A EFICIÊNCIA DOS ALGORITMOS NUMÉRICOS

FERNANDO FERREIRA

A forma como hoje nos habituámos a viver assenta em resultados da teoria dos números. Com efeito, alguns protocolos criptográficos importantes que estão na base da transmissão segura de informação têm a sua justificação última em resultados (relativamente elementares) da teoria dos números. O comércio pela *internet*, hoje tão ubíquo, assim como as operações bancárias que se fazem eletronicamente, dependem destes protocolos. No âmago da constituição e segurança destes protocolos estão questões de eficiência computacional. Por um lado, para os pôr de pé, é fulcral que certas operações numéricas se possam executar rapidamente (dada a tecnologia atual) enquanto que, para assegurar a segurança criptográfica, é mister que certas outras operações numéricas não possam ser efetuadas eficientemente. As operações numéricas de que falamos – e de que iremos dar exemplos – efetuam-se através de algoritmos (vertidos para linguagens de programação e executados em computadores, tábletes, telefones inteligentes, etc.). Se bem que o estudo de questões de computabilidade e complexidade computacional seja, por si só, objeto de disciplinas matematizadas (a noção seminal de *máquina de Turing* desempenha nestes estudos teóricos um papel fundamentador imprescindível), nesta secção vamos adoptar uma postura informal – mas suficientemente rigorosa – para as aplicações que nos interessam.

Suponhamos que nos é dado um certo número natural composto n e que nos pedem para encontrar um seu fator próprio. Podemos tentar fazer uma busca exaustiva (força bruta). Primeiro vemos se $2 \mid n$. Se sim, encontrámos um fator. Se não, vemos se $3 \mid n$. Se sim, já está. Se não, vemos se $4 \mid n$, depois se $5 \mid n$, e por aí fora até (no máximo) atingirmos $\sqrt{n}$. Este algoritmo funciona (mais tarde ou mais cedo encontramos um fator próprio), mas é extremamente ineficiente. Suponhamos que n tem comprimento 128 em notação posicional binária (trata-se de um número com cerca de 43 dígitos). Se trabalharmos com um computador que efetue mil milhões de divisões por segundo para números desta grandeza, ao fim de um ano far-se-iam $3,2 \times 10^{16}$ divisões. Como temos que fazer potencialmente 2^{64} divisões (aproximadamente $1,8 \times 10^{19}$ divisões), demoraríamos à volta de 563 anos a fazê-lo. Para n de comprimento binário 254 (cerca de 76 dígitos), 2^{128} é cerca de $3,4 \times 10^{38}$. O nosso sistema solar tem, aproximadamente, 6×10^9 anos. Consequentemente, todo este tempo está ainda muitíssimo longe de chegar para executar 2^{128} divisões.

Estes números estarrecedores surgem porque os algoritmos de força bruta têm de verificar um *número exponencial* de casos (por exemplo, 2^ℓ casos para entradas de comprimento ℓ). No primeiro dos exemplos acima, a entrada é um número n de comprimento binário $\ell = 128$ (n é *representado* por uma sequência finita de 128 zeros e uns) e o número de casos a verificar é 2^{64} (que tem crescimento exponencial em função do *comprimento* binário do número, ou melhor, da metade do comprimento de n quando este está representado em notação binária). Deve observar-se que em qualquer processo algorítmico, os dados vêm sempre sob uma representação, em geral dada por uma sequência finita de símbolos. Nos algoritmos numéricos, trata-se comummente da representação posicional numa certa base (para questões de eficiência algorítmica, a base concreta – sendo usualmente a binária – não é importante, porque os comprimentos das representações em bases distintas diferem apenas por uma constante multiplicativa e também porque a mudança de base é uma operação eficiente). É claro que, se queremos calcular eficientemente, devemos evitar um número exponencial de passos computacionais.

Os algoritmos que aprendemos em criança para somar, subtrair, multiplicar e dividir números naturais são eficientes. De facto, os dois primeiros trabalham em tempo linear e os dois últimos em tempo quadrático (por exemplo, o número de passos computacionais básicos necessários para efetuar um produto é majorado por uma função quadrática no comprimento do maior dos números a multiplicar). Isto é muito intuitivo, se o leitor pensar um pouco no assunto. Estes algoritmos trabalham em *tempo polinomial*. A partir da eficiência do cálculo das quatro operações aritméticas básicas, conclui-se imediatamente que a aritmética modular da soma e do produto também se faz eficientemente. Por exemplo, se queremos multiplicar dois números inteiros a e b módulo n (onde $0 \leqslant a, b < n$), basta primeiro calcular $a \cdot b$ e depois efetuar a divisão deste produto por n de modo a obter o resto da divisão. Trata-se, portanto, dum cálculo que é executado em tempo quadrático (no comprimento de n).

O algoritmo de Euclides para calcular o máximo divisor comum de dois números naturais n e m ($n > m$) é, também ele, muito eficiente. Com efeito, este algoritmo consiste numa série de divisões e o número de divisões a efetuar é majorado pelo dobro do comprimento de n. Esta estimativa resulta do facto de que, de duas em duas divisões, o resto da divisão diminui para, pelo menos, metade (ver um exercício). Como cada divisão tem um custo quadrático e o número de divisões a fazer é linear no comprimento de n, o algoritmo de Euclides trabalha em tempo cúbico. Considere-se agora a seguinte questão relacionada. Dados números naturais n e a, com $0 < a \prec n$ e $a \perp n$, sabemos que existe o inverso modular de a módulo n. Dito de outro modo, existe (e é único) um número natural b, com $0 < b < n$, tal que $ab \equiv 1 \pmod{n}$. Qual é o custo de calcular b nestas circunstâncias? Ora, podemos calcular b usando o algoritmo *estendido* de Euclides. Este algoritmo baseia-se no algoritmo (simples) de Euclides para calcular o máximo divisor comum entre n e a mas, ao longo das sucessivas divisões, vai também calculando coeficientes inteiros x e y tais que $xn + ya = r$, onde r é um dos sucessivos restos dado pelo algoritmo (simples) de Euclides. O cálculo destes sucessivos coeficientes obtém-se a partir dos dois anteriores coeficientes através dum número fixo de operações aritméticas (cada coeficiente obtém-se de coeficientes anteriores e de dados obtidos pelo algoritmo simples de Euclides por meio de um produto e de uma subtração). Finalmente, no último passo, quando $r = 1$, obtêm-se coeficientes inteiros x e y tais que $xn + ya = 1$. O inverso modular de a módulo n é dado por y. Ora, pode mostrar-se que estes sucessivos coeficientes não excedem n em valor absoluto (ver o segundo trabalho extra). Conclui-se, pois, que este modo de calcular o inverso modular trabalha em tempo cúbico. Trata-se, em boa verdade, duma maneira muito eficiente de calcular inversos modulares.

Concluímos esta secção com a discussão do cálculo da exponenciação em módulo: dados números naturais a, n e k com $0 < a < n$, pretende-se calcular $a^k \pmod{n}$. A exponenciação de números naturais é, claro está, uma operação que exige muito tempo. Há duas razões para isso. Em primeiro lugar, os números ficam muito grandes, muito depressa. A segunda razão é porque obter a^k iterativamente, calculando sucessivamente a^2, a^3, $\ldots$, a^k, exige efetuar $k - 1$ produtos. Ora, se ℓ for o comprimento de k em notação posicional binária (note-se que ℓ difere por uma unidade de $\log_2 k$), então o número de tais produtos é exponencial em ℓ. No cálculo da exponenciação modular é muito fácil evitar que os números cresçam, pois basta reduzir módulo n cada vez que se efetua um produto. Adicionalmente, quando estudámos o método da repetição do quadrado, vimos também que o número de produtos necessários ao cálculo da exponenciação é linear em ℓ. Basta calcular sucessivamente a^2, a^4, a^8, $\ldots$, a^{2^i}, $\ldots$ (sempre reduzindo módulo n) até que 2^{i+1} ultrapasse k (o que acontece em menos de ℓ passos). Cada entrada sucessiva é o quadrado modular da sua antecessora. Por último, tem-se ainda que se calcular um produtório (com menos de ℓ fatores) cuja forma exata depende das posições dos zeros e uns na representação de k em notação posicional binária. Em suma, o método da repetição do quadrado para calcular exponenciação modular trabalha em tempo cúbico. É, também, muito eficiente.