contração está em vermelho. Se no multigrafo final da figura 2.5 tem como a próxima contração de aresta a que

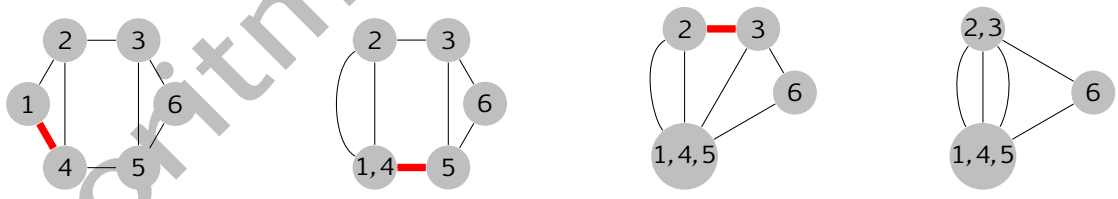
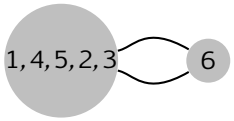


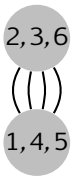
Figura 2.5: uma sequência de três contrações de aresta. Uma contração na aresta em vermelho resulta no multigrafo à direita na sequência. Para manter o registro das contrações acumulamos os rótulos nos vértices.

identifica os vértices representados por $1,4,5$ e por $2,3$ então o multigrafo resultante dessa contração é mostrado na figura 2.6(a) que corresponde ao corte definido por $A = \{6\}$ no grafo original G. Esse corte em G tem duas arestas e é um corte mínimo. Por outro lado, se identificarmos $2,3$ com 6 então o multigrafo obtido corresponde ao corte definido por $A = \{2,3,6\}$ em G e que tem 4 arestas, como na figura 2.6(b).

EXERCÍCIO 2.15. Seja G um grafo. Prove que após uma sequência qualquer de contrações de arestas de G, um corte no multigrafo resultante corresponde a um corte no grafo original. Conclua que a sequência de operações sortear aresta e contrair a aresta sorteada até que restem 2 vértices determina um corte em G.



(a)



(b)

Figura 2.6: dois resultados possíveis para contração a partir do último multigrafo da figura 2.5. Em (a) o resultado da contração da aresta de extremos $1,4,5$ e $2,3$. Em (b) o resultado da contração da aresta $2,3$ com 6. Ambos correspondem a um corte no grafo original G.

Sejam $G = (V, E)$ um grafo com n vértices e $C = \nabla(A)$ um corte mínimo em G. Vamos mostrar que a probabilidade do algoritmo que descrevemos encontrar o corte C é pelo menos $\binom{n}{2}^{-1}$. De $\text{mincut}(G) = |C|$ o grau mínimo de um vértice em G é pelo menos $|C|$, portanto, G tem pelo menos $|C|n/2$ arestas.

O espaço amostral nesse caso é dado pelas sequências de $n-2$ arestas distintas que correspondem as escolhas aleatórias do algoritmo. O algoritmo executado sobre G encontra o corte mínimo $C = \nabla(A)$ se nas $n-2$ rodadas somente contrai arestas com ambos os extremos em A, ou com ambos extremos em $\overline{A}$.

Denotemos por B_i o evento "a i-ésima escolha aleatória, a aresta e_i, não está em C". A probabilidade de escolher uma aresta de C na primeira escolha aleatória é

$$\frac{|C|}{|E(G)|} \leqslant \frac{|C|}{|C|n/2} = \frac{2}{n}$$

logo $\mathbb{P}(B_1) \geqslant 1 - \frac{2}{n}$. Agora, denotemos por G_1 o grafo resultante da primeira rodada de contrações. A probabilidade de escolher uma aresta de C na segunda escolha, dado que na primeira escolha não ocorreu uma aresta de C é

$$\frac{|C|}{|E(G_1)|} \leqslant \frac{|C|}{|C|(n-1)/2} = \frac{2}{n-1}$$

pois o multigrafo tem $n-1$ vértices e grau mínimo pelo menos $|C|$, logo

$$\mathbb{P}(B_2 \mid B_1) \geqslant 1 - \frac{2}{(n-1)}$$

e, genericamente, na i-ésima escolha a probabilidade de escolher uma aresta de C dado que até agora não foi escolhida uma aresta de C é

$$\mathbb{P}\left(B_i \;\middle|\; \bigcap_{j=1}^{i-1} B_j\right) \geqslant 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}.$$

A probabilidade de nenhuma aresta escolhida ser de C é $\mathbb{P}(B_1 \cap B_2 \cap \cdots \cap B_{n-2})$ e pelo exercício 1.24, página 22, temos que

$$\mathbb{P}\left(\bigcap_{i=1}^{n-2} B_i\right) \geqslant \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}.$$

O seguinte algoritmo, devido a Karger (1993), recebe um grafo G com pelo menos 3 vértices e um inteiro positivo k, e responde *sim* ou *não*. Quando o algoritmo responde *sim* é porque foi descoberto um corte com até k arestas, portanto, o corte mínimo tem tamanho no máximo k. Por outro lado, a resposta *não* significa que o algoritmo não achou um corte

com até k arestas, o que não significa que o grafo não o tenha, portanto, a resposta *não* pode estar errada.

Instância : um grafo G com $n \geqslant 3$ vértices e $k \in \mathbb{N}$.
Resposta : *sim* caso $\text{mincut}(G) \leqslant k$, senão *não* com probabilidade de erro $< 1/2$.

```
1  repita
2      i ← 0;
3      G_0 ← G;
4      repita
5          e ←_R E(G_i);
6          G_{i+1} ← G_i/e;
7          i ← i + 1;
8      até que i = n − 2;
9      se |E(G_{n−2})| ⩽ k então responda sim.
10 até que complete (n 2) rodadas;
11 responda não.
```

Algoritmo 9: corte mínimo.

Acima provamos o seguinte resultado.

Proposição 2.16 *Seja G um grafo com pelo menos três vértices. Fixado um corte mínimo C em G, a probabilidade do algoritmo 9 determinar C no laço da linha 4 é pelo menos $1/\binom{n}{2}$.* □

Agora, vamos determinar a probabilidade de erro.

Teorema 2.17 *Supondo que as rodadas do laço da linha 1 sejam independentes temos*

$$\mathbb{P}[\,\text{erro}\,] = \mathbb{P}[\,\text{resposta } \textit{não} \mid \text{mincut}(G) \leqslant k\,] < \frac{1}{e}.$$

Demonstração. Se G tem um corte com no máximo k arestas, então a probabilidade do algoritmo não encontrar um tal corte em nenhuma das iterações do laço na linha 1 é no máximo

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}}.$$

Da sequência decrescente $(1-(1/n))^n$ convergir para e^{-1} (veja (s.8)) temos que qualquer subsequência converge para o mesmo valor

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}} \leqslant \frac{1}{e}$$

e isso prova o teorema. □

O número de instruções executadas no laço mais interno desse algoritmo é $O(|E|) = O(n^2)$. Contando as execuções do laço externo são $O(n^4)$ instruções executadas.

Notemos que se o número de rodadas na linha 10 for $\ell\binom{n}{2}$ então a probabilidade de erro é menor que $e^{-\ell}$, para $\ell = c \log n$ a probabilidade de erro é n^{-c}. Se executarmos o laço externo $c\log(n)\binom{n}{2}$ vezes, então a complexidade de tempo é $O(n^4 \log n)$.

2.2.2 VERIFICAÇÃO DO PRODUTO DE MATRIZES

Nesta seção veremos um algoritmo que recebe as matrizes A, B e C e verifica o produto $A \cdot B = C$ realizando menos operações aritméticas que o próprio produto $A \cdot B$ realiza usando o melhor algoritmo conhecido até hoje.

Problema computacional do teste de produto de matrizes:

Instância : matrizes A, B, C quadradas de ordem n.

Resposta : *sim* se $AB = C$, caso contrário *não*.

Esse teste pode ser feito usando o algoritmo usual (escolar) para o produto de matrizes, o qual realiza $O(n^3)$ operações aritméticas. Um dos algoritmos mais eficientes conhecidos é o de Coppersmith–Winograd (veja em Knuth, 1981), que realiza o produto de duas matrizes $n \times n$ perfazendo da ordem de $n^{2,376}$ operações aritméticas. O algoritmo aleatorizado devido a Freivalds (1977) apresentado a seguir decide se $AB = C$ com $O(n^2)$ operações aritméticas, mas pode responder errado caso $AB \neq C$.

A ideia do algoritmo de Freivalds para esse problema é que se $AB = C$ então $(vA)B = vC$ para todo vetor v e esse último teste tem custo da ordem de n^2 operações aritméticas. Porém, se $AB \neq C$ então é possível termos $(vA)B = vC$, por exemplo, caso v seja nulo. O que conseguimos garantir é que se o vetor v é aleatório então tal igualdade ocorre com probabilidade pequena.

Por exemplo, sejam

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ e } B = \begin{pmatrix} 3 & 6 & 9 \\ 1 & 2 & 1 \\ 3 & 1 & 3 \end{pmatrix}, \; AB = \begin{pmatrix} 14 & 15 & 13 \\ 35 & 40 & 39 \\ 56 & 67 & 63 \end{pmatrix} \text{ e } C = \begin{pmatrix} 14 & 15 & 13 \\ 10 & 20 & 10 \\ 56 & 67 & 63 \end{pmatrix}$$

de modo que $AB \neq C$. Consideremos v um vetor binário. Para $v = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$ temos

$$vAB = 0\begin{pmatrix} 14 & 15 & 13 \end{pmatrix} + 1\begin{pmatrix} 35 & 40 & 39 \end{pmatrix} + 0\begin{pmatrix} 56 & 67 & 63 \end{pmatrix} = \begin{pmatrix} 35 & 40 & 39 \end{pmatrix}$$
$$vC = 0\begin{pmatrix} 14 & 15 & 13 \end{pmatrix} + 1\begin{pmatrix} 10 & 20 & 10 \end{pmatrix} + 0\begin{pmatrix} 56 & 67 & 63 \end{pmatrix} = \begin{pmatrix} 10 & 20 & 10 \end{pmatrix}$$

portanto, $vABC \neq vC$, enquanto que para $v = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ temos

$$vAB = 0\begin{pmatrix} 14 & 15 & 13 \end{pmatrix} + 0\begin{pmatrix} 35 & 40 & 39 \end{pmatrix} + 1\begin{pmatrix} 56 & 67 & 63 \end{pmatrix} = \begin{pmatrix} 56 & 67 & 63 \end{pmatrix} \tag{2.5}$$
$$vC = 0\begin{pmatrix} 14 & 15 & 13 \end{pmatrix} + 0\begin{pmatrix} 10 & 20 & 10 \end{pmatrix} + 1\begin{pmatrix} 56 & 67 & 63 \end{pmatrix} = \begin{pmatrix} 56 & 67 & 63 \end{pmatrix} \tag{2.6}$$

portanto, $vABC = vC$. Notemos que $vAB = vC$ para todo $v \in \{0,1\}^3$ cuja segunda coluna seja 0, isto é, para

$$v \in \left\{\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 \end{pmatrix}\right\}$$

vale a igualdade, para qualquer outro vetor binário vale a diferença $vAB \neq vC$. Nesse exemplo a probabilidade de erro quando sorteamos v é $4/8 = 1/2$. Se escolhermos as coordenadas do vetor v dentre $\{0,1,2\}$ então 9 dos 27 vetores farão essa estratégia falhar, ou seja, a probabilidade de erro é $1/3$.

Instância : matrizes A, B, C quadradas de ordem n.
Resposta : *sim* se $AB = C$, caso contrário *não* com probabilidade de erro no máximo $1/2$.

1 $v \leftarrow_R \{0,1\}^n$;
2 **se** $(vA)B = vC$ **então responda** *sim*.
3 **senão responda** *não*.

Algoritmo 11: teste de produto de matrizes.

O produto $v(AB)$ é uma combinação linear das linhas de AB com coeficientes em v, assim como vC, como pode ser visto em (2.5) e (2.6), de modo que se $AB \neq C$ então há k linhas em que AB e C diferem, para algum $k \in \{1, \ldots, n\}$ e se as coordenadas do vetor v que são os coeficientes correspondentes a tais linhas forem 0, então teremos $v(AB) = vC$ e isso ocorre com probabilidade $(1/2)^k \leqslant 1/2$, pois $k > 0$.

Proposição 2.18 *Sejam* A, B, C *matrizes* $n \times n$. *Se* $AB \neq C$ *então* $\mathbb{P}_{v \in \{0,1\}^n}[(vA)B = vC] \leqslant 1/2$. □

A técnica a seguir (Mitzenmacher e Upfal, 2005) nos dá o cálculo da probabilidade de erro desse algoritmo e é útil em outras situações.

Princípio da decisão adiada Muitas vezes um experimento probabilístico é modelado como uma sequência de escolhas aleatórias independentes. O princípio da decisão adiada diz que podemos optar pela ordem com que as escolhas são feitas, adiando as escolhas mais relevantes para efeito de cálculos. Aqui, ao invés de uma escolha uniforme $v \in \{0,1\}^n$ podemos considerar uma escolha de cada coordenada de v de modo uniforme e independente em $\{0,1\}$.

Outra demonstração da proposição 2.18. Assumamos que cada coordenada de $v \in \{0,1\}^n$ é sorteada com probabilidade $1/2$ e independentemente uma das outras. Sejam A, B e C matrizes como acima e $D := AB - C$ matriz não nula. Queremos estimar $\mathbb{P}[vD = 0]$.

Se $D \neq 0$ e $vD = 0$, então existem ℓ e c tais que $d_{\ell,c} \neq 0$ com $\sum_{j=1}^{n} v_j d_{j,c} = 0$, assim podemos escrever

$$v_\ell = -\frac{1}{d_{\ell,c}} \sum_{\substack{j=1 \\ j \neq \ell}}^{n} v_j d_{j,c} \tag{2.7}$$

e se consideramos que cada coordenada de v foi sorteada independentemente, podemos assumir que v_i, para todo $i \neq \ell$, foi sorteado antes de v_ℓ de modo que o lado direito da igualdade (2.7) acima fica determinado e a probabilidade de sortear v_ℓ que satisfaça a igualdade é ou 0, caso o valor da direita não esteja em $\{0,1\}$, ou $1/2$ caso contrário. Portanto, $\mathbb{P}[vD = 0] = \mathbb{P}[vAB = vC] \leqslant \frac{1}{2}$. □

Desaleatorização Os bits aleatórios que um algoritmo usa para resolver um problema é um recurso que queremos otimizar, diminuir a quantidade utilizada sem penalizar muito os outros recursos, como por exemplo, o número de operações aritméticas realizadas. A isso chamamos de ***desaleatorização***. Por ora, vejamos uma versão do teste de produto de matrizes que usa menos bits aleatórios (devida a Kimbrel e Sinha, 1993).

Instância : matrizes A, B, C quadradas de ordem n.
Resposta : *sim* se $AB = C$, caso contrário *não* com probabilidade de erro no máximo $1/2$.

1 $x \leftarrow_R \{1, 2, \ldots, 2n\}$;
2 $v \leftarrow \begin{pmatrix} 1 & x & x^2 & \ldots & x^{n-1} \end{pmatrix}$;
3 **se** $(vA)B = vC$ **então responda** *sim*,
4 **senão responda** *não*.

Algoritmo 12: teste de Kimbrel–Sinha para produto de matrizes.

Vamos assumir que $AB \neq C$ e supor que existam n escolhas em $\{1, 2, \ldots, 2n\}$, denotadas $x_1, x_2, \ldots, x_n$, para as quais $(vA)B = vC$. Com essas escolhas formamos a matriz de Vandermonde

$$V = V(x_1, x_2, \ldots, x_n) = \begin{pmatrix} 1 & x_1 & {x_1}^2 & \ldots & {x_1}^{n-1} \\ 1 & x_2 & {x_2}^2 & \ldots & {x_2}^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & {x_n}^2 & \ldots & {x_n}^{n-1} \end{pmatrix}.$$

Se para cada linha $v(x_i) = \begin{pmatrix} 1 & x_i & x_i^2 & \ldots & x_i^{n-1} \end{pmatrix}$ dessa matriz vale que $(v(x_i) \cdot A) \cdot B = v(x_i) \cdot C$, então $VAB = VC$, o que implica $AB = C$ pois V é invertível, contrariando $AB \neq C$. Portanto, no caso em que $AB \neq C$, temos que o algoritmo

responde errado em no máximo $n-1$ escolhas de $x \in \{1,2,\ldots,2n\}$. A probabilidade de erro é a probabilidade de escolher uma das no máximo $n-1$ escolhas ruins descritas no parágrafo acima e é menor que $n/m \leqslant 1/2$.

O número de bits aleatórios utilizados é $\lceil \log_2(2n) \rceil$, necessários para x. O algoritmo de Freivalds usa n bits aleatórios, portanto, exponencialmente mais.

2.2.3 IDENTIDADE POLINOMIAL REVISITADA

O problema computacional que nos interessa é: dado um polinômio p de n variáveis sobre um corpo $\mathbb{F}$, determinar se p é identicamente nulo.

Para descrevermos o caso geral do problema de testar a igualdade de polinômios começamos com algumas definições para evitar ambiguidades. O termo "polinômio" tem, usualmente, dois significados. No cálculo, geralmente significa uma função, cujas variáveis podem ser instanciadas. Na álgebra, é simplesmente uma soma formal de termos, com cada termo sendo um produto de uma constante e um monômio. Um *polinômio* formal sobre um corpo $\mathbb{F}$ com indeterminadas $x_1, x_2, \ldots, x_n$ é uma expressão finita da forma

$$\sum_{i_1,i_2,\ldots,i_n \in \mathbb{N}} c_{i_1,i_2,\ldots,i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \tag{2.8}$$

em que $c_{i_1,i_2,\ldots,i_n} \in \mathbb{F}$ são os coeficientes do polinômio. O conjunto de todos esses polinômios é denotado por $\mathbb{F}[x_1, x_2, \ldots, x_n]$. O *grau total* do polinômio f dado pela equação (2.8), denotado por ∂f, é o maior valor de $i_1 + i_2 + \cdots + i_n$ dentre todos os índice para os quais $c_{i_1,i_2,\ldots,i_n} \neq 0$ e o *grau em* x_j, denotado $\partial_{x_j} f$, é o maior valor de i_j tal que $c_{i_1,i_2,\ldots,i_n} \neq 0$.

Quando usamos o significado funcional, polinômio não nulo significa que a função não é identicamente igual a zero. Quando usamos a definição algébrica, polinômio não nulo significa que pelo menos um coeficiente não é igual a zero. Essa distinção muitas vezes passa desapercebida porque se o polinômio é avaliado em um corpo infinito, como os números reais ou complexos, ou mesmo num domínio de integridade como $\mathbb{Z}$, então um polinômio como combinações lineares de termos com coeficientes diferentes de zero induz uma função que não é identicamente zero e vice-versa. Nos corpos finitos pode acontecer de polinômios distintos determinarem a mesma função polinomial. Por exemplo, no corpo finito com 7 elementos os polinômios 1 e $x^7 - x + 1$ são diferentes, mas como funções de $\mathbb{F}_7$ em $\mathbb{F}_7$ são iguais pois $x^7 \equiv x \pmod{n}$ pelo Pequeno Teorema de Fermat (teorema 2.41, página 68). No corpo finito com 3 elementos os polinômios 0 e $x^3 - x$ são diferentes, mas como funções de $\mathbb{F}_3$ em $\mathbb{F}_3$ são iguais também pelo Pequeno Teorema de Fermat (ou, porque $x^3 - x = x(x-1)(x-2)$).

Temos de fato dois problemas computacionais: dado um polinômio p, (1) decidir se p vale zero, como função, em todo elemento do corpo e (2) decidir se p na forma canônica tem todos os coeficientes nulos. Certamente, (2) implica (1) mas a inversa nem sempre vale. Em corpos infinitos como $\mathbb{Q}$, $\mathbb{R}$ e $\mathbb{C}$ e em corpos finitos que contenham uma quantidade suficientemente grande de elementos ($|\mathbb{F}| > \partial p$) vale que (1) implica (2) e isso segue do teorema 2.19 abaixo. Portanto, sob a hipótese de que $|\mathbb{F}|$ é suficientemente grande os problemas (1) e (2) descritos acima são realmente equivalentes.

Vamos fixar que para dois polinômios $p, q \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ escrevemos $p = q$ se os polinômios são iguais *quando escritos* na forma canônica, como na equação (2.8). Por exemplo, a seguinte identidade entre expressões algébricas que correspondem a polinômios é verdadeira (ambas são o determinante da matriz de Vandermonde)

$$\sum_{\sigma \in \mathbb{S}_n} \operatorname{sinal}\left(\prod_{1 \leqslant i < j \leqslant n} (\sigma(j) - \sigma(i)) \right) \prod_{i=1}^{n} x_i^{\sigma(i)-1} = \prod_{1 \leqslant i < j \leqslant n} (x_j - x_i)$$

onde $\mathbb{S}_n$ é o conjunto de todas as permutações $\sigma \colon \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$.

Quando nos referirmos a uma instância do problema computacional, "dado p" significa que

- ou polinômio é dado como uma *caixa-preta* para a qual fornecemos $a \in \mathbb{F}^n$ e recebemos $p(a)$;

- ou é dado explicitamente por uma expressão como um determinante, por exemplo, ou, mais concretamente, como um circuito aritmético, que definiremos no capítulo 5. Nesse caso, faz parte do algoritmo avaliar $p(a)$.

Claramente, polinômios dados na representação canônica tornam o problema trivial. Ademais, obter o polinômio na forma canônica a partir do circuito aritmético ou reescrevendo uma expressão algébrica está fora de questão pois é uma tarefa que pode ser muito custosa se procuramos por um algoritmo eficiente para o problema. Por exemplo, o polinômio $\prod_{1\leqslant i<j\leqslant n}(x_i - x_j)$ escrito como combinação linear de monômios resulta numa expressão da forma

$$\sum z_1 z_2 \cdots z_{\binom{n-1}{2}} z_{\binom{n}{2}}$$

onde $z \in \{x_i, x_j\}$ para cada par $1 \leqslant i < j \leqslant \binom{n}{2}$, com a soma de 2^n parcelas. Cada parcela tem tamanho da ordem de n^2 termos, totalizado uma expressão com tamanho da ordem de $n^2 2^n$, contra a expressão original com tamanho da ordem de n^2 termos.

Os dois modelos derivados da forma como é dado o polinômio são bem estudados porque o problema da identidade polinomial é muito importante. O primeiro é o chamado *modelo caixa preta* e o segundo é chamado de modelo *caixa branca*. A vantagem da representação implícita é que o polinômio pode ser avaliado eficientemente em qualquer entrada, desde que as operações do corpo possam ser feitas eficientemente, mas o problema é mais fácil no modelo caixa branca (Shpilka e Yehudayoff, 2010).

Problema computacional do teste de identidade polinomial (PIT):
Instância : circuito aritmético que computa um polinômio p de n variáveis sobre um corpo $\mathbb{F}$.
Resposta : *sim* se p é identicamente nulo, *não* caso contrário.

Problema 1 (*Identidade polinomial*). Existe para PIT um algoritmo determinístico que executa um número de operações em $\mathbb{F}$ polinomial no tamanho do circuito aritmético?

A estratégia do algoritmo probabilístico para esse problema é idêntica ao caso de uma variável, sorteamos n números e avaliamos o polinômio nessa n-upla. Notemos que não há uma generalização imediata do Teorema Fundamental da Álgebra pois, por exemplo, um polinômio sobre um corpo como $\mathbb{Q}$ e com várias variáveis pode ter um número infinito de raízes, como é o caso de $x_1 x_2$. A estratégia é baseada no seguinte teorema.

Teorema 2.19 (teorema de Schwartz–Zippel) *Sejam $\mathbb{F}$ um corpo, $p \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ um polinômio não nulo com grau total d e $S \subset \mathbb{F}$ finito. Então*

$$\mathbb{P}_{(x_1,\ldots,x_n)\in S^n}[p(x_1,\ldots,x_n) = 0] \leqslant \frac{d}{|S|}.$$

Esse resultado foi descoberto várias vezes, de modo independente (DeMillo e Lipton, 1978; Schwartz, 1979; Zippel, 1979, dentre outros). Com o teorema 2.19 nós podemos resolver probabilisticamente o problema de identidade polinomial. Suponha que nos seja dado um polinômio $p(x_1,\ldots,x_n)$ de grau total $d < 2|\mathbb{F}| \leqslant |S|$. O algoritmo sorteia $r_1,\ldots,r_n$ em $S \subset \mathbb{F}$ (ou em $\mathbb{F}$) finito e suficientemente grande, computa $p(r_1,\ldots,r_n)$ e decide, de modo que a probabilidade de erro é no máximo $d/|S| \leqslant 1/2$.

Seja $p \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ um polinômio de grau positivo e $S \subset \mathbb{F}$ finito. Se $n = 1$, então p tem no máximo ∂p raízes em S pois em $\mathbb{F}[x_1]$, que é um domínio de fatoração única, vale o teorema da divisão de modo que se $r \in \mathbb{F}$ é raiz de p, então existe $q \in \mathbb{F}[x_1]$ tal que $p(x) = (x - r)q(x)$. Portanto são no máximo ∂p raízes.

Se $n = 2$, tome $k = \partial_{x_2} p$ (se $k = 0$ então caímos no caso anterior). Podemos reescrever p como

$$p(x_1, x_2) = a_k(x_1) \cdot x_2^k + a_{k-1}(x_1) \cdot x_2^{k-1} + \cdots + a_0 \cdot (x_1)$$

com polinômios $a_i \in \mathbb{F}[x_1]$ para todo i. Tome o conjunto das raízes de p em S^2

$$R_p := \{(x_1, x_2) \in S^2 \colon p(x_1, x_2) = 0\}$$

e o conjunto das raízes r de $a_k(x_1)$ em R_p

$$\mathsf{R}_{a_k} \coloneqq \{(r, x_2) \in \mathsf{R}_p \colon a_k(r) = 0\}.$$

Como a_k é polinômio em uma variável há $\leqslant \partial a_k$ raízes em $\mathbb{F}$, logo há $\leqslant \partial a_k \cdot |\mathsf{S}|$ pares em R_{a_k}, isto é, $|\mathsf{R}_{a_k}| \leqslant \partial a_k \cdot |\mathsf{S}|$. Agora, os pares em $\mathsf{R}_p \setminus \mathsf{R}_{a_k}$ são aqueles que anulam p, mas a primeira coordenada não anula a_k. Assim se $(r_1, r_2) \in \mathsf{R}_p \setminus \mathsf{R}_{a_k}$, então $p(r_1, x_2) \in \mathbb{F}[x_2]$ e $\partial p(r_1, x_2) = k$, logo, é anulado para $\leqslant k$ valores $x_2 \in \mathsf{S}$. Daí, $|\mathsf{R}_p \setminus \mathsf{R}_{a_k}| \leqslant k|\mathsf{S}|$ e

$$|\mathsf{R}_p| = |\mathsf{R}_{a_k}| + |\mathsf{R}_p \setminus \mathsf{R}_{a_k}| \leqslant (\partial a_k + k)|\mathsf{S}| \leqslant \partial p \cdot |\mathsf{S}|$$

Essa é uma boa estimativa, o polinômio $(x_1 - x_2)^2 - 1$ de grau total 2 no corpo $\mathbb{F}_3$ tem raízes $(0,1)$, $(1,0)$, $(2,1)$, $(1,2)$, $(0,2)$ e $(2,0)$, ou seja, $6 = 2 \cdot 3 = 2 \cdot |\mathbb{F}_3|$ raízes em $\mathbb{F}_3$.

O argumento que acabamos de descrever é indutivo, provamos o caso de duas variáveis usando o caso de uma variável. Podemos, sem muita dificuldade, generalizar o passo acima para provar o seguinte resultado.

Lema 2.20 (lema de Schwartz) *Sejam $\mathbb{F}$ um corpo, $\mathsf{S} \subset \mathbb{F}$ finito, $n \geqslant 1$, e $p \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ um polinômio não nulo. Então, p tem no máximo $\partial p \cdot |\mathsf{S}|^{n-1}$ raízes em S^n.*

Demonstração. Por indução em $n \geqslant 1$. Pela dedução acima a base, $n = 1$, vale. basta verificarmos o passo da indução.

Assumimos que para $n \geqslant 2$, todo $q \in \mathbb{F}[x_1, \ldots, x_{n-1}]$ tem no máximo $\partial q \cdot |\mathsf{S}|^{n-2}$ raízes em S^{n-1}. Vamos mostrar que $p \in \mathbb{F}[x_1, \ldots, x_n]$ tem no máximo $\partial p \cdot |\mathsf{S}|^{n-1}$ raízes em S^{n-1}.

Suponhamos, sem perda de generalidade, que $\partial_{x_n} p = k > 0$ e com isso podemos escrever

$$f(x_1, \ldots, x_{n-1}, x_n) = \sum_{j=0}^{k} g_j(x_1, \ldots, x_{n-1}) \cdot x_n^j.$$

Definimos o conjunto R_p das raízes de p em S^n

$$\mathsf{R}_p \coloneqq \{(x_1, \ldots, x_{n-1}, x_n) \in \mathsf{S}^n \colon p(x_1, \ldots, x_{n-1}, x_n) = 0\}$$

e definimos o conjunto R_{g_k} das raízes de $g_k(x_1, \ldots, x_{n-1})$ em R_p

$$\mathsf{R}_{g_k} \coloneqq \{(a_1, \ldots, a_{n-1}, x_n) \in \mathsf{R}_f \colon g_k(a_1, \ldots, a_{n-1}) = 0\}.$$

Pela hipótese indutiva g_k tem no máximo $\partial g_k \cdot |\mathsf{S}|^{n-2}$ raízes em S^{n-1}, portanto, $|\mathsf{R}_{g_k}| \leqslant \partial g_k \cdot |\mathsf{S}|^{n-1}$. Em $\mathsf{R}_p \setminus \mathsf{R}_{g_k}$ temos os pontos $(a_1, \ldots, a_{n-1}, x_n) \in \mathsf{R}_p$ tais que $g_k(a_1, \ldots, a_{n-1}) \neq 0$, portanto $p(a_1, \ldots, a_{n-1}, x_n)$ é um polinômio em x_n de grau k, logo tem $\leqslant k$ raízes. Daí, $|\mathsf{R}_p \setminus \mathsf{R}_{g_k}| \leqslant k|\mathsf{S}|^{n-1}$. Portanto,

$$|\mathsf{R}_p| = |\mathsf{R}_{a_k}| + |\mathsf{R}_p \setminus \mathsf{R}_{a_k}| \leqslant (\partial g_k + k)|\mathsf{S}|^{n-1} \leqslant \partial p \cdot |\mathsf{S}|^{n-1} \tag{2.9}$$

é a estimativa procurada para o número de raízes de p em S^n. □

Demonstração do teorema 2.19. Se $\mathbb{F}$, S, n e p são como no enunciado

$$\mathop{\mathbb{P}}_{(x_1,\ldots,x_n)\in\mathsf{S}^n}[p(x_1, \ldots, x_n) = 0] \leqslant \frac{|\mathsf{R}_p|}{|\mathsf{S}|^n}$$

e o teorema segue de (2.9). □

O algoritmo para polinômios de várias variáveis é uma adaptação simples do algoritmo 1 e é como segue.

Instância : $d > 0$ e $f(x_1, \ldots, x_n)$ de grau total no máximo d.
Resposta : *não* se $f \neq 0$, caso contrário *sim* com probabilidade de erro no máximo 1/4.

1 **para cada** $i \in \{0, \ldots, n-1\}$ **faça** $x_i \leftarrow_R \{1, 2, \ldots, 4d\}$;
2 **se** $f(x_1, x_2, \ldots, x_n) \neq 0$ **então responda** *não*.
3 **senão responda** *sim*.

Algoritmo 14: identidade entre polinômios.

A probabilidade de erro do algoritmo 14 segue do teorema de Schwartz–Zippel, para S e f como acima, um algoritmo que escolhe aleatoriamente $x_1, \ldots, x_n$ em S e decide "$f = 0$?" baseado no teste $f(x_1, \ldots, x_n) = 0$ erra com probabilidade no máximo $d/|S| = 1/4$. Repetindo k vezes o algoritmo, se $f \neq 0$ então o algoritmo responde *sim* somente se nas k iterações (independentes) foi sorteada uma raiz de f cuja probabilidade é

$$\mathbb{P}[\text{erro}] \leqslant \left(\frac{1}{4}\right)^k.$$

EXERCÍCIO 2.21. Qual é a probabilidade de resposta errada em k repetições (independentes) do algoritmo se as escolhas aleatórias são garantidas ser sem repetição.

2.2.4 RAÍZES PRIMITIVAS

Um grupo multiplicativo $(G, \cdot)$ é *cíclico* se possui um *gerador* g, isto é, para todo $h \in G$ existe um inteiro positivo l tal que $g^l = h$. Nesses casos, o expoente l é o *logaritmo discreto* de h em G na base g. Vários algoritmos importantes na criptografia de chave pública baseiam sua segurança na suposição de que o problema do logaritmo discreto com G e g cuidadosamente escolhidos não tem algoritmo eficiente (veja o exemplo 5.2, página 136). Em Criptografia é frequente o uso do grupo multiplicativo dos inteiros módulo um primo p como, por exemplo, no protocolo Diffie–Hellman–Merkle para troca pública de chaves criptográficas (Diffie e Hellman, 1976), onde precisamos determinar de modo eficiente um gerador desse grupo (veja o protocolo na página 137).

Um elemento que gera o grupo multiplicativo dos inteiros invertíveis módulo n, denotado $\mathbb{Z}_n^*$, é chamado de **raiz primitiva módulo** n. É um resultado conhecido que raízes primitivas existem se, e só se, $n = 1, 2, 4, p^k, 2p^k$ com p primo e $k \in \mathbb{N}$. O problem que estamos interessado aqui é o seguinte.

Problema computacional da raiz primitiva módulo p:

Instância : $p > 2$ primo.
Resposta : uma raiz primitiva módulo p.

Não se conhece algoritmo eficiente para determinar uma raiz primitiva módulo p a menos que seja dado a fatoração de $p - 1$. Lembramos que, atualmente, não é conhecido algoritmo eficiente para o problema da fatoração, problema 5, página 136.

> *Problema* 2 (*Raiz primitiva módulo p*). Dado um primo p é possível determinar em tempo polinomial em $\log_2 p$ uma raiz primitiva módulo p?

A seguir, $n \geqslant 2$ é inteiro, $\mathbb{Z}_n$ denota o conjunto das classes dos restos da divisão identificado com $\{0, 1, \ldots, n-1\}$ que munido da soma e do produto módulo n é um anel comutativo com unidade. Um elemento a do anel tem inverso multiplicativo se, e só se, $\text{mdc}(a, n) = 1$ e $\mathbb{Z}_n^*$ denota o conjunto $\{a \in \mathbb{Z}_n : \text{mdc}(a, n) = 1\}$ que munido do produto módulo n é um grupo comutativo, chamado grupo das unidades do anel $\mathbb{Z}_n$.

A **ordem (multiplicativa)** de $a \in \mathbb{Z}_n^*$, denotada $\text{ord}_*(a)$, é o menor inteiro positivo k tal que $a^k \equiv 1 \pmod{n}$.

O algoritmo trivial para determinar a ordem de um elemento $a \in \mathbb{Z}_n^*$ calcula todas as potências $a^k \pmod n$, o que é inviável se $|\mathbb{Z}_n^*|$ é grande como, por exemplo, no caso das aplicações em criptografia. Mesmo $\varphi(n) := |\mathbb{Z}_n^*|$ é difícil de computar quando n não é primo ou potência de primo. A função $\varphi(n)$ é conhecida como função totiente de Euler, voltaremos a ela no final da seção.

EXERCÍCIO 2.22. Prove que, se $a \in \mathbb{Z}_n^*$ e sua ordem é k, então

1. Teorema de Euler: $a^{\varphi(n)} \equiv 1 \pmod n$;
2. $a^m \equiv a^\ell \pmod n$ se e só se $m \equiv \ell \pmod k$. Em particular, se $a^m \equiv 1 \pmod n$ então k divide m;
3. $\mathrm{ord}_*(a^m) = k/\mathrm{mdc}(m,k)$. (Dica: verifique que, para quaisquer inteiros a e x diferentes de 0 vale que $ax \equiv 0 \pmod n$ se, e só se, $x \equiv 0 \pmod{n/\mathrm{mdc}(a,n)}$.)

O seguinte resultado deu origem ao algoritmo probabilístico 16 que veremos abaixo.

Lema 2.23 *Seja $a \in \mathbb{Z}_n^*$ e $m = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$ positivo tal que $a^m \equiv 1 \pmod n$. Então*

$$\mathrm{ord}_*(a) = \prod_{i=1}^{k} p_i^{m_i - f_i},$$

onde f_i é o maior inteiro não negativo tal que $a^{m/p_i^{f_i}} \equiv 1 \pmod n$, para todo $i \in \{1,2,\ldots,k\}$.

Demonstração. Seja $o = \mathrm{ord}_*(a)$. Pelo exercício 2.22 temos que o divide m, portanto, $o = p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$, com $0 \leqslant r_i \leqslant m_i$ para todo i. Para determinar r_1 tomemos a sequência

$$a^{m/p_1^{m_1}},\ a^{m/p_1^{m_1-1}},\ a^{m/p_1^{m_1-2}},\ \ldots,\ a^{m/p_1}, a^m$$

módulo n e seja f_1 o maior inteiro não negativo tal que $a^{m/p_1^{f_1}} \equiv 1 \pmod n$ (a primeira ocorrência de 1). Então temos

$$a^{m/p_1^{f_1}} \equiv 1 \pmod n \text{ e } a^{m/p_1^{f_1+1}} \not\equiv 1 \pmod n \text{ ou } f_1 = m_1.$$

De $a^{m/p_1^{f_1}} \equiv 1 \pmod n$ temos que o divide $m/p_1^{f_1}$, ou seja,

$$o \text{ divide } p_1^{m_1-f_1} p_2^{m_2} \cdots p_k^{m_k}$$

mas de $a^{m/p_1^{f_1+1}} \not\equiv 1 \pmod n$ temos que o não divide $m/p_1^{f_1+1}$, ou seja

$$o \text{ não divide } p_1^{m_1-f_1-1} p_2^{m_2} \cdots p_k^{m_k},$$

isso só pode ocorrer se $r_1 = m_1 - f_1$. Analogamente, $r_i = m_i - f_i$ para todo $i \in \{2,\ldots,k\}$. □

Corolário 2.24 *Se $a^m \equiv 1 \pmod n$ e $a^{m/p} \not\equiv 1 \pmod n$ para todo primo p divisor m, então a tem ordem m.* □

EXERCÍCIO 2.25. Prove que se p é primo e no $\mathbb{Z}_p^*$ a ordem de a_1 é n_1, a ordem de a_2 é n_2 e $\mathrm{mdc}(n_1,n_2) = 1$ então a ordem de $a_1 a_2 \in \mathbb{Z}_p^*$ é $n_1 n_2$.

Exemplo 2.26. O $\mathbb{Z}_{11}^*$ é um grupo de ordem $10 = 2 \cdot 5$. Pelo corolário 2.24, se $a^{10} \equiv 1 \pmod{11}$ (que vale pelo Teorema de Euler) e $a^2 \not\equiv 1 \pmod{11}$ e $a^5 \not\equiv 1 \pmod{11}$, então a tem ordem 10, logo é um gerador.

a	1	2	3	4	5	6	7	8	9	10
$a^2 \bmod 11$	1	4	9	5	3	3	5	9	4	1
$a^5 \bmod 11$	1	10	1	1	1	10	10	10	1	10

Os geradores são $2,6,7,8$. Se usarmos o fato de que, certamente, 1 e 10 não são geradores, então a probabilidade de escolher um gerador num sorteio em $\{2,\ldots,9\}$ é $1/2$. Se escolhemos um elemento uniformemente e repetimos a escolha de modo independente, o número médio (ponderado pelas probabilidades) de rodadas até sair um gerador é $1\frac{1}{2}+2\frac{1}{4}+3\frac{1}{8}+\cdots=2$. ◊

Proposição 2.27 *Para $p>2$ primo, o $\mathbb{Z}_p^*$ admite $\varphi(p-1)$ geradores.*

Demonstração. Se $a\in\mathbb{Z}_p^*$ é um gerador, $\mathbb{Z}_p^*=\{a \bmod p, a^2 \bmod p,\ldots,a^{p-1} \bmod p\}$ e a ordem de $a^k \bmod p$ é, pelo exercício 2.22, $(p-1)/\mathrm{mdc}(k,p-1)$ logo a^k é gerador se, e só se, $\mathrm{mdc}(k,p-1)=1$. Com isso, concluímos que são $\varphi(p-1)$ geradores de $\mathbb{Z}_p^*$ □

Portanto, $\varphi(p-1)/(p-1)$ é a probabilidade de sortear um gerador no $\mathbb{Z}_p^*$. Podemos usar os limitantes conhecidos para a função φ (veja Bach e Shallit, 1996, teorema 8.8.7) para ter, por exemplo,

$$p=\frac{\varphi(n)}{n}>\frac{1}{6\log\log(n)} \tag{2.10}$$

para todo $n>10$. O número médio (como na equação (2.4)) de sorteios de números do $\mathbb{Z}_p$ até sair um gerador é $\sum_{k\geqslant 1}k(1-p)^{k-1}p<6\log\log(n)$.

O resultado a seguir mostra que uma escolha uniforme em $\mathbb{Z}_p^*$ implica numa escolha uniforme nos elemento do $\mathbb{Z}_p^*$ da forma $x^{(p-1)/p_i}$ para p_i primo que divide $p-1$.

Proposição 2.28 *Sejam p um primo e p_i um fator primo de $p-1$. Se $f:\mathbb{Z}_p^*\to\mathbb{Z}_p^*$ é o homomorfismo de grupos definido por $f(x)=x^{(p-1)/p_i} \bmod p$, então $|f^{-1}(a)|=|f^{-1}(b)|$ para quaisquer $a\neq b\in\mathrm{Im}(f)$; ademais $|\mathrm{Im}(f)|=p_i$.*

Demonstração. Sejam r uma raiz primitiva e a e b elementos distintos do $\mathbb{Z}_p^*$. Sejam $k,\ell\in\{1,2,\ldots,p-1\}$ distintos tais que $b=r^k$ e $a=r^\ell$. Pelo exercício 2.22 temos $f(b)=f(a)$ se e só se $k\frac{p-1}{p_i}\equiv\ell\frac{p-1}{p_i} \pmod{p-1}$. De $1\leqslant k,\ell\leqslant p-1$ distintos temos $k\not\equiv\ell \pmod{p-1}$. Portanto $k\equiv\ell \pmod{p_i}$.

Agora, fixado k a quantidade de inteiros ℓ com $k\equiv\ell \pmod{p_i}$ é $(p-1)/p_i$, logo, a pré-imagem $f^{-1}(b)$, para qualquer b, tem cardinalidade $(p-1)/p_i$. Finalmente $|\mathrm{Im}(f)|=(p-1)/|f^{-1}(b)|=p_i$. □

Corolário 2.29 *Com as hipóteses acima*

$$\mathop{\mathbb{P}}_{x\in\mathbb{Z}_p^*}\left[x^{\frac{p-1}{p_i}}\equiv 1 \pmod p\right]=\frac{1}{p_i}$$

para todo p_i fator primo de $p-1$. □

O algoritmo 16 abaixo já era conhecido de Gauss que, como exemplo, determinou um gerador de $\mathbb{Z}_{73}^*$ no seu trabalho *Disquisitiones Arithmeticæ*.

Instância : um primo p e a fatoração $p-1=p_1^{m_1}p_2^{m_2}\cdots p_k^{m_k}$.
Resposta : raiz primitiva módulo p.

1 **para** *i de* 1 *até k* **faça**
2 **repita**
3 $a\leftarrow_{\mathrm{R}}\{2,3,\ldots,p-1\}$;
4 $b\leftarrow a^{(p-1)/p_i} \bmod p$;
5 **até que** $b\neq 1$;
6 $q_i\leftarrow a^{(p-1)/p_i^{m_i}} \bmod p$;
7 **responda** $\prod_{i=1}^k q_i \bmod p$.

Algoritmo 16: raiz primitiva.

A prova de que o algoritmo 16 está correto segue da observação de que no final da i-ésima iteração do **para** na linha 1 vale que

$$q_i^{p_i^{m_i}} \equiv 1 \pmod{p} \quad \text{e} \quad q_i^{p_i^{m_i-1}} \not\equiv 1 \pmod{p}$$

e nesse caso, pelo corolário 2.24 a ordem de q_i no $\mathbb{Z}_p^*$ é $p_i^{m_i}$. Usando uma generalização natural do exercício 2.25 para um produto com mais que 2 fatores, temos que a ordem de $q_1 q_2 \cdots q_k$ em $\mathbb{Z}_p^*$ é $p-1$, portanto, é um gerador do grupo.

O tempo de execução do algoritmo depende do número de rodadas do **repita** na linha 2 que por sua vez depende dos sorteios. Para cada i no laço da linha 1 a probabilidade de sair do laço na linha 2 é $1-(1/p_i)$ pelo corolário acima, portanto como em (2.4), o número médio de rodadas é $1/(1-(1/p_i)) = p_i/(p_i-1) \leqslant 2$. As exponenciações na linha 4 podem ser feita em tempo $O(\log^3 p)$ (algoritmo 7, página 47). Assim, o laço da linha 2 tem custo médio $O(\log^3 p)$. As exponenciações na linha 6 também são feitas em tempo $O(\log^3 p)$. Assim, o laço da linha 1 tem custo final $O(k \cdot \log^3 p)$ em média. Os $k-1$ produtos módulo p na linha 7 envolvem números da ordem de $\log p$ dígitos, o que custa $O(k \cdot \log^2 p)$. Portanto, em média, o tempo de execução do algoritmo é $O(k \cdot \log^3 p)$.

O parâmetro k é a quantidade de divisores primos distintos de $p-1$, usualmente denotada nos livros de teoria dos números por $\omega(p-1)$, cuja ordem de grandeza é (veja Bach e Shallit, 1996, teorema 8.8.10)

$$\omega(n) = O(\log n / \log\log n). \tag{2.11}$$

Dito isso, podemos concluir o seguinte resultado.

Teorema 2.30 *O tempo médio de execução para o algoritmo 16 é* $O(\log^4 p / \log\log p)$. □

A função φ de Euler A função φ de Euler associa a cada inteiro positivo n a quantidade de inteiros positivos menores que n que são coprimos[3] com n

$$\varphi(n) := |\{a \in \mathbb{N} : \operatorname{mdc}(a,n) = 1 \text{ e } 1 \leqslant a \leqslant n\}| = |\mathbb{Z}_n^*|.$$

Decorre da definição que se p é primo então $\varphi(p) = p-1$. Para potências de primo temos que dentre $1, 2, \ldots, p^k$ não são coprimos com p^k aquele que têm p como fator primo, a saber $p, 2p, 3p, \ldots, p^{k-1}p$, portanto $p^k - p^{k-1}$ são coprimos, isto é,

$$\varphi(p^k) = p^k\left(1 - \frac{1}{p}\right).$$

Para determinarmos o valor da função de Euler nos naturais que não são potência de primo o seguinte é fundamental: a função φ é multiplicativa.

Teorema 2.31 *Se* $n, m \in \mathbb{Z}^+$ *são coprimos então* $\varphi(nm) = \varphi(n)\varphi(m)$.

Demonstração. O caso $m = 1$ ou $n = 1$ é imediato. Sejam $n, m > 1$ inteiros. Para todo inteiro a vale (verifique)

$$\operatorname{mdc}(a, nm) = 1 \Leftrightarrow \operatorname{mdc}(a,n) = 1 \text{ e } \operatorname{mdc}(a,m) = 1$$

Desse modo, $\varphi(nm)$ é a quantidade de naturais entre 1 e nm que são coprimos com n e com m concomitantemente. Em

$$\begin{array}{ccccccc}
1 & 2 & \ldots & i & \ldots & m \\
m+1 & m+2 & \ldots & m+i & \ldots & 2m \\
2m+1 & 2m+2 & \ldots & 2m+i & \ldots & 3m \\
\vdots & \vdots & \ldots & \vdots & \ldots & \vdots \\
(n-1)m+1 & (n-1)m+2 & \ldots & (n-1)m+i & \ldots & nm
\end{array}$$

[3] a e n são coprimos se $\operatorname{mdc}(a,n) = 1$.

há $\varphi(m)\varphi(n)$ coprimos com n e m pois: há na tabela acima $\varphi(m)$ colunas que começam com um inteiro coprimo com m. Se um divisor de m divide i, então divide todos os números na coluna i. Portanto, os coprimos com m em $\{1,\ldots,nm\}$ aparecem nas $\varphi(m)$ colunas dos coprimos com m.

Os inteiros $\{1,\ldots,nm\}$ com m e n também aparecem nas $\varphi(m)$ colunas dos coprimos com m. Porém, cada uma dessas colunas é um sistema completo de restos módulo n, portanto, há em cada $\varphi(n)$ coprimos com n. Assim, há $\varphi(m)\varphi(n)$ coprimos com n e m. □

EXERCÍCIO 2.32. Use indução para mostrar que se $\text{mdc}(n_i, n_j) = 1$ para todo $i \neq j$ então $\varphi(n_1 n_2 \cdots n_k) = \varphi(n_1)\varphi(n_2)\cdots\varphi(n_k)$.

Corolário 2.33 *Se $n = p_1^{m_1} \cdots p_k^{m_k}$ é a fatoração canônica de n então*

$$\varphi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

para todo inteiro $n > 1$. □

EXERCÍCIO 2.34. Mostre que $\varphi(n) = n - 1$ se e só se n primo.

Então, concluímos que é fácil determinar $|\mathbb{Z}_n^*|$ se temos a fatoração em primos de n. Por outro lado, tomando, por exemplo, o caso $n = pq$ temos $\varphi(n) = (p-1)(q-1)$ logo, de n e $\varphi(n)$ temos $p+q = n+1-\varphi(n)$. Também, $(p-q)^2 = (p+q)^2 - 4n$. De $p+q$ e $p-q$ é fácil determinar p e q. Esse fato parece indicar que o problema de determinar $\varphi(n)$ é tão difícil computacionalmente quanto fatoração de n.

> *Problema* 3 (*Problema da função φ de Euler*). Dado um inteiro $n > 1$, é possível determinar $|\mathbb{Z}_n^*|$ em tempo polinomial em $\log_2 n$?

O teorema de Euler Para finalizar essa seção, vamos esboçar uma demonstração do teorema de Euler (exercício 2.22): *se a e $n > 0$ são inteiros com* $\text{mdc}(a, n) = 1$, *então* $a^{\varphi(n)} \equiv 1 \pmod{n}$.

Se os elementos de $R := \{r_1, r_2, \ldots, r_{\varphi(n)}\}$ são coprimos com n e incongruentes módulo n entre si, então para todo a coprimo com n também são coprimos com n e incongruentes módulo n entre si os inteiros de $aR = \{ar_1, ar_2, \ldots, ar_{\varphi(n)}\}$. De fato, $ar_i \equiv ar_j \pmod{n}$ implica $r_i \equiv r_j \pmod{n}$ pois a tem inverso multiplicativo módulo n.

Todo inteiro coprimo com n é congruente a um, e só um elemento de R, logo isso vala para os elementos de aR. Assim $\prod_i ar_i \equiv \prod_i r_i \pmod{n}$, portanto, $a^{\varphi(n)} \prod_i r_i \equiv \prod_i r_i \pmod{n}$ e como são todos invertíveis $a^{\varphi(n)} \equiv 1 \pmod{n}$.

2.2.5 POLINÔMIOS IRREDUTÍVEIS E ARITMÉTICA EM CORPOS FINITOS

Os corpos finitos têm um papel importante em Computação. Veremos adiante neste texto várias construções que usam tal estrutura algébrica para, por exemplo, desaleatorizar algoritmos; também são úteis na Teoria dos Códigos, em Complexidade Computacional e na Criptografia. Nos algoritmos sobre esses corpos precisamos de uma descrição explícita deles, assim como das operações aritméticas do corpo, de fato assumimos que a aritmética é dada e a medida de custo de um algoritmo é dada em função do número de operações no corpo. Veremos abaixo como fazer isso e mais detalhes do que damos aqui são encontrados em Gathen e Gerhard (2013) e Lidl e Niederreiter (1997).

Para todo primo p existe um corpo com p^n elementos, para todo $n \in \mathbb{N}$, que é único a menos de isomorfismos e é denotado por $\mathbb{F}_{p^n}$. Reciprocamente, todo corpo finito tem p^n elementos para algum p primo e algum n natural. No caso $n = 1$ temos os corpos mais simples dados por $\mathbb{Z}_p$ com p primo e as operações módulo p, mas em geral $\mathbb{F}_{p^n} \neq \mathbb{Z}_{p^n}$ se $n > 1$.

Denotamos por $\mathbb{Z}_n[x]$, para $n \geqslant 2$, o conjunto dos polinômios

$$a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0$$

com indeterminada x e com coeficientes $a_0, a_1, \ldots$ em $\mathbb{Z}_n$ que com a soma e o produto usual de polinômios é um anel comutativo com unidade 1. O coeficiente a_0 de x^0 é o **termo constante** e o coeficiente a_m da maior potência de x é o **coeficiente principal** do polinômio. O grau de um polinômio não nulo f, denotado ∂f, é o maior expoente de x com coeficiente *não nulo*, polinômio de grau 0 é chamado de **polinômio constante** e o grau do polinômio nulo não está definido. Um polinômio é **mônico** se o coeficiente principal é 1.

EXERCÍCIO 2.35 (divisão com resto para polinômios). Sejam $n \geqslant 2$ inteiro e $h, f \in \mathbb{Z}_n[x]$ polinômios com h mônico. Prove que existem únicos $q, r \in \mathbb{Z}_n[x]$ com $f = hq + r$ e $\partial r < \partial h$. Verifique que tal divisão pode ser realizada com $O(\partial f \partial h)$ operações do $\mathbb{Z}_n$.

No caso $r = 0$ dizemos que h **divide** f. Agora, se $f = h \cdot q$ com $0 < \partial h < \partial f$ então h é um **divisor próprio** de f.

No caso de polinômios com coeficientes sobre um corpo $\mathbb{F}$ o resultado do exercício acima vale para qualquer $h \in \mathbb{F}[x]$ não nulo. Um polinômio $f \in \mathbb{F}[x]$ com grau positivo é dito **irredutível** em $\mathbb{F}[x]$ se em toda fatoração $f = h \cdot q$ ou h ou q é polinômio constante. Em $\mathbb{F}[x]$ vale ainda a **fatoração única**: *todo polinômio não nulo $f \in \mathbb{F}[x]$ pode ser escrito de forma única a menos da ordem dos fatores como um produto $a \cdot h_1 \cdot h_2 \cdots h_s$ com $a \in \mathbb{F}$ e $h_i \in \mathbb{F}[x]$ mônicos, irredutíveis e de grau positivo.* Notemos que $\mathbb{F}[x]$ não é corpo pois, por exemplo, o polinômio x não tem inverso multiplicativo.

Um aplicação importante dos polinômios irredutíveis é a construção explícita de corpos finitos. Essas construções são baseadas no seguinte resultado. Quando $h \in \mathbb{F}_p[x]$ é um polinômio mônico e irredutível o conjunto $\{g \in \mathbb{F}[x] \colon \partial g < \partial h\}$ munido da adição e multiplicação módulo h é um corpo com p^n elementos, para $n = \partial h$, portanto (isomorfo a) o $\mathbb{F}_{p^n}$. Todos os corpos finitos são obtidos desse modo e não há uma escolha canônica para h, escolhas diferentes resultam em corpos diferentes, porém isomorfos. Não é conhecido algoritmo determinístico eficiente para achar um polinômio irredutível $h \in \mathbb{F}[x]$ de grau n.

Se $h \in \mathbb{Z}_n[x]$ é mônico então dizemos que $f, g \in \mathbb{Z}_n[x]$ são **congruentes módulo** $h(x)$, denotado $f \equiv g \pmod{h}$, se existe $q \in \mathbb{Z}_n[x]$ tal que $qh = f - g$, ou seja, $f - g$ é divisível por h. Então $\mathbb{Z}_n[x]/(h)$ denota o conjunto de todos os polinômios em $\mathbb{Z}_n[x]$ de grau menor que ∂h que, munido das operações $+_h$ e $\cdot_h$ definidas por

$$f +_h g \coloneqq (f + g) \pmod{h}$$
$$f \cdot_h g \coloneqq (f \cdot g) \pmod{h}$$

é um anel com unidade. Nos algoritmos podemos considerar os elementos de $\mathbb{Z}_n[x]/(h)$ como vetores de comprimento $d = \partial h$. Somar dois deles pode ser feito facilmente usando $O(d)$ adições em $\mathbb{Z}_n$. A multiplicação e a divisão podem ser feitas do modo usual com $O(d^2)$ multiplicações e adições no anel dos coeficientes. Finalmente, calculamos $fg \pmod{h}$ pelo algoritmo de divisão polinomial, resultando no final um tempo de execução de $O(d^2)$ e o mdc de dois polinômios de grau no máximo d pode ser computado em $O(d^2 \log d)$. De fato, é possível fazer melhor, como na observação 2.12; multiplicação em tempo $M(d) = O(d \log d \log\log d)$ usando a técnica de Schönhage–Strassen (transformada rápida de Fourier), o mdc em tempo $O(\log(d)M(d))$ e $f^k \bmod h$ em tempo $O(\log(k)M(d))$.

Um caso particularmente interessante em Computação são os **corpos binários**, corpos com 2^n elementos para $n \in \mathbb{N}$. Os elementos de $\mathbb{F}_{2^n}$ podem ser identificados com as sequências binárias $\{0,1\}^n$ da seguinte forma. Para $n = 1$, temos $\mathbb{F}_2 = \mathbb{Z}_2$, ou seja, o conjunto $\{0,1\}$ com as operações binárias usuais de soma ("ou" exclusivo lógico) e multiplicação módulo 2 ("e" lógico). Como já sabemos, $\mathbb{F}_2[x]$ denota o anel dos polinômios com coeficientes em $\mathbb{F}_2$ com soma e multiplicação usuais de polinômios e as operações nos coeficientes são módulo 2. O quociente $\mathbb{F}_2[x]/(h)$ é um corpo com 2^n elementos para qualquer $h \in \mathbb{F}_2[x]$ irredutível em $\mathbb{F}_2[x]$ de grau n. Os elementos desse corpo são dados por todos os polinômios $p \in \mathsf{F}_2[x]$ de grau no máximo n os quais são representados pelas sequências $(b_0, b_1, \ldots, b_{n-1}) \in \{0,1\}^n$ de seus coeficientes de maneira natural, $p = x^n + b_0 x^{n-1} + \cdots + b_{n-2} x + b_{n-1}$.

Por exemplo, o corpo $\mathbb{F}_{2^2}$ é dado por $\mathbb{Z}_2[x]/(x^2+x+1) = \{0, 1, x, x+1\}$ e pode ser representado por $\{(0,0), (0,1), (1,0), (1,1)\}$. A representação polinomial de $\mathbb{F}_{2^3}$ com $h(x) = x^3 + x + 1$ é dada pelos polinômios $\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$.

O corpo com dezesseis elemento é dado por $\mathbb{Z}_2[x]/(x^4+x+1) = \{0,1,x,x^2,x^3,x+1,x^2+1,x^3+1,x^2+x,x^3+x,x^3+x^2,x^3+x^2+x,x^2+x+1,x^3+x+1,x^3+x^2+1,x^3+x^2+x+1\}$ ou por $\mathbb{Z}_2[x]/(x^4+x^3+x^2+x+1)$.

A soma nesses conjuntos é feita como o usual para polinômios, no caso das sequências binárias é coordenada a coordenada. Por exemplo, em $\mathbb{F}_{2^3}$ temos $(x^2+1)+(x^2+x+1) = (1+1)x^2+x+(1+1) = x$ e o produto é o produto usual tomado módulo h, por exemplo temos $(x^2+x)(x^2+x+1) = x^4+x$ que módulo $h(x)$ é x^2.

De modo geral, um elemento de $\mathbb{F}_{p^n}$ é composto por d elementos de $\mathbb{F}_p$ e eficiência numa operação significa tempo polinomial em $d\log p$, assumindo tempo constante para as operações em $\mathbb{F}$. Assim, se h é dado, a aritmética em $\mathbb{F}_{p^n}$ é fácil. Uma caracterização de polinômios irredutíveis é dada no execício 2.37, ela é consequência do seguinte resultado sobre polinômios irredutíveis que enunciaremos sem prova (Lidl e Niederreiter, 1997, teorema 3.20).

Teorema 2.36 *Seja P_k o produto de todos os polinômios mônicos, irredutíveis no corpo $\mathbb{Z}_p[x]$ de grau k. Então o produto de todos os P_k para k que divide n é*

$$\prod_{k|n} P_k = x^{p^n} - x$$

para todo n.

Esse teorema fornece o seguinte algoritmo para testar se um polinômio é irredutível.

EXERCÍCIO 2.37. Um polinômio mônico h de grau n é irredutível se, e somente se, para cada $i = 1,2,\ldots,\lfloor n/2\rfloor$ vale que $\mathrm{mdc}(x^{p^i} - x \bmod h, h) = 1$.

No pior caso, tal algoritmo computa $n/2$ vezes uma exponenciação modular com potência p e um mdc com polinômios de grau no máximo n, resultando no tempo de execução $O(n^3\log(pn))$.

Para estimar a probabilidade de achar polinômios mônicos irredutíveis vamos usar o seguinte resultado (veja Lidl e Niederreiter, 1997, exercício 3.27) que pode ser deduzido do teorema acima usando a formula de inversão de Möbius

$$f(n) = \sum_{d|n} g(d) \Longleftrightarrow g(n) = \sum_{d|n} \mu(d) f\left(\frac{n}{d}\right)$$

em que $\mu(1) = 1$, $\mu(n) = 0$ se n é divisível por um quadrado maior que 1, senão $\mu(n) = (-1)^k$ onde k é a quantidade de primos distintos que dividem n.

Do teorema, $x^{p^n} - x = \prod_{d|n} P_d$, portanto, tomando os graus

$$p^n = \sum_{d|n} \partial P_d \Longleftrightarrow \partial P_n = \sum_{d|n} \mu(d) p^{\frac{n}{d}}$$

e se $I(n)$ é o número de polinômios mônicos irredutíveis de grau n no $\mathbb{Z}_p[x]$, então $\partial P_n = nI(n)$, logo

$$I(n) = \frac{1}{n}\sum_{d|n} \mu(d) p^{\frac{n}{d}}.$$

Também, $\partial P_n \leqslant \sum_{d|n} \partial P_d = p_n$ donde tiramos

$$I(n) \leqslant \frac{p^n}{n}.$$

Por outro lado,

$$p^n - \partial P_n = \sum_{\substack{d|n \\ d<n}} \partial P_d \leqslant \sum_{\substack{d|n \\ d<n}} p^d \leqslant \sum_{d=1}^{n/2} p^d = \frac{p^{n/2+1}-p}{p-1} \leqslant \frac{p}{p-1} p^{n/2} \leqslant 2p^{n/2}$$

portanto, $\partial P_n \geqslant p^n - 2\sqrt{p^n}$.

Lema 2.38 *Se p é primo então há pelo menos $(p^n - 2\sqrt{p^n})/n$ polinômios mônicos irredutíveis de grau n no $\mathbb{Z}_p[x]$.* □

Teste de Rabin para irredutibilidade de polinômios Como vimos, polinômios irredutíveis em $\mathbb{F}_p[x]$ são usados para transportar a aritmética do corpo $\mathbb{F}_p$ para extensões do $\mathbb{F}_p$, o corpo $\mathbb{F}_{p^n}$ é isomorfo a $\mathbb{F}_p[x]/(h)$ qualquer que seja o polinômio $h \in \mathbb{F}_p[x]$ irredutível e de grau n (Lidl e Niederreiter, 1997). Esse isomorfismo permite a construção das tabelas aritméticas a partir das operações em polinômios. O problema que precisamos resolver é encontrar tal h. A ideia de Rabin para se obter polinômios irredutíveis é a mais óbvia possível sortear o polinômio e testar a irredutibilidade.

Instância : inteiro positivo n.
Resposta : polinômio mônico irredutível em $\mathbb{Z}_p[x]$ de grau n.

1 **repita**
2 | **para** i *de* 1 *até* n **faça** $a_i \leftarrow_R \{0, 1, \ldots, n-1\}$
3 **até que** $x^n + \sum_{i=0}^{n-1} a_i x^i$ *seja irredutível*

Algoritmo 17: gerador de polinômios irredutíveis.

O tempo de execução desse algoritmo depende dos sorteios e custo do teste de irredutibilidade. Este último deixamos pra depois, por ora temos, usando o lema 2.38 acima, que

$$\frac{I(n)}{p^n} \geqslant \frac{1}{n}\left(1 - \frac{2}{\sqrt{p^n}}\right) > \frac{1}{2n}$$

para $p^n > 16$. Da equação (2.4) com a probabilidade acima, que o número médio de rodadas do **repita** é menor que $2n$, portanto o custo médio do laço é $O(nT(n))$ em que T é o tempo para testar irredutibilidade de um polinômio de grau n.

O teste de irredutibilidade é baseado no seguinte resultado.

Teorema 2.39 (Rabin, 1980b) *Sejam n um natural e p um primo. Um polinômio $h \in \mathbb{Z}_p[x]$ de grau n é irredutível em $\mathbb{Z}_p$ se, e só se,*

$$h(x) \,\Big|\, \left(x^{p^n} - x\right) \tag{2.12}$$

$$\operatorname{mdc}\left(h(x), x^{p^{n/q}} - x\right) = 1 \text{ para todo fator primo } q \text{ de } n. \tag{2.13}$$

Demonstração. Se h é irredutível então (2.12) e (2.13) seguem do teorema 2.36. Para a recíproca, assumamos (2.12). Então, novamente pelo teorema 2.36, temos que se f é um fator irredutível de h com grau d então d divide n. Suponhamos que $d < n$, então $d|(n/q)$ para algum fator primo q de n. Mas se esse é o caso, então f divide $x^{p^{n/q}} - x$, contradizendo (2.13), portanto $d = n$. □

Instância : $h \in \mathbb{Z}_p[x]$ mônico de grau n e os fatores primos $q_1, q_2, \ldots, q_k$ de n.
Resposta : *sim*, h é irredutível ou *não*, h é redutível.

1 **para** i *de* 1 *até* k **faça**
2 | **se** $\operatorname{mdc}(h, x^{p^{n/q_i}} - x \pmod h) \neq 1$ **então responda** *não*.
3 **se** $x^{p^n} - x \pmod h = 0$ **então responda** *sim*.
4 **senão responda** *não*.

Algoritmo 18: teste de irredutibilidade de Rabin.

O $\operatorname{mdc}(h, x^{p^{n/q_i}} - x \pmod h)$ é computado calculando-se a potência $x^{p^n} \pmod h$, o que pode ser feito com $O(n \log p)$ operações de custo $O(n^2)$ realizadas $k = O(\log n)$ vezes, portanto, $O(n^3 \log(n) \log(p))$ operações do corpo. De fato, é possível melhorar essa estimativa, o tempo de execução desse algoritmo é $O(nM(n) \log\log(n) \log(p))$ (Gao e Panario, 1997). Ademais, Gao e Panario (1997) propuseram uma melhoria no algoritmo de Rabin que resulta em tempo de execução $O(nM(n) \log p)$.

2.3 TESTES DE PRIMALIDADE ALEATORIZADOS

Recordemos que um natural $n > 1$ é **primo** se os únicos divisores positivos de n são 1 e n. Um natural $n > 1$ é **composto** se admite divisores positivos além de 1 e n.

O problema de decidir se um dado inteiro $n \geqslant 2$ é primo é muito antigo, o crivo de Eratóstenes (200 aC) é um dos primeiros algoritmos para teste de primalidade.

Problema computacional da primalidade:
Instância : dado $n \geqslant 2$ inteiro ímpar.
Resposta : *sim* se n é primo, *não* caso contrário.

Uma solução para esse problema é eficiente se a decisão é tomada tempo polinomial em $\log_2 n$. Do ponto de vista computacional o crivo de Eratóstenes não é eficiente, tem complexidade exponencial em $\log_2 n$.

A busca por soluções eficientes parece ter sido proposta pela primeira vez por Gödel em 1956. Decidir se um número é primo pode ser resolvido em tempo polinomial. Os indianos Agrawal, Kayal e Saxena (2004) mostraram um algoritmo que determina se n é primo em tempo $O(\log^{12+\varepsilon} n)$. Esse algoritmo já foi bastante aperfeiçoado, o expoente no logaritmo diminuiu para $6+\varepsilon$ mas ainda está longe de ser mais viável que os algoritmos probabilísticos.

Os algoritmos probabilísticos são extremamente simples e eficientes, como é o caso do teste de Miller–Rabin, de complexidade $O(k \log^2(n) \log\log(n) \log\log\log(n))$, em que k é o número de rodadas do teste (Monier, 1980). A probabilidade de erro é exponencialmente pequena em k. Para $k = 100$ a probabilidade do algoritmo responder *primo* todas as vezes quando a entrada é um número composto é menor que 10^{-30}, que é um número muito pequeno (aproximadamente a massa do elétron). Esse teste é bastante usado e no exemplo a seguir damos alguns casos.

Exemplo 2.40. O seguinte texto do manual do Maxima, um sistema de computação algébrica sob a licença pública GNU é um exemplo típico de como a primalidade de um número é testada na prática.

> "`primep (n)` Teste de primalidade. Se primep(n) retorna false, n é um número composto, e se ele retorna true, n é um número primo com grande probabilidade.
>
> Para n menor que 10^16 uma versão deterministica do teste de Miller-Rabin é usada. Se primep(n) retorna true, então n é um número primo.
>
> Para n maior do que 10^16 primep realiza primep_number_of_tests testes de pseudo-primalidade de Miller-Rabin e um teste de pseudo-primalidade de Lucas. A probabilidade com que n passe por um teste de Miller-Rabin é inferior a 1/4. Usando o valor padrão 25 para primep_number_of_tests, a probabilidade de n ser composto é muito menor do que 10^-15."

O *Mathematica* também implementa seu teste de primalidade usando algoritmos probabilísticos.

> "`PrimeQ` primeiro testa divisibilidade por primos pequenos, em seguida usa o teste de Miller-Rabin para bases 2 e 3, e em seguida usa o teste de Lucas."

No Python, a biblioteca SymPy usada para computação simbólica implementa um teste de primalidade; a documentação descreve

> "`sympy.ntheory.primetest.isprime(n)` Test if n is a prime number (True) or not (False). For n < 10^16 the answer is accurate; greater n values have a small probability of actually being pseudoprimes.
>
> Negative primes (e.g. -2) are not considered prime.
>
> The function first looks for trivial factors, and if none is found, performs a safe Miller-Rabin strong pseudoprime test with bases that are known to prove a number prime. Finally, a general Miller-Rabin test is done

with the first k bases which, which will report a pseudoprime as a prime with an error of about 4^-k. The current value of k is 46 so the error is about 2 x 10^-28."

Notemos que em todos os casos as probabilidades de erro são muito pequenas. ◊

2.3.1 OS TESTES DE FERMAT E LUCAS

Um resultado célebre da teoria dos números, o Pequeno Teorema de Fermat enunciado a seguir, garante que para inteiros n e a, *se* $a^{n-1} \not\equiv 1 \pmod{n}$ *então* n *é composto.* O seguinte teorema decorre do Teorema de Euler pois, de acordo com o exercício 2.34, temos $\varphi(p) = p-1$ para todo p primo.

Teorema 2.41 (Pequeno Teorema de Fermat) *Se* $a \in \mathbb{Z} \setminus \{0\}$ *e* p *é primo que não divide* a*, então* $a^{p-1} \equiv 1 \pmod{p}$. □

Esse teorema nos dá o teste de primalidade

$$p \text{ é primo se, e somente se, } a^{p-1} \equiv 1 \pmod{p} \text{ para todo } 1 \leqslant a \leqslant p-1.$$

Dado um n, para qual queremos testar primalidade, se existe um inteiro $a \in \{1, 2, \ldots, n-1\}$ tal que $a^{n-1} \not\equiv 1 \pmod{n}$, então esse teorema nos garante que n é composto. Chamamos tal a de uma **testemunha de Fermat** para o fato de n ser composto.

A ideia do teste de Fermat é, para dado n, sortear a e testar se a é testemunha para n, se for testemunha então n é composto, senão possivelmente um primo.

Instância : um inteiro ímpar $n \geqslant 4$.
Resposta : *sim* se n é primo, *não* caso contrário.

1 $a \leftarrow_R \{2, 3, \ldots, n-2\}$;
2 **se** $a^{n-1} \not\equiv 1 \pmod{n}$ **então responda** não.
3 **senão responda** sim.

Algoritmo 20: teste de Fermat.

Usando o algoritmo de exponenciação modular (algoritmo 7, página 47), o tempo de execução do teste de Fermat é $O(\log^3 n)$, logo de tempo polinomial em $\log_2 n$. O algoritmo só erra ao declarar que n é primo, a resposta *não* está sempre correta. Vamos estimar a probabilidade de erro.

Uma testemunha fixa não serve para todo número composto. Por exemplo, é possível verificar que 2 é testemunha para todo número natural composto até 340, mas que não é testemunha para $341 = 11 \cdot 31$ pois $2^{340} \equiv 1 \pmod{341}$.

Dizemos que um natural n é um **pseudoprimo de Fermat para a base** a quando vale que

$$n \text{ é ímpar, composto e } a^{n-1} \equiv 1 \pmod{n}.$$

A base $a \in \{1, 2, \ldots, n-1\}$ para o qual n é pseudoprimo é uma testemunha **mentirosa** para n.

Podemos descobrir que 341 é composto testando-o contra outras bases e nesse caso $3^{340} \equiv 54 \pmod{341}$ o que atesta que 341 é composto. É possível estender essa estratégia e testar se n contra toda base $a \in \{2, 3, \ldots, n-2\}$, porém isso não resulta em um algoritmo viável, o tempo de execução seria exponencial no tamanho da representação de n. A proposição abaixo garante que se incrementamos a base a e fazemos o teste "$a^{n-1} \equiv 1 \pmod{n}$?", então o mais longe que iremos é até o menor divisor primo de n, mas isso pode não ser muito melhor do que usar crivo de Eratóstenes.

Proposição 2.42 *Todo* n *ímpar e composto é pseudoprimo para as bases* 1 *e* $n-1$*. Ademais, não há* $n \geqslant 4$ *pseudoprimo para toda base* $a \in \{1, 2, \ldots, n-1\}$.

Demonstração. A primeira afirmação, que todo n ímpar e composto é pseudoprimo para as bases 1 e $n-1$ segue trivialmente de que valem $1^{n-1} \equiv 1 \pmod{n}$ e

$$(n-1)^{n-1} = \sum_{i=0}^{n-1} \binom{n-1}{i} n^i (-1)^{n-i-1} \equiv (-1)^{n-1} \pmod{n}$$

logo $(n-1)^{n-1} \equiv 1 \pmod{n}$ pois n é ímpar.

Se n é impar, composto e $a^{n-1} \equiv 1 \pmod{n}$, então $a^{(n-1)/2} a^{(n-1)/2} \equiv 1 \pmod{n}$, ou seja, $a^{(n-1)/2}$ tem inverso multiplicativo módulo n, portanto $\operatorname{mdc}(a^{(n-1)/2}, n) = 1$ donde deduzimos que $\operatorname{mdc}(a, n) = 1$. Então, para n ser pseudoprimo para todo $a \in \{2, \ldots, n-2\}$ ele deve ser primo, uma contradição. □

Como argumentamos na demonstração acima, se $a^k \equiv 1 \pmod{n}$ para algum $k > 1$, então $aa^{k-1} \equiv 1 \pmod{n}$, ou seja, a tem inverso multiplicativo módulo n, portanto, $a \in \mathbb{Z}_n^*$. Nesse caso, o conjunto das bases mentirosas para $n \geqslant 3$ ímpar e composto

$$\mathsf{M}_n := \left\{ a \in \{1, 2, \ldots, n-1\} \colon a^{n-1} \equiv 1 \pmod{n} \right\}$$

é um subconjunto do $\mathbb{Z}_n^*$. Ademais, pela proposição acima $1 \in \mathsf{M}_n$. Se $a, b \in \mathsf{M}_n$, então $a \cdot b \pmod{n} \in \mathsf{M}_n$ pois $(ab)^{n-1} \equiv a^{n-1} b^{n-1} \equiv 1 \pmod{b}$, portanto M_n é subgrupo de $\mathbb{Z}_n^*$. Pelo Teorema de Lagrange[4] temos, para algum inteiro m, que $m|\mathsf{M}_n| = |\mathbb{Z}_n^*|$.

Se M_n for subgrupo próprio temos $m \geqslant 2$, portanto uma escolha na linha 1 do algoritmo causa erro na resposta com probabilidade

$$\frac{|\mathsf{M}_n \setminus \{1, n-1\}|}{|\{2, \ldots, n-2\}|} \leqslant \frac{|\mathsf{M}_n|}{n-1} \leqslant \frac{|\mathbb{Z}_n^*|/2}{n-1} < \frac{1}{2}$$

Senão, $m = 1$ e a igualdade $\mathsf{M}_n = \mathbb{Z}_n^*$ ocorre quando n é um **número de Carmichael**

$$n \text{ é impar, composto e } a^{n-1} \equiv 1 \pmod{n} \text{ para todo } a \in \mathbb{Z}_n^*.$$

Esses números são bastante raros. Sobre a distribuição, sabemos que se $c(n)$ é a quantidade de números de Carmichael até n, então

$$n^{0{,}332} < c(n) < n e^{-\frac{\log n \log\log\log n}{\log\log n}}.$$

A cota inferior é de Harman (2005) e dela deduz-se que há infinitos números de Carmichael, a cota superior é de Erdős (1956). Note que desse fato podemos concluir que a densidade dos números de Carmichael até 10^{256} é menor que $0{,}8 \times 10^{-58}$.

Nesse caso, quando um número de Carmichael n é dado como entrada no teste de Fermat, o algoritmo responde certo somente se escolher $a \in \{2, 3, \ldots, n\}$ tal que $\operatorname{mdc}(a, n) > 1$. A probabilidade de erro é

$$\frac{\varphi(n) - 2}{n-3} > \frac{\varphi(n)}{n} = \prod_p \left(1 - \frac{1}{p}\right)$$

onde o produto é sobre todo primo p que divide n (corolário 2.33). Se n tem poucos e grandes fatores primo, resulta num número próximo de 1.

EXERCÍCIO 2.43 (critério de Korselt). Prove que n é um número de Carmichael se, e só se, é composto, livre de quadrado (não é divisível por um quadrado maior que 1) e $p-1$ divide $n-1$ para todo p que divide n.

EXERCÍCIO 2.44. Prove que n tem pelo menos 3 fatores primos distintos. (Dica: assuma $n = pq$ e derive uma contradição usando o exercício anterior.)

[4] Em grupos finitos, a cardinalidade de um subgrupo divide a cardinalidade do grupo.

Segundo Garfinkel (1994) o teste Fermat é usado pelo PGP[5], versão 2.6.1, para gerar números primos. A chance do PGP gerar um número de Carmichael é menor que 1 em 10^{50}. O PGP escolhe um primo para seus protocolos de criptografia e assinatura digital da seguinte maneira: escolhe $b \in_{\mathcal{U}} \{0,1\}^{t-2}$ e acrescenta 11 como os 2 bits mais significativos, com isso temos um natural m de t bits; verifica se m é divisível por algum dos 100 menores primos; se m não é primo então recomeça-se com o próximo ímpar; caso contrário, usa 4 rodadas do teste de Fermat.

O teste de Lucas Acabamos de ver um método probabilístico que ao declarar que um número é primo, tal número ou de fato é primo ou tivemos muito azar em tentar provar que o número é composto. Uma vez que não esperamos uma sequência de eventos ruins, depois de algumas repetições aceitamos que o número é primo. No entanto, não temos uma prova de primalidade do número. Esta seção é dedicada a um teste que pode provar que um número é primo.

O teste de Lucas é baseado no Pequeno Teorema de Fermat e na fatoração de $n-1$. Existem alguns testes para primalidade de n baseados na fatoração de $n+1$ ou de $n-1$ e que funcionam bem quando essas fatorações são fáceis como é o caso dos números de Fermat e de Mersenne.

Teorema 2.45 *Um inteiro $n>2$ é primo se, e somente se, existe $a \in \{1,2,\ldots,n-1\}$ tal que*

(1) $a^{n-1} \equiv 1 \pmod{n}$ *e*

(2) $a^{(n-1)/p} \not\equiv 1 \pmod{n}$ *para todo p primo e divisor de $n-1$.*

Antes de provar esse teorema, verifiquemos o seguinte fato que será usado: se $a^{n-1} \equiv 1 \pmod{n}$ então $\mathrm{mdc}(a,n)=1$. Assuma que $n \mid a^{n-1}-1$ e que $d \mid n$ e $d \mid a$. De $d \mid n$ e $n \mid a^{n-1}-1$ temos $d \mid a^{n-1}-1$, mas $d \mid a^{n-1}$ portanto $d \mid 1$, logo $d = \pm 1$. Tomando $d = \mathrm{mdc}(a,n)$, temos $d=1$.

Demonstração. Se valem (1) e (2), então pelo corolário 2.24 a tem ordem multiplicativa $n-1$. De (1) temos que $\mathrm{mdc}(a,n)=1$, isso e o Teorema de Euler implicam que $n-1$ divide $\varphi(n)$ (exercício 2.22) e como $\varphi(n) = |\mathbb{Z}_n^*| \leqslant n-1$ temos que $\varphi(n) = n-1$, ou seja, n é primo (exercício 2.34).

Se n é primo, tome para a uma raiz primitiva. O item (1) decorre do teorema de Fermat. O item (2) decorre de a ser raiz primitiva, logo $\mathrm{ord}_*(a) = n-1$, portanto, $a^k \not\equiv 1 \pmod{n}$ para $k < n-1$ positivo. □

Instância : inteiros positivos $n>1$, k e os fatores primos distintos de $n-1$.
Resposta : *sim* n é primo, *não* se n é composto.

1 **repita**
2 $a \leftarrow_R \{2,3,\ldots,n-1\}$;
3 **se** $a^{n-1} \equiv 1 \pmod{n}$ **então**
4 **para cada** *fator primo p de $n-1$* **faça**
5 **se** $a^{(n-1)/p} \not\equiv 1 \pmod{n}$ **então responda** *sim.*
6 **até que** *completar k iterações;*
7 **responda** *não.*

Algoritmo 21: teste de Lucas.

A quantidade de divisores primos distintos de $n-1$ é $k = O(\log n)$ (veja equação (2.11)). Cada iteração no algoritmo calcula $k+1$ exponenciais e uma exponenciação modular custa $O(\log^3 n)$, portanto, o tempo de execução é $kO(\log^4 n) = O(\log^5 n)$.

[5] *Pretty Good Privacy*, é um software de criptografia que fornece autenticação e privacidade criptográfica para comunicação de dados desenvolvido por Phil Zimmermann em 1991.

2.3.2 O TESTE DE MILLER–RABIN

O teste de Miller–Rabin surgiu de duas contribuições. Primeiro, Miller (1975) propôs um teste de primalidade determinístico baseado na validade da hipótese de Riemann generalizada (ainda uma conjectura) e Rabin (1980a) introduziu aleatoriedade no teste tornando-o independente dessa hipótese. O teste de Miller–Rabin é baseado no fato de que se $n > 2$ é primo, então a equação $x^2 \equiv 1 \pmod{n}$ tem exatamente duas soluções inteiras, como demonstramos a seguir. Observemos que para todo $n \geqslant 1$ vale que $-1 \equiv n-1 \pmod{n}$ e que no caso $n = 2$ temos $1 \equiv -1 \pmod{2}$.

Proposição 2.46 *Se $p > 2$ é primo então $x^2 \equiv 1 \pmod{p}$ tem exatamente duas soluções inteiras, a saber $x = 1$ e $x = -1$.*

Demonstração. De $p > 2$ temos $1 \not\equiv -1 \pmod{p}$. Claramente, ± 1 satisfazem a equação $x^2 \equiv 1 \pmod{p}$. Agora, se $x^2 \equiv 1 \pmod{p}$ então $p \mid (x+1)(x-1)$, portanto, $p \mid x-1$ ou $p \mid x+1$. Mas, se $p \mid x-1$ então $x \equiv 1 \pmod{p}$ e se $p \mid x+1$ então $x \equiv -1 \pmod{p}$. Portanto, essa são as únicas soluções. □

Assim, se para n ímpar e um inteiro a temos $a^{n-1} \not\equiv 1 \pmod{n}$, então n é composto. Senão $a^{n-1} \equiv 1 \pmod{n}$, então $a^{(n-1)/2} \equiv 1 \pmod{n}$ ou $a^{(n-1)/2} \equiv -1 \pmod{n}$ ou outro inteiro; nos últimos casos n é composto e no primeiro caso temos $a^{(n-1)/4} \equiv 1 \pmod{n}$ ou $a^{(n-1)/4} \equiv -1 \pmod{n}$ ou outro inteiro, e assim por diante até que teremos ou $a^{(n-1)/2^j} \equiv -1 \pmod{n}$ para algum j ou $a^r \equiv \pm 1 \pmod{n}$ para r ímpar. Colocando de outra maneira, se n é ímpar, então podemos escrever $n-1 = 2^s r$ com r ímpar e a sequência de inteiros módulo n

$$\left(a^{2^0 r}, a^{2^1 r}, a^{2^2 r}, \cdots, a^{2^s r}\right)$$

para n primo é da forma

$$(\pm 1, 1, \ldots, 1, 1, 1) \text{ ou } (—, —, \ldots, —, -1, 1, \ldots, 1)$$

onde "—" significa $\notin \{1, -1\}$. Se for da forma

$$(—, —, \ldots, —) \text{ ou } (—, —, \ldots, -1) \text{ ou } (—, —, 1, 1, \ldots, 1) \text{ ou } (—, —, \ldots, —, 1)$$

então n é composto, isto é, se em algum momento temos uma sequência onde a primeira ocorrência de 1, se houver, não é precedida por -1 então n é composto.

Exemplo 2.47. Tomemos $n = 561$, o menor número de Carmichael. Temos $560 = 2^4 \cdot 35$. Para $a = 2$ temos $2^{35} \equiv 263 \pmod{561}$, $2^{2\cdot 35} \equiv 166 \pmod{561}$, $2^{4\cdot 35} \equiv 67 \pmod{561}$, finalmente $2^{8\cdot 35} \equiv 1 \pmod{561}$, portanto, n é composto. ◊

Uma vantagem da teste de Miller–Rabin com relação ao teste de Fermat é que agora não existe o efeito análogo ao dos números de Carmichael no teste de Fermat.

Lema 2.48 *Sejam n um primo ímpar e r, s inteiros tais que $n-1 = 2^s r$ com r ímpar. Então, para todo inteiro a não divisível por n vale ou $a^r \equiv 1 \pmod{n}$ ou $a^{2^j r} \equiv -1 \pmod{n}$ para algum j, $0 \leqslant j \leqslant s-1$.*

Demonstração. Seja a como no enunciado, considere a sequência módulo n

$$a^{2^0 r}, a^{2^1 r}, a^{2^2 r}, \cdots, a^{2^s r}$$

e seja t o menor expoente tal que $a^{2^t r} \equiv 1 \pmod{n}$. Como $a^{2^s r} = a^{n-1} \equiv 1 \pmod{n}$, o número t está bem definido. Então, ou $t = 0$ e temos $a^r \equiv 1 \pmod{n}$ ou $1 \leqslant t \leqslant s$ e $a^{2^t r} - 1 \equiv 0 \pmod{n}$, mas

$$a^{2^t r} - 1 = (a^{2^{t-1} r} + 1)(a^{2^{t-1} r} - 1).$$

Como n divide $a^{2^t r} - 1$ e, pela minimalidade de t, não divide $a^{2^{t-1} r} - 1$, necessariamente n divide $a^{2^{t-1} r} + 1$, ou seja, $a^{2^{t-1} r} \equiv -1 \pmod{n}$. □

Pela contrapositiva, caso não valham as conclusões do lema, ou seja, valem

$$a^r \not\equiv \pm 1 \pmod{n} \quad \text{e} \quad a^{2^j r} \not\equiv -1 \pmod{n} \quad (\forall j \in \{1,\ldots,s-1\})$$

então $n|a$ ou n é composto. Se tomamos $a \in \{1,\ldots,n-1\}$ e vale a equação acima então n é composto e a é uma **testemunha forte** para o fato de n ser composto.

Instância : inteiro $n \geqslant 3$ ímpar.
Resposta : *verdadeiro* se encontrou testemunha do fato "n é *composto*", *falso* caso contrário.

```
1 Determine s e r tal que n − 1 = 2^s r com r ímpar;
2 a ←_R {2, 3, ..., n − 2};
3 x ← a^r mod n;
4 se x ∈ {1, n − 1} então responda falso.
5 para i de 1 até s − 1 faça
6 |   x ← x^2 mod n;
7 |   se x = n − 1 então responda falso.
8 responda verdadeiro.
```

Algoritmo 22: teste de Miller–Rabin.

Cada divisão por 2 na linha 1 custa $O(\log n)$, no total são s divisões. No restante, o custo é o de uma sorteio, $O(\log n)$, mais uma exponenciação modular, $O(\log r \log^2 n)$, mais $s = O(\log n)$ multiplicações módulo n, cada uma custa $O(\log^2 n)$, portanto $O((s+1)\log n + (\log r + s)\log^2 n)$, como $\log_2 r + s = \log_2(n-1)$, o tempo de execução é $O(\log^3 n)$.

Se Teste Miller–Rabin(n, r, s) responde *falso* é porque para os parâmetros n, r, s e o sorteio a valem

$$a^r \equiv \pm 1 \pmod{n} \text{ ou} \tag{2.14}$$

$$a^{2^j r} \equiv -1 \pmod{n} \text{ para algum } j,\ 1 \leqslant j \leqslant s-1. \tag{2.15}$$

e o inteiro a *não é testemunha forte* para a composição de n, nesse caso n pode ser primo ou composto. Se n é ímpar e composto então a é uma **base mentirosa forte** para n.

Para todo composto $n = 2^s r + 1 \geqslant 3$ com $r > 1$ ímpar definimos

$$M_n := \{a \in \{1,2,\ldots,n-1\}: \text{ vale (2.14) ou (2.15)}\}$$

o conjunto das as bases mentirosas, aquelas que enganam o teste de primalidade de n fazendo-o responder primo. Em qualquer um dos casos (2.14) ou (2.15) temos $a^{n-1} \equiv 1 \pmod{n}$, portanto, como já provamos, $\text{mdc}(a, n) = 1$, logo $M_n \subset \mathbb{Z}_n^*$.

O algoritmo de Miller citado no começo dessa seção, que assume a hipótese generalizada de Riemann, é baseado num teorema que garante sob tal hipótese que se n é composto então existe uma testemunha $a = O(\log^2 n)$, ou seja, existe uma testemunha de tamanho suficientemente pequeno para que possamos descobri-la em tempo polinomial.

O teste de Miller–Rabin pode ser iterado para diminuirmos a probabilidade de erro. Veremos que a probabilidade de erro é $\leqslant 1/4$. Agora, também consideramos um parâmetro de entrada k que controla a probabilidade de erro do algoritmo, quanto maior k, menor a chance de responder errado. O tempo de execução é $O(k \log^3 n)$.

Dizemos que um inteiro ímpar e composto n é um **pseudoprimo forte** para a base a, $1 \leqslant a < n$, se vale a conclusão do lema 2.48. Os pseudoprimos fortes são pseudoprimos de Fermat, mas são mais raros que os de Fermat.

Instância : um inteiro ímpar $n \geqslant 3$ e o número de rodadas k.
Resposta : *não* se n não é primo, caso contrário *sim* com probabilidade de erro $\leqslant (1/4)^k$.

```
1 repita
2 |   se teste de Miller–Rabin(n) então responda não.
3 até que complete k rodadas;
4 responda sim.
```

Algoritmo 23: teste de Miller–Rabin com erro controlado.

Exemplo 2.49. Para $n = 91$, temos $n - 1 = 90 = 2^1 \cdot 45$. Como $9^r = 9^{45} \equiv 1 \pmod{91}$ temos que 91 é pseudoprimo forte para a base 9. Ainda, 1, 9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82, 90 são todas as bases para as quais 91 é um pseudoprimo forte. ◊

A seguir limitaremos em duas abordagens a probabilidade de erro do algoritmo. A primeira é mais simples e o resultado é mais fraco. Na segunda há maior exigência de pré-requisitos em Teoria dos Grupos. Ambas mostram que o número de testemunhas no caso composto é abundante, o que propicia um ambiente favorável para um teste aleatorizado.

Teorema 2.50 *O algoritmo de Miller–Rabin responde errado com probabilidade no máximo* $(1/2)^k$.

Demonstração. Vamos assumir que n seja um número de Carmichael: $n = 2^s r + 1$, com r ímpar, um inteiro ímpar e composto. Definimos

$$t := \max\{j \in \{0, 1, \ldots, s-1\} : \exists b \in \mathsf{M}_n,\ b^{2^j r} \equiv -1 \pmod{n}\}.$$

Notemos que $(-1)^{2^0 r} \equiv -1 \pmod{n}$ e, como n é Carmichael, $b^{n-1} = b^{2^s r} \equiv 1 \pmod{n}$, portanto $t \in \{0, 1, \ldots, s-1\}$ está definido.

Definimos $m := 2^t r$ e

$$\mathsf{K} := \{a \in \mathbb{Z}_n : a^m \equiv \pm 1 \pmod{n}\}.$$

Se $a \in \mathsf{M}_n$ então $a \in \mathsf{K}$: se $a^r \equiv 1 \pmod{n}$ então $a^{2^t r} \equiv 1 \pmod{n}$; se $a^{2^j r} \equiv -1 \pmod{n}$ para algum j, então $j \leqslant t$. Ademais, se $a \in \mathsf{K}$, então de $t < s$ deduzimos que $a^{n-1} \equiv 1 \pmod{n}$, portanto, $\mathrm{mdc}(a, n) = 1$. Dessa forma

$$\mathsf{M}_n \subset \mathsf{K} \subset \mathbb{Z}_n^*.$$

Resta mostrarmos que K é um subgrupo próprio.

Certamente, $1 \in \mathsf{K}$. Agora, tomemos $a, b \in \mathsf{K}$. Então $(ab)^m \equiv a^m b^m \equiv (\pm 1)(\pm 1) \equiv \pm 1 \pmod{n}$.

Para finalizar, vamos mostrar um elemento de $\mathbb{Z}_n^*$ que não está em K. Para isso, vamos usar o exercício 2.44 que garante que n tem pelo menos três divisores primos distintos, logo podemos escrever $n = n_1 n_2$ com n_1 e n_2 ímpares e coprimos.

Definimos $a_1 := b \pmod{n_1}$ e considere a única solução $x \in \mathbb{Z}_n$ de

$$\begin{cases} x \equiv a_1 & \pmod{n_1} \\ x \equiv 1 & \pmod{n_2} \end{cases} \tag{2.16}$$

dada pelo Teorema Chinês do Resto (A.2, página 190).

Módulo n_1 temos que $x^m \equiv a_1^m \equiv b^m \equiv -1 \pmod{n_1}$. Módulo n_2 temos que $x^m \equiv 1^m \equiv 1 \pmod{n_2}$. Portanto, $x^m \not\equiv \pm 1 \pmod{n}$, ou seja, $x \notin \mathsf{K}$. Por outro lado, $x^{m^2} \equiv 1 \pmod{n_1}$ e $x^{m^2} \equiv 1 \pmod{n_2}$, portanto $x^{m^2} \equiv 1 \pmod{n}$ pelo teorema chinês do resto, logo $\mathrm{mdc}(x, n) = 1$, ou seja, $x \in \mathbb{Z}_n^*$.

Agora, se n não é número de Carmichael, então

$$\mathsf{M}_n \subset \mathsf{F} \subsetneq \mathbb{Z}_n^*$$

com os mentirosos de Fermat F subgrupo próprio de $\mathbb{Z}_n^*$.

Em ambos os casos, existe um X subgrupo próprio de $\mathbb{Z}_n^*$ com $M_n \subset X \subset \mathbb{Z}_n^*$ e, pelo Teorema de Lagrange, $|X| = \varphi(n)/m \leqslant \varphi(n)/2$ pois $m \geqslant 2$, e $|M_n| \leqslant |X|$. Assim, a probabilidade de erro em uma rodada é

$$\frac{|M_n \setminus \{1, n-1\}|}{\{2,3,\ldots,n-2\}} \leqslant \frac{|M_n|}{\{1,2,\ldots,n-1\}} \leqslant \frac{|X|}{\varphi(n)} \leqslant \frac{1}{2}$$

e em k rodadas independentes $(1/2)^k$. □

Com mais esforço conseguimos um resultado melhor. A seguir vamos dar uma demonstração que requer do leitor conhecimento de alguns resultados básicos da Teoria dos Grupos.

Vamos definir uma sequência ordenada por inclusão de subgrupos de $\mathbb{Z}_n^*$ que contêm as não testemunhas M_n (que não é subgrupo)

$$M_n \subset K \subset L \subset F \subset \mathbb{Z}_n^*$$

e das três últimas inclusões, se duas forem próprias temos $|M_n| \leqslant 4\varphi(n)$, logo a probabilidade de sortear um mentiroso é $\leqslant 1/4$. Nas inclusões acima F são os mentirosos de Fermat

$$F = \{a \in \mathbb{Z}_n : a^{n-1} \equiv 1 \pmod{n}\}.$$

Como já vimos acima, se n não for um número de Carmichael, então pelo menos um $a \in \mathbb{Z}_n^*$ não é um mentiroso de Fermat logo a última inclusão é estrita.

Suponha que n é um número de Carmichael fatorado como $n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$ com $k \geqslant 3$ e $p_i > 2$ para todo i.

EXERCÍCIO 2.51. Assuma que $n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$ e prove que para quaisquer inteiros x e y, se $x \equiv y \pmod{n}$ então $x \equiv y \pmod{p_i^{m_i}}$ para todo i.

Escrevemos $n = 2^s r + 1$, com r ímpar, e definimos

$$t := \max\{j \in \{0,1,\ldots,s-1\} : \exists b \in \mathbb{Z}_n^*,\ b^{2^j} \equiv -1 \pmod{n}\}.$$

Definimos $m := 2^t r$ e os grupos $K \subset L$ dados por

$$L := \{a \in \mathbb{Z}_n : a^m \equiv \pm 1 \pmod{p_i^{m_i}} \text{ para todo } i\},$$
$$K := \{a \in \mathbb{Z}_n^* : a^m \equiv \pm 1 \pmod{n}\}$$

e, finalmente, definimos o homomorfismo de grupos

$$f : L \to \{\pm 1 \bmod p_1^{m_1}\} \times \{\pm 1 \bmod p_2^{m_2}\} \times \cdots \times \{\pm 1 \bmod p_k^{m_k}\}$$
$$a \bmod n \mapsto \left(a^m \bmod p_1^{m_1}, a^m \bmod p_k^{m_k}, \ldots, a^m \bmod p_k^{m_k}\right).$$

O *kernel* do homomorfismo é o subgrupo de L dado pelos elementos de L cuja imagem por f é $(1,1,\ldots,1)$. Pelo teorema do isomorfismo de grupos temos $L/\ker f$ é isomorfo a $\mathrm{Im}(f) = \prod_i \{\pm 1 \bmod p_i^{m_i}\}$ pois f é sobrejetora. Ademais $f(K) = \{(-1,-1,\ldots,-1),(1,1,\ldots,1)\}$.

Do parágrafo acima concluímos que $f(L)$ tem ordem 2^k e $F(K)$ tem ordem 2, portanto $|L|/|\ker f| = 2^k$ e $|K|/|\ker f| = 2$, então $|L|/|K| = 2^{k-1} \geqslant 4$ pois $k \geqslant 3$.

Se para o número de fatores primos distintos de n vale $k = 2$ então n não é um número de Carmichael, logo $F \neq \mathbb{Z}_n^*$. Vamos mostrar que $F \neq L$. Escrevemos $n = n_1 n_2$ com n_1 e n_2 ímpares e coprimos e tomamos x como em (2.16) de modo que $x \notin K$, porém $x \in F$ pois $x^{m^2} \equiv 1 \pmod{n}$ e $m^2 \mid n-1$, portanto $x^{n-1} \equiv 1 \pmod{n}$. Com isso,

$$K \subsetneq F \subsetneq \mathbb{Z}_n^*$$

de modo que $|K| \leqslant \varphi(n)/4$.

Finalmente, se $k = 1$ então $n = p^e$ com $e \geqslant 2$ dado que n não é primo. Nesse caso, sabemos que $\mathbb{Z}_n^*$ é cíclico (tem raiz primitiva), portanto, temos um isomorfismo entre os grupos $(\mathbb{Z}_n^*, \cdot \bmod n)$ e $(\mathbb{Z}_{\varphi(n)}, + \bmod n)$. O número de soluções módulo n de $x^{n-1} \equiv 1 \pmod n$ é igual ao número de soluções de $(n-1)x \equiv 0 \pmod{\varphi(n)}$, portanto $|F| = \text{mdc}(n-1, \varphi(n)) = \text{mdc}(p^e - 1, (p-1)p^{e-1}) = p - 1$. Portanto vale

$$\frac{|\mathbb{Z}_n^*|}{|F|} = p^{e-1} \geqslant 4$$

todo $n > 9$. Quando $n = 9$, verifica-se que $M_n = \{1, n-1\} = (n-1)/4$. De fato, $n - 1 = 8 = 2^3$, logo $e = 3$ e $k = 1$. A sequência de teste do algoritmo de Miller–Rabin para a é $(a \bmod 9, a^2 \bmod 9, a^4 \bmod 9)$

$a \bmod 9$	1	2	3	4	5	6	7	8
$a^2 \bmod 9$	1	4	0	7	7	0	4	1
$a^4 \bmod 9$	1	7	0	4	4	0	7	1

portanto 1 e 8 são mentirosos para 9. Em todos os casos, para todo $n \geqslant 9$ vale que $|M_n| \leqslant \varphi(n)/4$. Assim, a probabilidade de erro em uma rodada é

$$\frac{|M_n \setminus \{1, n-1\}|}{\{2, 3, \ldots, n-2\}} \leqslant \frac{|M_n|}{\{1, 2, \ldots, n-1\}} \leqslant \frac{\varphi(n)/4}{\varphi(n)} \leqslant \frac{1}{4}$$

e em k rodadas independentes $(1/2)^k$.

Teorema 2.52 *Para todo $n \geqslant 9$, o algoritmo de Miller–Rabin responde errado com probabilidade no máximo $(1/4)^k$.* □

Para muitos inteiros compostos n a quantidade de bases para as quais n é pseudoprimo forte é bem pequeno, muito menor que a estimativa $\varphi(n)/4$. Por exemplo, para $n = 105$ temos $M_{105} = \{1, 104\}$. Entretanto, existe inteiro n tal que $|M_n|/\varphi(n) = 1/4$, como é o caso do 9 e do 91, onde temos $|M_n| = 18$, como vimos no exemplo 2.49, e $\varphi(n) = 72$.

2.3.3 TESTE PRIMALIDADE DE AGRAWAL–BISWAS

O algoritmo aleatorizado para primalidade proposto por Agrawal e Biswas (2003) é menos eficiente que o teste de Miller–Rabin e a probabilidade de erro é maior, entretanto esse algoritmo tem a sua importância histórica pois foi o ponto partida para Agrawal, Kayal e Saxena construírem o algoritmo determinístico de tempo polinomial: "o novo algoritmo é simplesmente uma desaleatorização do nosso algoritmo. Isto pode ser feito da seguinte maneira. Nosso algoritmo aleatorizado é baseado em testes de identidade $(1 + x)^n = 1 + x^n$ modulo um polinômio g escolhido aleatoriamente, de grau $\lceil \log n \rceil$ e acerta com probabilidade pelo menos 2/3. O espaço amostral de g é claramente de tamanho exponencial $(n^{\lceil \log n \rceil})$. Foi mostrado em Agrawal et al. [2002] que o espaço amostral pode ser reduzido para $O(\log^4 n)$, sem reduzir a probabilidade de sucesso!" (Agrawal e Biswas, 2003).

Os algoritmos probabilísticos eficientes conhecidos até o momento para teste de primalidade baseiam-se no pequeno teorema de Fermat. O presente algoritmo é basado numa generalização do teorema de Fermat

$$n \text{ é primo se, e somente se, } (x + a)^n \equiv x^n + a \pmod n \text{ para todo } 1 \leqslant a \leqslant n - 1.$$

Por exemplo, $(x+1)^5 = x^5 + x^4 + 10x^3 + 10x^2 + 5x + 1$ que módulo 5 fica $x^5 + 1$. Agora, $(x+1)^6 = x^6 + 6x^5 + 15x^4 + 20x^3 + 15x^4 + 6x^2 + 1$ que módulo 6 fica $x^6 + 3x^4 + 2x^3 + 3x^2 + 1 \neq x^6 + 1$.

Teorema 2.53 *Sejam a e n inteiros coprimos. Então, $n \geqslant 2$ é primo se e somente se*

$$(x + a)^n = x^n + a \text{ no anel } \mathbb{Z}_n[x]. \tag{2.17}$$

Demonstração. Se n é primo, então n divide $\binom{n}{i}$ para todo $i \in \{1, 2, \ldots, n-1\}$, portanto, usando o binômio de Newton,

$$(x+a)^n = \sum_{i=0}^{n} \binom{n}{i} x^n a^{n-i} = \binom{n}{0} x^n + \binom{n}{n} a^n = x^n + a^n$$

no $\mathbb{Z}_n[x]$ (2.17) segue do teorema de Fermat.

Por outro lado, se $n > 1$ é composto, seja p^k a maior potência de um fator primo p de n. Então $n = p^k c$ e

$$\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1} = p^{k-1} c \binom{n-1}{p-1}.$$

Supondo que p^k divide o lado direito dessa igualdade, temos que $p | \binom{n-1}{p-1}$, um absurdo, portanto, $\binom{n}{p} \not\equiv 0 \pmod{p^k}$, logo $\binom{n}{p} \not\equiv 0 \pmod{n}$. Como $n \nmid a$, também $n \nmid a^{n-p}$, ou seja, o coeficiente $\binom{n}{p} a^{n-p}$ de x^p em $(x+a)^n$ é não nulo, logo, $(x+a)^n \neq x + a^n$ nesse anel. □

O teste dado pelo teorema 2.53 não é prático porque computar os coeficientes da expansão de $(x+1)^n$ na equação (2.17) toma tempo $\Omega(n)$. Uma alternativa seria usar o teste probabilístico para identidade de polinômio. No entanto, isso falha por dois motivos. Primeiro é que se n não é primo, então $\mathbb{Z}_n$ não é um corpo como assumimos em nossa análise na seção 2.2.3. O segundo é que o grau do polinômio é $n = |\mathbb{Z}_n|$ e, portanto, muito grande pois o teorema de Schwartz–Zippel requer que os valores sejam escolhidos de um conjunto de tamanho estritamente maior que o grau.

Outra alternativa é testar a igualdade módulo um polinômio com grau cuidadosamente escolhido. A alternativa proposta por Agrawal e Biswas (2003) é sortear um polinômio de baixo grau $h \in \mathbb{Z}_n[x]$ e usar o teste

$$\text{se } n \text{ é primo, então } (x+a)^n \equiv x^n + a \pmod{x^r - 1, n} \text{ para todo } 1 \leqslant n \leqslant n-1 \text{ e todo } r \in \mathbb{N},$$

ou seja, testamos $(x+a)^n \equiv x^n + a$ no anel quociente $\mathbb{Z}_n[x]/(h)$. Tomando $h = x^r - 1$ comparamos $(x+a)^n \pmod{x^r-1}$ com $x^n + a \pmod{x^r - 1}$. No cálculo da expansão da potência $(x+a)^n$ módulo h os coeficientes não excedem n e x^t é trocado por $x^{t \bmod r}$, portanto, os graus dos polinômios podem ser mantidos baixo no desenvolvimento do cálculo da potência $(x+a)^n$. Para o tempo de execução ser polinomial devemos ter $r = \log^{O(1)} n$.

Instância : inteiro $n > 3$.
Resposta : n é *composto*, ou *primo* com probabilidade de erro $< 1/2$.

1 **se** n *é da forma* a^b **então responda** *composto*.
2 $h \leftarrow_R \{p\colon p$ é um polinômio mônico de grau $\lceil \log n \rceil$ em $\mathbb{Z}_n[x]\}$;
3 **se** h *divide* $(x+1)^n - (x^n + 1)$ *no* $\mathbb{Z}_n[x]$ **então responda** *primo*.
4 **senão responda** *composto*.

Algoritmo 24: teste de primalidade Agrawal–Biswas.

Vamos agora analisar o algoritmo. Na linha 1 testamos

1 **para** b *de* 2 *até* $\log_2 n$ **faça** $P \leftarrow \lfloor n^{\frac{1}{b}} \rfloor$
2 **se** $P^b = 1$ **então responda** *sim*.
3 **senão responda** *não*.

pois se n é da forma a^b, então $2 \leqslant b \leqslant \log_2(n)$. Há várias maneiras de determinar uma raiz b de n, uma delas usa busca binária e é descrita no exercício 2.60, página 85, com tempo de execução[6] $O(\log^3 n)$. Na linha 2, escolhemos

[6] De fato, esse teste pode ser realizado em tempo menor que $c \log^{1+\varepsilon} n$ para qualquer $\varepsilon > 0$ e alguma constante positiva c, ou seja, praticamente linear em $\log(n)$ (Bernstein, 1998).

$\lceil \log n \rceil$ coeficientes no $\mathbb{Z}_n$ uniforme e independentemente, cada um de tamanho $O(\log n)$, portanto a linha contribui com $O(\log^2 n)$ para o tempo de execução do algoritmo. Na linha 3, $x^n + a$ em $\mathbb{Z}_n[x]/(h)$ pode ser representado por $x^{n \bmod r} + a$ (verifique), onde $r = \lceil \log n \rceil$ é o grau de h. Para computar a expansão de $(x+a)^n$ módulo h podemos adaptar facilmente o algoritmo 7 para exponenciação modular da seguinte forma

$P(x) \leftarrow 1$;
$n = b_{\ell-1} \dots b_0$ em binário;
para k *de* $\ell - 1$ *até* 0 **faça**
  $P(x) \leftarrow P(x)^2$;
  se $b_k = 1$ **então** $P(x) \leftarrow P(x)(x+a)$;
  $P(x) \leftarrow P(x) \bmod (h(x), n)$;
responda $P(x)$.

Temos $\ell = O(\log n)$ rodadas do laço. No caso $b_k = 0$, como $P(x)$ tem grau no máximo $r-1$ temos

$$P(x)^2 = \sum_{j=0}^{2r-2} a_j x^j \quad = \quad \sum_{j=0}^{r-1} a_j x^j + \sum_{j=r}^{r-2} a_j x^{q_j r + j \bmod r} = \sum_{j=0}^{r-1} a_j x^j + \sum_{j=0}^{r-2} a_{j+r} x^j (x^r)^{q_j}$$

O produto $P(x)^2$ custa $O((\log n)^4)$, são $O(r^2)$ multiplicações e somas de números de tamanho $O(\log n)$. Como $x^r \equiv 1 \pmod h$, se fizermos $a_{2r-1} = 0$ então podemos escrever

$$P(x)^2 \equiv \sum_{j=0}^{r-1} (a_j + a_{j+r}) x^j \pmod h.$$

O resto módulo n custa $O(\log^2 n)$, portanto custo para determinar o lado direito na equação acima é $O(r \log^2 n) = O((\log n)^3)$. O tempo da linha 3 é $\ell O((\log n)^4) = O((\log n)^5)$, portanto, do teste de Agrawal–Biswas tem tempo de execução $O(\log^5 n)$.

Quanto à correção, se n é primo então $(x+a)^n = x^n + a$ de modo que qualquer h divide a diferença, logo algoritmo responde *primo* com probabilidade 1. Para limitar a probabilidade de erro, primeiro assumimos que todo fator primo de n é maior que 16. Isso não estraga o projeto porque o algoritmo pode ser facilmente modificado para testar divisibilidade de n por primos pequenos. Uma modificação mais drástica é que para simplificar provaremos um limitante pior.

Instância : inteiro $n > 3$.
Resposta : n é *composto*, ou *primo* com probabilidade de erro $< 1 - 0{,}49/\log n$.

1 **se** $n \in \{2,3,5,7,11,13\}$ **então responda** *primo*.
2 **se** $a \in \{2,3,5,7,11,13\}$ *divide* n **então responda** *composto*.
3 **se** n *é da forma* a^b **então responda** *composto*.
4 $h \leftarrow_R \{p\colon p$ é um polinômio mônico de grau $\lceil \log n \rceil$ em $\mathbb{Z}_n[x]\}$;
5 **se** h *divide* $(x+1)^n - (x^n+1)$ *no* $\mathbb{Z}_n[x]$ **então responda** *primo*.
6 **senão responda** *composto*.

Em $10\lceil \log n \rceil$ testes independentes a probabilidade de sucesso é maior que $1 - 1/e^5 \approx 0{,}99$ de sucesso.

Agora, consideramos n composto que não é potência de primo e não tem divisor primo menor que 16, ou seja, a execução passou pelas três primeiras linhas do algoritmo acima.

Como n é composto, o polinômio

$$P(x) = (x+1)^n - (x^n + 1)$$

é não nulo no $\mathbb{Z}_n[x]$ e também é não nulo no $\mathbb{Z}_p[x]$ para todo p que divide n. De fato, para a maior potência do primo p que divide n, digamos que p^m, o coeficiente de x^{p^m} na expansão de $P(x)$ é não nulo no $\mathbb{Z}_p$, pois p não divide $\binom{n}{p^m}$ (verifique).

Fixemos $p > 16$ um fator primo de n. Vamos mostrar que há uma probabilidade positiva de, ao escolher h como acima, termos $P(x) \not\equiv 0 \pmod h$ no $\mathbb{Z}_p[x]$ e, portanto, $P(x) \not\equiv 0 \pmod h$ no $\mathbb{Z}_n[x]$.

Primeiro, observemos que se escolhemos h mônico de grau $\lceil \log n \rceil$ em $\mathbb{Z}_n[x]$ e em seguida realizamos operações módulo p, então o resultado é o mesmo que escolher h mônico de grau $\lceil \log n \rceil$ em $\mathbb{Z}_p[x]$. Ainda, cada polinômio mônico de grau r do $\mathbb{Z}_p[x]$ ocorre no $\mathbb{Z}_n[x]$ exatamente $(n/p)^r$ vezes (por quê?), logo o sorteio dá um polinômio em $\mathbb{Z}_p$ com probabilidade uniforme.

O polinômio mônico não nulo $P(x) \in \mathbb{Z}_p[x]$ de grau n tem uma fatoração única como produto de polinômios mônicos irredutíveis e no máximo n/r fatores podem ter grau r. Se h for irredutível, então ou é igual a qualquer um desses n/r fatores e nesse caso o algoritmo responde errado ou é diferente do fatores e $P(x) \pmod h$ não é zero, nesse último caso o algoritmo responde corretamente, n é *composto*.

A probabilidade de h ser uma testemunha de que n é composto é a probabilidade do evento

$$h(x) \text{ é irredutível e não é um fator irredutível de } P(x)$$

cuja probabilidade é $\mathbb{P}[\, h(x) \text{ é irredutível}\,] - \mathbb{P}[\, h(x) \text{ não é um fator irredutível de } P(x)\,]$. Do lema 2.38, página 65, e $p > 16$ temos probabilidade $I(r)/p^r > 1/(2r)$ de sortear h irredutível. A probabilidade de que h é um fator irredutível de $P(x)$ é no máximo $(n/r)/p^r$ e assim a probabilidade de encontrar um h que seja irredutível e não divida $P(x)$ é pelo menos

$$\frac{1}{2r} - \frac{n}{rp^r} \geqslant \frac{1}{r}\left(\frac{1}{2} - \frac{n}{16^{\log n}}\right) > \frac{49}{100r} = \Omega\left(\frac{1}{\log n}\right)$$

pois $n > p > 16$ e $r = \lceil \log n \rceil \geqslant 3$.

Estimativa com probabilidade de erro constante para $n > 100$ Assuma $n > p > 100$. Assumimos que n é testado na força bruta contra os cem primeiros números primos: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.

Seja Q um polinômio mônico irredutível do $\mathbb{Z}_p[x]$ com grau entre $1 + r/2$ e r e seja C_Q o conjunto dos polinômios de grau r que têm Q como fator, logo $|C_Q| = p^{r-\partial Q}$. Ademais, $C_Q \cap C_{Q'} = \varnothing$ para polinômios $Q \neq Q'$, caso contrário seriam dois fatores com grau maior que $r/2$ cada, um absurdo. Somando sobre todo $Q \in \mathbb{Z}_p[x]$ irredutível mônico com grau entre $1 + r/2$ e r, a quantidade de tais polinômios é, usando o lema 2.38,

$$\sum_Q |C_Q| = \sum_{i=1+r/2}^{r} I(i)p^{r-i} \geqslant \sum_{i=1+r/2}^{r} \left(\frac{p^i - 2\sqrt{p^i}}{i}\right) p^{r-i} \geqslant \sum_{i=1+r/2}^{r} \left(\frac{1}{i} - \frac{1}{\sqrt{p^i}}\right) p^r$$

Usando que o N-ésimo número harmônico $H_N := \sum_{i=1}^{N} 1/i$ satisfaz (Graham, Knuth e Patashnik, 1994, seção 6.3)

$$H_N = \log(N) + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + \frac{\varepsilon_N}{120N^4},$$

com $0 < \varepsilon_N < 1$ e $\gamma \approx 0{,}5772156649$, a constante de Euler–Mascheroni[7], deduzimos que

$$\sum_{i=1+r/2}^{r} \frac{1}{i} = H_r - H_{r/2} > \log 2 - \frac{1}{2r} + \frac{3}{12r^2} - \frac{16}{120r^4} \geqslant \log 2 - \frac{3383}{37500}$$

para $r \geqslant 5$. Agora, temos que estima a soma de uma progressão geométrica

$$\sum_{i=1+r/2}^{r} \left(\frac{1}{\sqrt{p}}\right)^i = \frac{(1/\sqrt{p})^{r/2} - (1/\sqrt{p})^r}{\sqrt{p} - 1}.$$

[7] Essa constante é o limite de $H_N - \log N$ quando $N \to \infty$.

De $n > p > 100$ temos $r = \lceil \log n \rceil \geqslant 5$ e

$$\sum_{i=1+r/2}^{r} \left(\frac{1}{\sqrt{p}}\right)^i < \frac{1}{9}\left(\left(\frac{1}{10}^{r/2}\right) - \left(\frac{1}{10}\right)^r\right) < \frac{1}{9}\left(\frac{1}{10^{5/2}} - \frac{1}{10^5}\right)$$

Logo, a quantidade de polinômios de grau r com algum fator irredutível de grau entre $r/2+1$ e r é

$$\sum_{Q} |C_Q| > \left(\log 2 - \frac{3383}{37500} - \frac{1}{9}\left(\frac{1}{10^{5/2}} - \frac{1}{10^5}\right)\right)p^r > 0{,}6p^r$$

e a probabilidade de sortear h como um deles é $> 0{,}6$. Como o grau de Q é pelo menos $r/2+1$ temos $|C_Q| \leqslant p^{(r/2)-1}$ e no máximo $(n/(r/2))|C_Q|$ desses polinômios dividem $P(x)$ e a probabilidade de sortear um deles é menor que

$$\frac{1}{p^r}\frac{2n}{r}p^{(r/2)-1} = \frac{2n}{rp^{(r/2)+1}} \leqslant \frac{2n}{5 \cdot 10^{\log n}} < 0{,}001$$

para todo $n > 100$. Assim, a probabilidade com que sorteamos h como um desses Q que não divide $P(x)$, portanto uma testemunha de que n é composto, é pelo menos $0{,}6 - 0{,}001 = 0{,}599$.

2.3.4 GERADOR DE NÚMEROS PRIMOS

O nosso próximo exemplo é um algoritmo aleatorizado para escolher um número primo. Para primos pequenos, com a tecnologia do início do século 21, números até 10^{12} são rapidamente gerados por boas implementações do Crivo de Eratóstenes. Para gerar primos grandes, podemos usar o seguinte algoritmo.

Instância : inteiro positivo n.
Resposta : um primo escolhido uniformemente em $\{n+1,\ldots,2n\}$.

1 **repita** $m \leftarrow_R \{n+1, n+2, \ldots, 2n\}$ **até que** Teste_Primo(m) =*sim*;
2 **responda** m.

Algoritmo 25: gerador de números primos aleatórios.

O postulado de Bertrand garante que, para todo $n > 3$, existe pelo menos um primo em $\{n+1,\ldots,2n-1\}$. Esse postulado foi provado por Chebyshev que garantiu que se $\pi(n)$ é a quantidade de primos menores ou iguais a n, então

$$\frac{1}{3}\frac{n}{\log n} < \pi(2n) - \pi(n) < \frac{7}{5}\frac{n}{\log n} \tag{2.18}$$

onde $\pi(n)$ é a quantidade de primos menores ou iguais a n (Ribenboim, 1996). Logo uma escolha aleatória em $\{n+1,\ldots,2n\}$ tem probabilidade $\rho(n) = (\pi(2n) - \pi(n))/n$ de resultar um primo e

$$\frac{1}{3\log n} < \rho(n) < \frac{7}{5\log n}. \tag{2.19}$$

Nas aplicações em criptografia, por exemplo, queremos números primos cada vez maiores em função da tecnologia da época. No início do século 21 precisamos de primos com 2.048 bits pelo menos. Nesses casos, os algoritmos aleatorizados para primalidade é a melhor opção, senão a única viável, para os teste. A equação (2.18) garante que existem $\approx 2^k/k$ primos com k bits para todo $k \geqslant 2$. Assim, a probabilidade de um sorteio (uniforme) no intervalo $\{2^{k-1},\ldots,2^k-1\}$ produzir um primo é $\approx 1/(2k)$, logo o número esperado de sorteios até encontrar um provável primo é $O(k)$.

No caso do teste de Miller–Rabin para primalidade, vimos que a probabilidade de erro em declarar um número composto como primo é $< (1/4)^t$. Isso não significa que a probabilidade de erro do algoritmo acima é $< (1/4)^t$ porque devemos levar em conta a distribuição dos primos. Numa iteração do laço, seja C o evento "m, sorteado na linha 2, é

composto" e E o evento "Teste_Primo(m), na linha 3, declara m primo". Então $\mathbb{P}(\,\mathsf{E} \mid \mathsf{C}\,) < (1/4)^t$ e queremos $\mathbb{P}(\,\mathsf{C} \mid \mathsf{E}\,)$. Pelo Teorema de Bayes

$$\mathbb{P}(\,\mathsf{C} \mid \mathsf{E}\,) = \frac{\mathbb{P}(\,\mathsf{E} \mid \mathsf{C}\,)\mathbb{P}(\mathsf{C})}{\mathbb{P}(\mathsf{E})} \leqslant \frac{\mathbb{P}(\,\mathsf{E} \mid \mathsf{C}\,)}{\mathbb{P}(\mathsf{E})} < \frac{(1/4)^t}{1/(3k)} = \frac{3k}{4^t}$$

pois $\mathbb{P}(\mathsf{E}) > 1/(3k)$ por (2.19).

Observação 2.54. Na prática, após sortear m é mais eficiente testar divisibilidade de m por inteiros primos até uma constante k antes de testar primalidade, pois é relativamente grande a quantidade de números que tem um divisor pequeno. De imediato números pares podem ser descartados. Testar divisibilidade pelos primos até 256 descarta 80% dos números ímpares com k bits candidatos a primo (Menezes, Oorschot e Vanstone, 1997).

2.4 O JANTAR DOS FILÓSOFOS, UM CASO NÃO-ENUMERÁVEL

Nessa seção vamos considerar um problema de computação distribuída cuja solução probabilística tem espaço amostral não enumerável; o tratamento que daremos a ambos os tópicos, computação distribuída e espaço não enumerável, será informal.

O jantar dos filósofos é um problema originalmente proposto por Dijkstra em 1965 e ilustra o problema de alocação de recursos em sistemas distribuídos. Esse problema não tem solução determinística, entretanto apresentaremos uma solução probabilística devida a Lehmann e Rabin (1981).

A seguinte formulação do problema representa toda uma classe de problemas em computação distribuída:

> Cinco filósofos estão reunidos para um jantar em torno de uma mesa circular. A vida de um filósofo consiste basicamente em pensar. Enquanto está pensando, um filósofo não interage com os outros filósofos, entretanto, o filósofo acaba por sentir fome em algum momento. Para se alimentar, ele dispõe de um prato de macarrão que nunca se esvazia, um garfo a sua esquerda e um garfo a sua direita, mas o macarrão encontra-se de tal forma oleoso que é impossível comê-lo com apenas um garfo, sendo necessários dois para tal. Um filósofo pode pegar apenas um garfo de cada vez e, obviamente, é impossível utilizar um garfo que esteja sendo utilizado por um vizinho. Uma vez que um filósofo tenha dois garfos ele se alimenta, devolve os dois garfos e volta a pensar. Um filósofo faminto e que não seja capaz de pegar seus dois garfos todas as vezes que tente pegá-lo (pois o garfo sempre está em posse de seu vizinho) entra em inanição.

Em suma, um filósofo opera indefinidamente no ciclo: pensar, tentar comer, comer. Para comer um filósofo necessita de acesso exclusivo a dois recursos, cada um deles é compartilhado com um vizinho. O problema computacional consiste em projetar um protocolo que represente os filósofos e os garfos de maneira apropriada, para que se comportem como as entidades descritas no enunciado. O protocolo deve garantir que os filósofos comam.

O modelo probabilístico para esse problema envolve um espaço amostral equivalente ao de lançar uma moeda infinitas vezes. Se temos dois espaços de probabilidade $(\Omega_1, \mathcal{A}_1, \mathbb{P}_1)$ e $(\Omega_2, \mathcal{A}_2, \mathbb{P}_2)$ o espaço produto tem espaço amostral $\Omega_1 \times \Omega_2$ mas o espaço de eventos não é simplesmente $\mathcal{A}_1 \times \mathcal{A}_2$, mas sim a menor[8] σ-álgebra que contém todos os produtos de eventos $\mathsf{A}_1 \times \mathsf{A}_2$, que denotamos por $\mathcal{A}_1 \otimes \mathcal{A}_2$. No produto, a medida de probabilidade é tal que $\mathbb{P}(\mathsf{A}_1 \times \mathsf{A}_2) = \mathbb{P}_1(\mathsf{A}_1)\mathbb{P}_2(\mathsf{A}_2)$ para todos $\mathsf{A}_1 \in \mathcal{A}_1$ e $\mathsf{A}_2 \in \mathcal{A}_2$. O espaço de probabilidade $(\Omega_1 \times \Omega_2, \mathcal{A}_1 \otimes \mathcal{A}_2, \mathbb{P})$ é o *espaço produto*. Essa definição pode ser estendida para o produto de vários espaços, até uma quantia infinita enumerável deles, com $\Omega = \prod_n \Omega_n$ e $\mathcal{A} = \otimes_n \mathcal{A}_n$ é a menor σ-álgebra que contém os eventos $\mathsf{A}_1 \times \mathsf{A}_2 \cdots \mathsf{A}_k \times \Omega_{k+1} \times \Omega_{k+2} \times \cdots$ para todo k, para todo $\mathsf{A}_i \in \mathcal{A}_i$, para todo i. É possível mostrar que há uma única medida de probabilidade $\mathbb{P}$, tal que $\mathbb{P}(\mathsf{A}_1 \times \mathsf{A}_2 \cdots \mathsf{A}_k \times \Omega_{k+1} \times \Omega_{k+2} \times \cdots) = \mathbb{P}_1(\mathsf{A}_1)\mathbb{P}_2(\mathsf{A}_2) \cdots \mathbb{P}_k(\mathsf{A}_k)$.

[8] É a intersecção de todas as σ-álgebras de $\Omega_1 \times \Omega_2$. Note-se que a intersecção de σ-álgebras de $\Omega_1 \times \Omega_2$ é uma σ-álgebra de $\Omega_1 \times \Omega_2$.

Exemplo 2.55. Consideremos o espaço amostral formado por todas as sequências binárias

$$\{0,1\}^{\mathbb{N}} := \{(b_0, b_1, b_2, \dots) \colon b_i \in \{0,1\}\ (\forall i)\}.$$

Denotemos por $\mathcal{C}_k$ a família de todos os eventos de $\{0,1\}^{\mathbb{N}}$ cuja ocorrência é decidida pelos k primeiros bits das sequências. Por exemplo, as sequências tais que $b_1 \neq b_2$ e $b_3 = 0$ é um elemento de $\mathcal{C}_3$; o evento "dois zeros nos sete primeiros lançamentos" é um elemento de $\mathcal{C}_7$. Dado subconjunto $B \subset \{0,1\}^k$ definimos $B_\Omega \subset \{0,1\}^{\mathbb{N}}$ por

$$B_\Omega := \{(b_0, b_1, b_2, \dots) \colon (b_1, b_2, \dots, b_k) \in B\}.$$

O conjunto B_Ω pode ser identificado com $B \times \{0,1\}^{\mathbb{N}}$, temos $B_\Omega \in \mathcal{C}_k$ e todo elemento de $\mathcal{C}_k$ pode ser escrito dessa forma para algum $B \subset \{0,1\}^k$. A família $\mathcal{C}_k$ é uma σ-álgebra de subconjuntos de $\{0,1\}^{\mathbb{N}}$, para cada inteiro positivo k, e no jargão de Probabilidade, esses são chamados de *eventos cilíndricos*. Notemos que $\mathcal{C}_k \subset \mathcal{C}_{k+1}$ e que o evento "não ocorre 1" não pode ser expresso por nenhuma dessas famílias.

Agora, fazemos

$$\mathcal{C} := \bigcup_{k \geqslant 1} \mathcal{C}_k$$

a família dos eventos cuja ocorrência é decidida por um número fixo de bits iniciais. A família $\mathcal{C}$ não é uma σ-álgebra de subconjuntos de $\{0,1\}^{\mathbb{N}}$ pois se tomamos B_k o conjunto das sequências com $b_k = 1$ então temos $B_k \in \mathcal{C}_k$ mas $\overline{\bigcup_k B_k} \notin \mathcal{C}$. Entretanto, $\mathcal{C}$ é uma *álgebra* de subconjuntos de $\{0,1\}^{\mathbb{N}}$, isto é, satisfaz: (i) $\varnothing$ é um elemento da família, (ii) o complemento de um elemento da família também pertence a família e (iii) a união de dois elementos da família pertence a família. Um teorema famoso, conhecido como *Teorema de Extensão de Carathéodory* nos diz que, nesse caso, uma função $P \colon \mathcal{C} \to [0,1]$ que satisfaz (i) $P(\{0,1\}^{\mathbb{N}}) = 1$ e (ii) $P(\bigcup_n B_n) = \sum_n P(B_n)$, para $\{B_n\}_{n\in\mathbb{N}}$ elementos disjuntos da álgebra, pode ser estendida de maneira única para uma medida de probabilidade sobre a menor σ-álgebra que contém a álgebra $\mathcal{C}$.

O próximo passo é definir P de acordo com as hipóteses do parágrafo anterior. Todo $A \in \mathcal{C}$ é da forma $B \times \{0,1\}^{\mathbb{N}}$ para algum $B \subset \{0,1\}^k$, para algum natural k. Definimos

$$P(A) := \frac{|B|}{2^k}. \tag{2.20}$$

Essa definição é consistente (veja exercício 2.76 no final desse capítulo) e para tal P vale $P(\{0,1\}^{\mathbb{N}}) = 1$ (por quê?) e é enumeravelmente aditiva (isso é bastante difícil de provar, veja o exercício 2.77 no final do capítulo) portanto, como vimos no parágrafo anterior, pode ser estendida para uma medida de probabilidade $\mathbb{P}$ sobre a menor σ-álgebra que contém $\mathcal{C}$.

Nesse espaço de probabilidade os pontos amostrais têm probabilidade zero. Dado $(b_0, b_1, b_2, \dots) \in \{0,1\}^{\mathbb{N}}$, definimos o evento $E_k := \{(\omega_0, \omega_1, \dots) \in \{0,1\}^{\mathbb{N}} \colon \omega_j = b_j \text{ para todo } j \leqslant k\}$ para todo natural k, logo $(b_0, b_1, b_2, \dots) = \bigcap_{k\in\mathbb{N}} E_k$ e a probabilidade do ponto amostral é o limite de $\mathbb{P}(E_k)$ quando $k \to \infty$ pela continuidade de $\mathbb{P}$. Usando a equação (2.20) essa probabilidade é $\lim_{k\to\infty}(1/2)^k = 0$. Como consequência da aditividade, eventos enumeráveis têm probabilidade 0.

Esse espaço de probabilidade que acabamos de definir é equivalente a distribuição uniforme no intervalo $[0,1]$, descrito no exemplo 1.9, página 11. Para os detalhes dessa construção e da equivalência convidamos o leitor a consultar o capítulo 1 de Billingsley (1979). ◊

Definições preliminares No modelo que usaremos para representar computação distribuída é a computação é realizada pela execução de um conjunto de *processos* concorrentes, cada processo executa um algoritmo. Cada filósofo corresponde a um processo e seu algoritmo define as ações dos filósofos. Uma *ação atômica* é qualquer conjunto de instruções de um algoritmo distribuído executadas de modo indissociável, nenhum outro processo executa instrução enquanto uma ação atômica não termina. *Variáveis* representam os garfos e são compartilhadas, sendo cada garfo modelado por um espaço

de memória acessível apenas aos processos que representam os filósofos que o compartilham; pegar e devolver um garfo são mudanças no valor de uma variável. O acesso às variáveis é uma ação atômica, um filósofo verifica se um garfo está disponível e, caso disponível, o pega sem que seja incomodado por algum de seus vizinhos nesse ínterim. Ademais,

> *é garantido que sempre que um processo requisita o conteúdo de uma variável compartilhada, ele acabará por recebê-lo em algum momento futuro.* (†)

Um *escalonamento* é uma função que define, a partir do comportamento passado de todos os processos, o próximo processo a efetuar uma ação atômica. Conhecer o passado dos processos inclui conhecer os resultados de sorteios aleatórios passados, as memórias compartilhadas e privadas dos processos. Não há nenhum tipo de hipótese em relação às taxas de atividade de cada processo. Não está excluída a possibilidade de que o escalonamento seja malicioso e trabalhe contra a solução, fazendo o máximo possível para impedir que os filósofos se alimentem. Um escalonamento é *justo* se

> *todos os processos são ativados um número infinito de vezes* (‡)

qualquer que sejam os resultados de sorteios aleatórios. Daqui em diante só consideramos escalonamentos justos.

Uma *solução* para o problema do jantar dos filósofos deve ser

- *distribuída*: não há um processo controlador ou uma memória central com a qual todos os outros processos possam se comunicar;
- *simétrica*: todos os processos devem executar o mesmo algoritmo e todas as variáveis têm a mesma inicialização. Além disso, os processos ignoram suas identidades.

O objetivo é encontrar um protocolo de ação que, respeitando as restrições acima, garanta que os filósofos se alimentem. Uma computação em *deadlock* é uma computação em que existe um instante t no qual um filósofo está tentando comer, mas a partir do qual nenhum filósofo come.

Não há solução determinística Para esse problema não há uma solução que seja implementada por algoritmos distribuídos determinísticos. Suponha que exista uma solução distribuída e simétrica e vamos definir um escalonamento que impeça os filósofos de se alimentarem. Sem perda de generalidade, podemos enumerar os processos de 1 a n. Basta que o escalonamento ative cada um dos processos por uma ação atômica, ordenadamente, e repita essa ordem de ativação indefinidamente. Considerando-se que os processos se encontram inicialmente no mesmo estado, a simetria é preservada a cada rodada e é impossível que todos os filósofos estejam se alimentando simultaneamente, logo temos um escalonamento que impede que todos os filósofos se alimentem.

Solução probabilística O que impede uma solução determinística para o problema do jantar dos filósofos é a simetria entre os processos. Para quebrar a simetria, vamos equipar os filósofos com moedas, permitindo que escolham aleatoriamente qual dos dois garfos tentarão pegar. A cada instante t o processo ativo tem a sua disposição um bit aleatório b_t com probabilidade 1/2 de ser qualquer um dos dois valores e de modo que em instantes distintos os valores dos bits são independentes. O espaço amostral $\{0,1\}^{\mathbb{N}}$ é formado de todas as sequências binárias $\omega = (b_0, b_1, b_2, \dots)$. Esse espaço não é enumerável e usaremos o tratamento descrito acima.

Consideraremos o caso de $n \geqslant 3$ filósofos, denotados por P_i, para $1 \leqslant i \leqslant n$, mas sem que eles reconheçam qualquer identidade e dispostos na ordem (cíclica) $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3, \dots, \mathsf{P}_n$ no sentido anti-horário. Os filósofos se comportam da maneira descrita pelo algoritmo 26, no qual representamos por 0 o garfo da esquerda, por 1 o garfo da direita e as linhas são instruções atômicas.

```
1 enquanto verdadeiro faça
2     pense;
3     ℓ ←_R {0,1};
4     se garfo ℓ disponível então pegue o garfo ℓ, senão vá para linha 4;
5     se garfo 1 − ℓ disponível então pegue o garfo 1 − ℓ e vá para linha 7;
6     devolva o garfo ℓ;
7     coma;
8     devolva um garfo;
9     devolva o outro garfo.
```

Algoritmo 26: Algoritmo dos Filósofos

Um escalonamento S e uma sequência infinita de bits $\omega = (b_i : i \in \mathbb{N})$ de $\{0,1\}^{\mathbb{N}}$ definem uma, e só uma, computação, que é uma sequência infinita de ações atômicas do algoritmo 26

$$\text{COMP}(S, \omega) := ((\alpha, P, b)_t : t \in \mathbb{N})$$

em que α é a ação atômica efetuada pelo filósofo P no instante t para a qual há a disposição um bit aleatório $b = b_t \in \{0,1\}$ que pode ser usado ou não.

Para um escalonamento S fixo COMP induz uma distribuição de probabilidade no espaço de todas as computações. O objetivo é demonstrar que no sistema dos filósofos com algoritmos aleatorizados a probabilidade de ocorrência *deadlock* é zero. Em particular, fixado S temos que $\omega \in \{0,1\}^{\mathbb{N}}$ define se a computação está ou não em *deadlock* de modo que $\{\omega \in \{0,1\}^{\mathbb{N}} : \text{COMP}(S, \omega)$ em *deadlock*$\}$ é um evento aleatório pois depende de uma quantidade finita de bits iniciais de ω.

Em uma computação em *deadlock* não é possível que todos os filósofos peguem algum garfo um número finito de vezes pois, nesse caso, se os dois vizinhos de um filósofo P pegam seus garfos apenas um número finito de vezes, então para P os garfos estarão sempre disponíveis a partir de um determinado momento, logo ele pega seus garfos um número infinito de vezes já que a computação é justa, pela hipótese (‡), de modo que a partir de um determinado instante P come sempre que deseja.

Em uma computação em *deadlock* não é provável que algum filósofo pegue seus garfos apenas um número finito de vezes enquanto seu vizinho pegue seus garfos infinitas vezes. Assumamos, sem perda da generalidade, que P_2 é um filósofo que pega seus garfos apenas um número finito de vezes e P_1 um filósofo que pega seus garfos infinitas vezes. Se P_2 pega seus garfos apenas um número finito de vezes, então existe um instante t_0 a partir do qual P_2 não pega mais seus garfos. Em particular, o garfo a direita de P_1 estará disponível para P_2. Logo, para todo $t > t_0$, caso P_1 sorteie o garfo a sua esquerda ele certamente se alimentará pois o garfo da esquerda estará disponível em algum momento futuro, por (†), e o garfo da direita está sempre disponível. Se P_1 sorteia o garfo esquerdo um número finito de vezes, podemos enumerar os casos em que isso acontece, identificando cada caso pela sequência de sorteios usados por P_1 até a última vez que pega seu garfo esquerdo, ou seja, temos um evento enumerável F donde concluímos que não-F ocorre com probabilidade 1, ou seja, P_1 sorteia o garfo esquerdo infinitas vezes e, portanto, se alimenta infinitas vezes.

A partir dos dois parágrafos acima concluímos o seguinte.

PROPOSIÇÃO 2.56 *Numa computação em* deadlock *todos os filósofos pegam algum dos seus garfos um número infinito de vezes com probabilidade* 1. □

Lembremos que em cada instante da computação um bit aleatório pode ou não ser usado pelo processo da vez. Para um instante t fixo temos um sequência formada pelos bits aleatórios que foram de fato usados por algum dos processos (no sorteio de um garfo). Chamemos essa sequência de *configuração de sorteios aleatórios* e chamemos duas configurações

A e uma posterior B de *disjuntas* caso entre A e B todos os filósofos utilizaram pelo menos um sorteio.

Lema 2.57 *Numa computação em* deadlock, *se para um dado instante* t *a configuração de sorteios aleatórios já efetuados é* A, *então com probabilidade* 1 *haverá num momento futuro uma configuração* B *disjunta de* A *em que o último sorteio de algum filósofo foi o garfo esquerdo e o último sorteio de seu vizinho à direita foi o garfo direito.*

Demonstração. Numa computação em *deadlock*, todos os filósofos pegam algum dos seus garfos um número infinito de vezes com probabilidade 1 pela proposição 2.56.

Se A e B são duas configurações disjuntas e subsequentes então, no instante que ocorre B, a probabilidade com que o último sorteio de cada filósofo sejam iguais é $2(1/2)^n = 1/2^{n-1}$. Agora, se consideramos um intervalo de k configurações disjuntas subsequentes a partir de uma dada configuração, digamos $A_i, A_{i+1}, \ldots, A_{i+k}$, a probabilidade de que todos os filósofos tenham sorteado o mesmo valor em todos os respectivos últimos sorteios que antecedem imediatamente alguma configuração A_j, com $i < j \leqslant i+k$, é de $(1/2^{n-1})^k$. A probabilidade desse evento ao longo da computação é $\lim_{k\to\infty}(1/2^{n-1})^k = 0$, assim a partir de qualquer configuração A surgirá, com probabilidade 1, uma configuração disjunta B tal que, considerando o último sorteio de todos os filósofos, haverá algum filósofo que sorteou 0 (garfo da esquerda) e seu vizinho à direita sorteou 1 (garfo da direita). □

Lema 2.58 *Seja* F *um segmento inicial finito de uma computação composto por* t *instantes e tal que no instante* t *temos:* (i) *tanto* P_1 *quanto* P_2 *estão tentando comer,* (ii) *o último sorteio de* P_1 *foi o garfo esquerdo e o último sorteio de* P_2 *foi o garfo direito. Considere todas as computações* $C = \text{comp}(S, \omega)$ *que sejam continuações de* F. *Nessas condições, em* C *pelo menos um dentre* P_1 *e* P_2 *se alimenta antes da próxima configuração disjunta da atual com probabilidade* 1.

Demonstração. No instante t os filósofos P_1 e P_2 estão tentando comer, P_1 sorteou 0 e P_2 sorteou 1 (estão na linha 4 do algoritmo 26), então antes do próximo sorteio cada um deles pode se encontrar em um dos seguintes estados:

1. o filósofo está esperando que o garfo sorteado seja disponibilizado, ou
2. o filósofo está em posse do garfo sorteado.

Se algum dos filósofos, dentre P_1 e P_2, está no estado 2 então um deles irá comer antes do próximo sorteio. De fato, no caso em que tanto P_1 quanto P_2 se encontram no estado 2 o próximo filósofo a ser ativado irá se alimentar antes do seu próximo sorteio pois encontrará o garfo compartilhado pelos dois disponível. No caso em que P_1 se encontra no estado 2 e P_2 no estado 1, se P_1 for o próximo dentre os dois a ser ativado, ele encontrará o garfo compartilhado disponível e comerá antes de ter feito algum sorteio; se P_2 for ativado, ele pode tanto permanecer no estado 1 e voltamos para a condição inicial, quanto progredir para o estado 2 e recaímos no caso anterior. Finalmente, o caso em que P_1 se encontra no estado 1 e P_2 no estado 2 é análogo ao anterior.

Por outro lado, no caso em que ambos os filósofos se encontram no estado 1, antes de algum sorteio de algum deles, um deverá avançar para o estado 2 e recaímos nos casos acima. □

Com as propriedades dadas nos lemas acima provaremos o resultado final desse capítulo. Seja S um escalonamento justo. Denotemos por D o evento "a computação $\text{comp}(D, \omega)$ está em *deadlock*" e suponha que $\mathbb{P}(D) > 0$. Podemos então nos referir às probabilidades dos eventos condicionados ao *deadlock*. Pelo lema 2.57, com probabilidade 1 ocorre uma sequência infinita, digamos $A_1, A_2, \ldots, A_n, \ldots$, de configurações disjuntas de sorteios aleatórios satisfazendo as hipóteses do lema 2.58. Pelo lema 2.58, algum filósofo come entre A_n e A_{n+1}, para todo n, com probabilidade 1. Chegamos então à conclusão de que, condicionado ao evento "computação em *deadlock*", computações livres de *deadlock* têm probabilidade 1. Desta maneira, a ocorrência de deadlock deve ter probabilidade zero.

Teorema 2.59 *Para todo escalonamento* S *justo,* $\text{comp}(S, \omega)$ *está em* deadlock *com probabilidade* 0. □

2.5 EXERCÍCIOS

EXERCÍCIO 2.60. Um algoritmo para testar se n é da forma a^b é com segue: seja k tal que $2^{k-1} \leqslant n < 2^k$, então uma b-ésima raiz de n pertence ao intervalo $2^{\lfloor (k-1)/b \rfloor} \leqslant n^{1/b} < 2^{\lceil k/b \rceil}$, faça uma busca binária nesse intervalo. Descreva o algoritmo, verifique que usando a estratégia do algoritmo 7 a potência x^y pode ser calculada em tempo $O((y \log(x))^2)$ e conclua que testar se n é da forma a^b com a estratégia acima tem tempo de execução $O((\log n)^3)$.

EXERCÍCIO 2.61. Dez dados equilibrados são lançados. Supondo que os resultados são independentes, use o princípio da decisão adiada para determinar a probabilidade da soma dos resultados ser divisível por seis.

EXERCÍCIO 2.62. Um baralho comum de 52 cartas é embaralhado de modo que a disposição final é qualquer uma dentre as 52! possibilidades com igual probabilidade. Denote por E o evento "a carta do topo é de espadas", que ocorre com probabilidade 1/4. Use o princípio da decisão adiada para provar que $\mathbb{P}(F) = \mathbb{P}(E)$ para F o evento "a quarta carta a partir do topo é espada".

EXERCÍCIO 2.63. Um baralho comum de 52 cartas é embaralhado de modo que a disposição final é qualquer uma as 52! possibilidades com igual probabilidade e em seguida é dividido em 13 montes de 4 cartas cada. Todo monte tem um único rótulo tomado em $\{A, 2, 3, \ldots, 9, 10, J, Q, K\}$ arbitrariamente. No primeiro movimento abrimos uma carta do monte K e o resultado indica o próximo monte donde abriremos uma carta e assim por diante seguimos. O jogo acaba quando uma jogada indica abrir a carta de um monte vazio. Use o princípio da decisão adiada para provar que a probabilidade de abrirmos todas as cartas do baralho é 1/13.

EXERCÍCIO 2.64. Prove que se o laço da linha 1 no algoritmo 9 for executado $n^2\lceil \log(n) \rceil$ vezes, então a probabilidade do algoritmo não encontrar um corte de tamanho $\leqslant k$ é menor que $1/n$.

EXERCÍCIO 2.65 (emparelhamento perfeito em grafos bipartidos). Seja $G = (A \cup B, E)$ um grafo com $|A| = |B| = n$ e todas as arestas em E tem um extremo em A e o outro em B, isto é G é um **grafo bipartido**. Defina a *matriz de adjacências* $A = (a_{i,j})$ pondo

$$a_{i,j} := \begin{cases} x_{i,j} & \text{se } \{a_i, b_j\} \in E \\ 0 & \text{caso contrário.} \end{cases}$$

Um **emparelhamento** M em G é um subconjunto de E formado por arestas não adjacentes, isto é, $e \cap d = \varnothing$ para quaisquer arestas e, $d \in M$ distintas. O emparelhamento M é dito **perfeito** se $|M| = n$.

Prove que G tem um emparelhamento perfeito se, e somente se, $\det(A) \neq 0$ (como polinômios). Escreva um algoritmo baseado no teorema de Schwartz–Zippel que determina um emparelhamento perfeito caso exista. Analise a probabilidade de erro e o tempo de execução do algoritmo.

EXERCÍCIO 2.66. Seja G um grafo bipartido e $\mathcal{F} := \{M_1, M_2, \ldots, M_k\}$ o conjunto dos emparelhamentos perfeitos de G. Tome $p\colon E(G) \to \{1, \ldots, 2|E|\}$ uma atribuição de pesos escolhidos uniformemente e independentemente para as arestas de G e defina o peso de um emparelhamento como a soma dos pesos de suas arestas. Na matriz A do exercício 2.65 tome $x_{i,j} = 2^{p(a_i, b_j)}$. Suponha, sem perda de generalidade, que M_1 é o único emparelhamento de peso mínimo. Prove que a maior potência de 2 que divide $\det(A)$ é o peso de M_1. Prove que para cada aresta $\{a_i, b_j\}$, o determinante da matriz resultante da eliminação da linha i e da coluna j de A vezes $2^{p(a_i,b_j)-p(M_1)}$ é ímpar se, e somente se, $\{a_i, b_j\} \in M_1$. Baseado no lema do isolamento (exercício 1.61), escreva um algoritmo probabilístico que ou devolve um emparelhamento perfeito ou falha. Analise seu algoritmo.

EXERCÍCIO 2.67. Resolva o problema da verificação do produto de matrizes, seção 2.2.2, usando a técnica da seção 2.2.3.

Em particular, use o teorema 2.19 para provar que se sorteamos $v \in S$, para um S adequado, então $\mathbb{P}[vAB = vC] \leqslant 1/|S|$.

EXERCÍCIO 2.68. Neste exercício provaremos o limitante inferior para $\varphi(n)$ um pouco pior que (2.10):

$$\varphi(n) > \frac{n}{4\log n};$$

mas bem mais fácil de se provar. Primeiro, observe que se n tem k divisores primos distintos e $q_1, \ldots, q_k$ são os primeiros k primos, então

$$\varphi(n) = n\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right) \geqslant n\prod_{i=1}^{k}\left(1-\frac{1}{q_i}\right).$$

Primeiro, justifique a desigualdade acima. Em seguida, prove usando a série geométrica que

$$\prod_{i=1}^{k}\frac{q_i}{q_i-1} = \sum \frac{1}{q_1^{i_1} q_2^{i_2}\cdots q_k^{i_k}}$$

onde a soma é sobre toda sequência $(i_1, i_2, \ldots, i_k)$ de inteiros não negativos. Finalmente, prove que soma do item anterior é pelo menos $\log(4n)$.

EXERCÍCIO 2.69. Escreva um algoritmo aleatorizado que recebe um inteiro $M \geqslant 2$ e devolve uma escolha aleatória uniforme $N \in \{1, \ldots, M-1\}$ tal que $\operatorname{mdc}(M, N) = 1$. Analise o algoritmo.

EXERCÍCIO 2.70. Dados inteiros $m_1, m_2, \ldots, m_k > 1$, sejam $m = m_1 m_2 \cdots m_k$ e $m_i' = m/m_i$. Para $a \in \mathbb{Z}_n$, mostre como computar $a^{m_1'}, a^{m_2'}, \ldots a^{m_k'}$ com $O(\log(k)\log(m))$ multiplicações.

EXERCÍCIO 2.71. O seguinte algoritmo é mais eficiente que o que vimos anteriormente para determinar um gerador do $\mathbb{Z}_p^*$. Suponha que são dados inteiros positivos p e $p_1, m_1, p_2, m_2, \ldots, p_k, m_k$ como nas instâncias do algoritmo 16.

1. Dado $a \in \mathbb{Z}_p^*$ mostre como computar a ordem de a em tempo $O(k\log^3 p)$ (dica: lema 2.23).
2. Use o exercício 2.70 para melhorar o tempo para $O(\log k \log^3 p)$.
3. Modifique o algoritmo do item anterior para construir um gerador do $\mathbb{Z}_p^*$ em tempo esperado $O(\log k \log^3 p)$.

EXERCÍCIO 2.72 (Arvind e Mukhopadhyay (2008) e Klivans e Spielman (2001)). Prove a seguinte versão do lema do isolamento (exercício 1.61, página 37). Dados C, ε constantes positivas e $\{\sum_i c_i x_i\}$ uma família de formas lineares distintas para inteiros $0 \leqslant c_i \leqslant C$, para todo i. Se cada x_i é escolhido aleatoriamente em $\{0, 1, \ldots, Cn/\varepsilon\}$, então existe uma única forma linear de valor mínimo com probabilidade $1-\varepsilon$. Use esse resultado para dar um algoritmo probabilístico para o problema da identidade de polinômios.

EXERCÍCIO 2.73. Dados naturais $a > 1$ e $p > 2$ primo que não divide $a^2 - 1$, mostre que

$$\frac{a^{2p}-1}{a^2-1}$$

é pseudoprimo para a base a. Conclua que há infinitos pseudoprimos para qualquer base fixa.

EXERCÍCIO 2.74 (teste de primalidade de Pocklington, 1914). Seja $n = LR+1$, $L > R$ e q fator primo de L. Prove que se para todo q existe um inteiro $a > 1$ tal que $a^{n-1} \equiv 1 \pmod{n}$ e $\operatorname{mdc}(a^{(n-1)/q} - 1, n) = 1$, então n é primo.

EXERCÍCIO 2.75 (teste de primalidade de Proth, 1878). Uma quantidade relevante do maiores primos conhecidos são caracterizados pelo resultado descrito a seguir. Isso se deve a facilidade de implementação desse teste. Prove o seguinte: seja n um natural da forma $n = r2^s + 1$, com $2^s > r$ e r ímpar. Então n é primo se, e só se, existe um inteiro a tal que $a^{(n-1)/2} \equiv -1 \pmod{n}$, então n é primo. Ademais, se a não é um quadrado módulo p vale a recíproca.

EXERCÍCIO 2.76. Prove que a definição de probabilidade dada na equação (2.20), página 81, é consistente, isto é, se existem $B \subset \{0,1\}^k$ e $B' \subset \{0,1\}^{k'}$, com $k \neq k'$, tais que $A := B \times \{0,1\}^{\mathbb{N}} = B' \times \{0,1\}^{\mathbb{N}}$, então $P(A)$ definido na equação (2.20) coincide nas duas representações de A.

EXERCÍCIO 2.77. Em geral, a parte difícil da aplicação do teorema de Carathéodory (descrito informalmente na página 81) é provar que a função que se quer estender é enumeravelmente aditiva. O que se faz, normalmente, é provar que a função é finitamente aditiva e contínua no sentido da seção 1.1.2. Prove que se $\mathbb{P}$ é não-negativa, finitamente aditiva e $\mathbb{P}(\Omega) = 1$ então são equivalentes

1. $\mathbb{P}$ é uma medida de probabilidade.
2. Para toda sequência decrescente $(A_n : n \geqslant 1)$ de elementos de $\mathcal{A}$ tal que $\bigcap_{n\geqslant 1} A_n = \varnothing$ vale que $\lim_{n\to\infty} \mathbb{P}(A_n) = 0$.

EXERCÍCIO 2.78 (algoritmo Las Vegas para raiz quadrada módulo p). O inteiro a é um quadrado módulo p se $x^2 \equiv a \pmod p$ tem solução. Considere o algoritmo 27 abaixo para achar uma raiz de um quadrado no $\mathbb{Z}_p$.

Instância : um primo $p > 2$ e um quadrado a módulo p.
Resposta : uma raiz quadrada de a.

1 **se** $p \equiv 3 \pmod 4$ **então responda** $a^{\frac{p-3}{4}+1} \bmod p$.
2 **se** $p \equiv 1 \pmod 4$ **então**
3 **repita**
4 $b \leftarrow_R \{1, 2, \ldots, p-1\}$;
5 **até que** $b^{\frac{p-1}{2}} \equiv -1 \pmod p$;
6 $i \leftarrow 2 \cdot \frac{p-1}{4}$;
7 $k \leftarrow 0$;
8 **repita**
9 $i \leftarrow \frac{i}{2}$;
10 $k \leftarrow \frac{k}{2}$;
11 **se** $a^i b^k \equiv -1 \pmod p$ **então** $k \leftarrow k + 2 \cdot \frac{p-1}{4}$;
12 **até que** *i seja ímpar*
13 **responda** $a^{\frac{i+1}{2}} b^{\frac{k}{2}} \bmod p$.

Algoritmo 27: raiz quadrada no $\mathbb{Z}_p^*$.

Verifique que a resposta é uma raiz quadrada de a. Prove que

$$\mathop{\mathbb{P}}_{b\in\mathbb{Z}_p^*}\left[b^{\frac{p-1}{2}} \equiv -1 \pmod p\right] = \frac{1}{2}.$$

Não se conhece algoritmo determinístico eficiente para realizar a computação das linhas 3 – 5. Verifique que sem considerar as linhas 3 – 5 o algoritmo tem tempo polinomial em $\log(p)$. Determine o tempo médio de execução do algoritmo.

EXERCÍCIO 2.79 (algoritmo Monte Carlo para raiz quadrada módulo p). No exercício 2.78 foi dado um algoritmo que nunca erra mas no pior caso pode executar para sempre. O algoritmo 28 abaixo modifica o algoritmo 27 transformando-o num algoritmo Monte Carlo, eficiente para determinar uma raiz quadrada módulo p mas que pode errar.

Determine o tempo de execução do algoritmo. Prove que o algoritmo erra somente no caso em que o primeiro laço

Instância : um primo $p > 2$, um quadrado a módulo p e $t \in \mathbb{N}$.
Resposta : uma raiz quadrada de a.

1 **se** $p \equiv 3 \pmod 4$ **então responda** $a^{\frac{p-3}{4}+1} \mod p$.
2 **se** $p \equiv 1 \pmod 4$ **então**
3 **repita**
4 $b \leftarrow_R \{1, 2, \ldots, p-1\}$;
5 **até que** $b^{\frac{p-1}{2}} \equiv -1 \pmod p$ ***ou*** *complete t rodadas;*
6 $i \leftarrow 2 \cdot \frac{p-1}{4}$;
7 $k \leftarrow 0$;
8 **repita**
9 $i \leftarrow \frac{i}{2}$;
10 $k \leftarrow \frac{k}{2}$;
11 **se** $a^i b^k \equiv -1 \pmod p$ **então** $k \leftarrow k + 2 \cdot \frac{p-1}{4}$;
12 **até que** *i seja ímpar*
13 **responda** $a^{\frac{i+1}{2}} b^{\frac{k}{2}} \mod p$.

Algoritmo 28: raiz quadrada no $\mathbb{Z}_p^*$.

repita, linha 3, termina por causa do número de rodadas t. Prove que

$$\mathbb{P}[\text{erro}] \leqslant \left(\frac{1}{2}\right)^t.$$

EXERCÍCIO 2.80. Vimos (proposição 2.46) que $x^2 \equiv 1 \pmod p$ tem exatamente duas soluções. Prove que $x^2 \equiv a \pmod p$ tem zero ou duas soluções. Prove que se n é um inteiro composto e ímpar com $k > 1$ fotores primos distintos então o número de soluções de $x^2 \equiv a \pmod n$ é 0 ou 2^k (*dica*: teorema chinês do resto).

EXERCÍCIO 2.81 (teste de primalidade de Micali). O teste de primalidade de Micali, algoritmo 29, testa primalidade de n e pode errar nas duas respostas. A ideia é sortear um quadrado a^2 do $\mathbb{Z}_p$ e extrair a raiz quadrada com o algoritmo Monte Carlo acima, algoritmo 28. Se n é primo, então as únicas raízes são a e $-a$; senão o algoritmo 28 computa uma raiz que pode ser diferente dessas duas.

Instância : inteiros positivos $n \geqslant 3$ e t.
Resposta : se n é primo ou composto.

1 **se** *n é par ou potência de primo* **então responda** *composto*.
2 $a \leftarrow_R \{2, 3, \ldots, n-1\}$;
3 **se** $\text{mdc}(a, n) \neq 1$ **então responda** *composto*.
4 $x \leftarrow a^2 \mod n$;
5 $y \leftarrow$ raiz quadrada determinada pelo algoritmo 28 com parâmetros (x, p, t);
6 **se** $y \neq \{-a, a\}$ *ou* $y^2 \neq x \mod n$ *ou a linha 3 do algoritmo 28 terminou por t* **então**
7 **responda** *composto*.
8 **senão responda** *primo*.

Algoritmo 29: teste de primalidade de Micali.

Determine o tempo de execução desse algoritmo. Prove que se n é primo então $\mathbb{P}[\text{erro}] \leqslant 2^{-t}$ e que se n é composto então $\mathbb{P}[\text{erro}] < 1/2$ (nesse caso o exercício 2.80 pode ser útil).

Bibliografia

Agrawal, Manindra e Somenath Biswas (2003). "Primality and identity testing via Chinese remaindering". Em: *J. ACM* 50.4, 429–443 (electronic) (ver pp. 75, 76).

Agrawal, Manindra, Neeraj Kayal e Nitin Saxena (2004). "PRIMES is in P". Em: *Ann. of Math. (2)* 160.2, pp. 781–793 (ver pp. 4, 67).

Aleliunas, Romas et al. (1979). "Random walks, universal traversal sequences, and the complexity of maze problems". Em: *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*. SFCS '79. Washington, DC, USA: IEEE Computer Society, pp. 218–223. DOI: http://dx.doi.org/10.1109/SFCS.1979.34 (ver p. 156).

Alexander, K. S., K. Baclawski e G. C. Rota (1993). "A stochastic interpretation of the Riemann zeta function". Em: *Proceedings of the National Academy of Sciences of the United States of America* 2.90, 697–699 (ver p. 105).

Arvind, V. e Partha Mukhopadhyay (2008). "Derandomizing the Isolation Lemma and Lower Bounds for Circuit Size". Em: *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*. Ed. por Ashish Goel et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 276–289 (ver p. 86).

Aspvall, Bengt, Michael F. Plass e Robert Endre Tarjan (1979). "A linear-time algorithm for testing the truth of certain quantified Boolean formulas". Em: *Inform. Process. Lett.* 8.3, pp. 121–123 (ver p. 144).

Bach, Eric e Jeffrey Shallit (1996). *Algorithmic Number Theory*. Cambridge, MA, USA: MIT Press (ver pp. 61, 62).

Bartle, Robert G. (1976). *The elements of real analysis*. Second. John Wiley & Sons, New York-London-Sydney, pp. xv+480 (ver p. 187).

Bernstein, Daniel J. (1998). "Detecting perfect powers in essentially linear time". Em: *Math. Comput.* 67.223, pp. 1253–1283. DOI: http://dx.doi.org/10.1090/S0025-5718-98-00952-1 (ver p. 76).

Billingsley, P. (1979). *Probability and measure*. Wiley series in probability and mathematical statistics. Probability and mathematical statistics. Wiley (ver p. 81).

Cormen, Thomas H., Charles E. Leiserson e Ronald L. Rivest (1990). *Introduction to algorithms*. The MIT Electrical Engineering and Computer Science Series. Cambridge, MA: MIT Press, pp. xx+1028 (ver pp. 108, 110).

DeMillo, Richard A. e Richard J. Lipton (1978). "A Probabilistic Remark on Algebraic Program Testing". Em: *Inf. Process. Lett.* 7.4, pp. 193–195 (ver p. 57).

Diffie, Whitfield e Martin E. Hellman (1976). "New directions in cryptography". Em: *IEEE Trans. Information Theory* IT-22.6, pp. 644–654 (ver pp. 59, 137).

Erdős, P. (1956). "On pseudoprimes and Carmichael numbers". Em: *Publ. Math. Debrecen* 4, pp. 201–206 (ver p. 69).

Feller, William (1968). *An introduction to probability theory and its applications. Vol. I*. Third edition. New York: John Wiley & Sons Inc., pp. xviii+509 (ver pp. 92, 167, 182).

Freivalds, Rusins (1977). "Probabilistic Machines Can Use Less Running Time". Em: *IFIP Congress*, pp. 839–842 (ver p. 54).

Gao, Shuhong e Daniel Panario (1997). "Tests and Constructions of Irreducible Polynomials over Finite Fields". Em: *Foundations of Computational Mathematics*. Ed. por Felipe Cucker e Michael Shub. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 346–361 (ver p. 66).

Garfinkel, Simson (1994). *PGP: Pretty Good Privacy*. O'Reilly (ver p. 70).

Gathen, Joachim von zur e Jürgen Gerhard (2013). *Modern Computer Algebra*. 3ª ed. Cambridge University Press. DOI: 10.1017/CBO9781139856065 (ver p. 63).

Gelbaum, B.R. e J.M.H. Olmsted (1964). *Counterexamples in Analysis*. Dover books on mathematics. Holden-Day (ver p. 11).

Golub, Gene H. e Charles F. Van Loan (1989). *Matrix computations*. Second. Vol. 3. Johns Hopkins Series in the Mathematical Sciences. Baltimore, MD: Johns Hopkins University Press, pp. xxii+642 (ver p. 171).

Graham, Paul (2002). *A Plan for Spam*. http://www.paulgraham.com/spam.html. Acesso em 06/04/2009 (ver p. 26).

Graham, Ronald L., Donald E. Knuth e Oren Patashnik (1994). *Concrete mathematics*. Second. A foundation for computer science. Reading, MA: Addison-Wesley Publishing Company, pp. xiv+657 (ver p. 78).

Harman, Glyn (2005). "On the Number of Carmichael Numbers up to x". Em: *Bulletin of the London Mathematical Society* 37.5, pp. 641–650. DOI: 10.1112/S0024609305004686. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/S0024609305004686 (ver p. 69).

Harvey, David e Joris Van Der Hoeven (2019). "Integer multiplication in time $O(n \log n)$". working paper or preprint (ver p. 47).

Håstad, Johan (2001). "Some Optimal Inapproximability Results". Em: *J. ACM* 48.4, pp. 798–859. DOI: 10.1145/502090.502098 (ver pp. 114, 115).

Haveliwala, Taher e Sepandar Kamvar (2003). *The Second Eigenvalue of the Google Matrix*. Rel. téc. 20. Stanford University (ver p. 171).

Ireland, Kenneth e Michael Rosen (1990). *A classical introduction to modern number theory*. Second. Vol. 84. Graduate Texts in Mathematics. New York: Springer-Verlag, pp. xiv+389 (ver p. 137).

Johnson, David S. (1974). "Approximation algorithms for combinatorial problems". Em: *J. Comput. System Sci.* 9. Fifth Annual ACM Symposium on the Theory of Computing (Austin, Tex., 1973), pp. 256–278 (ver p. 112).

Karger, David R. (1993). "Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm". Em: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (Austin, TX, 1993)*. New York: ACM, pp. 21–30 (ver p. 52).

Kimbrel, Tracy e Rakesh Kumar Sinha (1993). "A probabilistic algorithm for verifying matrix products using $O(n^2)$ time and $\log_2 n + O(1)$ random bits". Em: *Inform. Process. Lett.* 45.2, pp. 107–110. DOI: 10.1016/0020-0190(93)90224-W (ver p. 55).

Klivans, Adam R. e Daniel A. Spielman (2001). "Randomness efficient identity testing of multivariate polynomials". Em: *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pp. 216–223. DOI: 10.1145/380752.380801 (ver p. 86).

Knuth, Donald E. (1981). *The art of computer programming. Vol. 2*. Second. Seminumerical algorithms, Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Co., Reading, Mass., pp. xiii+688 (ver p. 54).

Lehmann, Daniel e Michael O. Rabin (1981). "On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem". Em: *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Williamsburg, Virginia: ACM, pp. 133–138. DOI: http://doi.acm.org/10.1145/567532.567547 (ver p. 80).

Lidl, Rudolf e Harald Niederreiter (1997). *Finite fields*. Second. Vol. 20. Encyclopedia of Mathematics and its Applications. With a foreword by P. M. Cohn. Cambridge: Cambridge University Press, pp. xiv+755 (ver pp. 63, 65, 66).

Maltese, George (1986). "A Simple Proof of the Fundamental Theorem of Finite Markov Chains". Em: *The American Mathematical Monthly* 93.8, pp. 629–630 (ver p. 148).

Menezes, Alfred J., Paul C. van Oorschot e Scott A. Vanstone (1997). *Handbook of applied cryptography*. CRC Press Series on Discrete Mathematics and its Applications. With a foreword by Ronald L. Rivest. Boca Raton, FL: CRC Press, pp. xxviii+780 (ver p. 80).

Miller, Gary L. (1975). "Riemann's hypothesis and tests for primality". Em: *Seventh Annual ACM Symposium on Theory of Computing (Albuquerque, N.M., 1975)*. Assoc. Comput. Mach., New York, pp. 234–239 (ver p. 71).

Mitzenmacher, Michael e Eli Upfal (2005). *Probability and computing*. Randomized algorithms and probabilistic analysis. Cambridge: Cambridge University Press, pp. xvi+352 (ver p. 55).

Monier, Louis (1980). "Evaluation and comparison of two efficient probabilistic primality testing algorithms". Em: *Theoretical Computer Science* 12.1, pp. 97 –108. DOI: https://doi.org/10.1016/0304-3975(80)90007-9 (ver p. 67).

Mulmuley, Ketan, Umesh V. Vazirani e Vijay V. Vazirani (1987). "Matching is as easy as matrix inversion". Em: *Combinatorica* 7.1, pp. 105–113 (ver p. 37).

Nightingale, Edmund B., John R. Douceur e Vince Orgovan (2011). "Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs". Em: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, pp. 343–356. DOI: 10.1145/1966445.1966477 (ver p. 5).

O'Gorman, T. J. et al. (1996). "Field Testing for Cosmic Ray Soft Errors in Semiconductor Memories". Em: *IBM J. Res. Dev.* 40.1, pp. 41–50. DOI: 10.1147/rd.401.0041 (ver p. 5).

Page, Lawrence et al. (1998). *The PageRank Citation Ranking: Bringing Order to the Web*. Rel. téc. Stanford Digital Library Technologies Project (ver pp. 169, 171).

Prasolov, V. V. (2006). *Elements of Combinatorial and Differential Topology*. Graduate Studies in Mathematics 74. American Mathematical Society (ver p. 148).

Pugh, William (1989). "Skip lists: a probabilistic alternative to balanced trees". Em: *Algorithms and data structures (Ottawa, ON, 1989)*. Vol. 382. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 437–449 (ver p. 124).

Raab, Martin e Angelika Steger (1998). ""Balls into Bins- A Simple and Tight Analysis". Em: *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*. RANDOM '98. Berlin, Heidelberg: Springer-Verlag, pp. 159–170 (ver p. 132).

Rabin, Michael O. (1980a). "Probabilistic algorithm for testing primality". Em: *J. Number Theory* 12.1, pp. 128–138 (ver p. 71).

— (1980b). "Probabilistic algorithms in finite fields". Em: *SIAM J. Comput.* 9.2, pp. 273–280 (ver p. 66).

Reingold, Omer (2008). "Undirected connectivity in log-space". Em: *J. ACM* 55.4, Art. 17, 24 (ver p. 156).

Ribenboim, Paulo (1996). *The new book of prime number records*. New York: Springer-Verlag, pp. xxiv+541 (ver p. 79).

Rosenthal, Jeffrey S. (2006). *A first look at rigorous probability theory*. Second. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, pp. xvi+219 (ver p. 11).

Ross, Sheldon M. (2010). *A first course in Probability*. 8th. New Jersey: Prentice Hall (ver pp. 23, 161).

Saldanha, Nicolau (1997). "Precisa-se de alguém para ganhar muito dinheiro". Disponível em http://www.mat.puc-rio.br/~nicolau/publ/papers/otario.pdf. Acesso em 07/2018 (ver p. 34).

Schroeder, Bianca, Eduardo Pinheiro e Wolf-Dietrich Weber (2009). "DRAM Errors in the Wild: A Large-scale Field Study". Em: *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '09. Seattle, WA, USA: ACM, pp. 193–204. DOI: 10.1145/1555349.1555372 (ver p. 5).

Schwartz, Jacob T. (1979). "Probabilistic algorithms for verification of polynomial identities". Em: *Symbolic and algebraic computation (EUROSAM '79, Internat. Sympos., Marseille, 1979)*. Vol. 72. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 200–215 (ver p. 57).

Shpilka, Amir e Amir Yehudayoff (2010). "Arithmetic Circuits: A Survey of Recent Results and Open Questions". Em: *Foundations and Trends® in Theoretical Computer Science* 5.3–4, pp. 207–388. DOI: 10.1561/0400000039 (ver p. 57).

We knew the web was big... (2008). http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html (ver p. 172).

Wills, Rebecca S. (2006). "Google's PageRank: the math behind the search engine". Em: *Math. Intelligencer* 28.4, pp. 6–11 (ver p. 172).

Zippel, Richard (1979). "Probabilistic algorithms for sparse polynomials". Em: *Symbolic and algebraic computation (EUROSAM '79, Internat. Sympos., Marseille, 1979)*. Vol. 72. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 216–226 (ver p. 57).