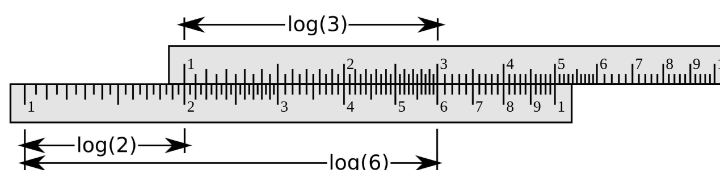
Ten artykuł dedykujemy Johnowi Napierowi (1550–1617) z okazji 400-lecia odkrycia logarytmu ogłoszonego w książce *Mirifici Logarithmorum Canonis Descriptio*

Myśl logarytmicznie!

*Maciej M. SYSŁO**, *Anna Beata KWIATKOWSKA**

W tym artykule ilustrujemy potęgę logarytmów w projektowaniu efektywnych algorytmów i obliczeń. Myślenie, w tle którego stoi logarytm, ukryty lub widoczny, nazwaliśmy *myśleniem logarytmicznym*. Stanowi ono jedną z podstawowych kompetencji niezbędnych przy efektywnym rozwiązywaniu rzeczywistych problemów informatycznych. Pokazujemy również – co może być ciekawe dla nauczycieli matematyki – jak wprowadzić pojęcie logarytmu, nie odwołując się do matematycznego formalizmu, a posługując się koncepcyjnym modelem redukcji rozmiaru problemu w każdym (lub w co drugim) kroku co najmniej o połowę. Może Cię zdziwić, że ta idea prowadząca do logarytmu występuje w algorytmie Euklidesa, który został opisany niemal 2000 lat przed wynalezieniem logarytmu przez Napiera.

Krótko po wynalezieniu logarytmu Edmund Gunter w 1620 roku utworzył skalę logarytmiczną, a z połączenia dwóch takich skal William Oughtred w 1622 roku zbudował *suwak logarytmiczny*. Oryginalnie, suwaki służyły do wykonywania mnożenia (patrz rys. 1) i dzielenia, inne skale na suwakach służą do podnoszenia do kwadratu



Rys. 1. Sposób obliczania iloczynu $2 \cdot 3 = 6$ za pomocą suwaka logarytmicznego.

i do wyciągania pierwiastków kwadratowych, do obliczania trzecich potęg i pierwiastków trzeciego stopnia, do obliczania wartości funkcji trygonometrycznych i do wielu innych obliczeń. Jeden z najbogatszych suwaków, Faber Castell 2/83N, zawiera aż 21 skal. Budowano suwaki podłużne, okrągłe i ze skalami nawiniętymi na cylindry, by móc wydłużyć ich długości dla osiągnięcia dokładniejszych wyników – skala na cylindrycznym suwaku Fullera ma długość 12 metrów.

Dodajmy tutaj, że Napier wynalazł również tak zwane *pałeczki Napiera*, które służą do mnożenia liczb, nie mają one jednak nic wspólnego z logarytmami. Posłużyły natomiast Wilhelmowi Schickardowi do zbudowania w 1624 roku pierwszego kalkulatora mechanicznego.

Rok 1972 to początek agonii suwaków – zaczęły je wypierać stworzone z ich pomocą kalkulatory elektroniczne. Ponad 40 milionów wcześniej wyprodukowanych suwaków stało się nagle bezużytecznych i obecnie stanowi głównie eksponaty kolekcjonerskie. Dzisiaj jednak nie można wyobrazić sobie zajmowania się informatyką, bez przynajmniej „otarcia” się o logarytmy, co ilustrujemy w tym artykule.

* * *

Funkcje nie są obiektami matematycznymi lubianymi przez uczniów, zwłaszcza gdy są wprowadzane w sposób formalny jako przekształcenia (odwzorowania). Wśród nich „najgorszą sławą” cieszy się *logarytm*, bo nie dość, że jest to funkcja, to na dodatek jest to funkcja odwrotna. Jednakże do zrozumienia tego, o czym tutaj piszemy, nie będzie potrzebna znajomość pojęcia logarytmu.

W przeszłości uzasadnieniem dla posługiwania się logarytmami, były właściwości, które legły u podstaw jego wprowadzenia do obliczeń. Logarytm ułatwia wykonywanie złożonych obliczeń dzięki zastąpieniu działań multiplikatywnych (takich jak mnożenie i dzielenie) przez działania addytywne (dodawanie i odejmowanie). Nie tak dawno jeszcze, w szkołach posługiwano się tablicami logarytmicznymi, a na uczelniach i w zakładach pracy przyszli i zawodowi inżynierowie korzystali z suwaków logarytmicznych. Obecnie logarytm pełni rolę tzw. *mental tool* – narzędzia myślowego, sposobu rozumowania.

Rozważmy pięć następujących pytań:

- Ile należy przejrzeć kartek w słowniku, aby znaleźć poszukiwane hasło?
- Ile miejsca w komputerze (a dokładniej ile bitów) zajmuje liczba naturalna?
- Jak szybko można wykonywać potęgowanie dla dużych wykładników potęg?
- Ile trwa obliczanie największego wspólnego dzielnika dwóch liczb za pomocą algorytmu Euklidesa?
- Ile kroków wykonuje algorytm typu *dziel i zwyciężaj* uruchomiony na danych o n elementach?

*Uniwersytet Mikołaja Kopernika w Toruniu

Wspólną cechą odpowiedzi na te pytania jest to, że nie można ich udzielić, nie dotykając logarytmu, pośrednio lub bezpośrednio, a ponadto wyjaśnienie tych odpowiedzi służy lepszemu zrozumieniu pojęcia logarytmu i jego roli w projektowaniu efektywnych algorytmów.

Znajdź szybko hasło w słowniku, odgadnij ukrytą liczbę. Papierowa książka telefoniczna ma 1000 stron. Jak znaleźć numer telefonu do pana Skarbka, aby przeglądnąć możliwie najmniejszą liczbę stron w tej książce? Zapewne szybko wpadniesz na pomysł, że najlepszą metodą jest podział pliku stron, na których może być numer telefonu pana Skarbka, na połowę i odrzucenie tej połowy, w której na pewno nie ma informacji o panu Skarbku. Ten podział jest kontynuowany, aż pozostanie tylko jedna strona, na której może się znaleźć numer telefonu pana Skarbka. Jest to tak zwane *poszukiwanie binarne* lub *przez połowienie*.

W dwuosobowej grze w odgadywanie liczby z podanego przedziału, pomyślanej przez jedną z osób, w której druga osoba może zadawać pytania „czy pomyślana liczba jest większa czy mniejsza niż x" również możemy zastosować podobną strategię. Jeśli każde pytanie będzie dotyczyło wartości x leżącej w połowie aktualnego przedziału, w którym znajduje się poszukiwana liczba, to liczba pytań niezbędna do odgadnięcia pomyślanej liczby będzie równa liczbie połowień oryginalnego przedziału.

Przy okazji warto zwrócić uwagę na następujące kwestie:

- Bardzo ważny jest alfabetyczny porządek nazwisk w książce telefonicznej i liczb w przedziale. Jak ci się wydaje, ile stron należałoby przejrzeć w 1000-stronicowej książce telefonicznej, by znaleźć nazwisko właściciela telefonu o numerze 1234567?
- W przypadku słownika, w którym chcemy znaleźć hasło zaczynające się początkową literą alfabetu, na ogół próbujemy znaleźć poszukiwane hasło na początkowych stronach słownika. Taka metoda nazywa się *interpolacyjnym poszukiwaniem* – na ogół działa szybciej, niż binarne poszukiwanie (więcej na ten temat znajdziesz w książce M.M. Sysło, *Algorytmy*, WSiP 1997; Helion 2014).

Binarna reprezentacja liczb i rozmiar liczby w komputerze. Zapewne wiesz, jak otrzymać *binarną reprezentację* liczby naturalnej n. Taka reprezentacja jest generowana w trakcie podziału liczby n i otrzymywanych ilorazów przez 2. Rozpoczynamy od podzielenia n przez 2 i resztę z tego dzielenia r (0 lub 1) przyjmujemy za najmniej znaczący bit w reprezentacji. Następnie powtarzamy tę procedurę dla ilorazu q tak długo, jak długo iloraz q jest większy od 0. Na przykład, dla $n = 23$ otrzymujemy 10111_2 jak w tabeli 2.

n	q	r
23	11	1
11	5	1
5	2	1
2	1	0
1	0	1

Tabela 2: Tworzenie reprezentacji binarnej

Czy zastanawiałeś się, z ilu bitów składa się binarna reprezentacja dziesiętnej liczby naturalnej n, lub inaczej, jak dużą pamięć w komputerze zajmuje liczba n?
By odpowiedzieć na to pytanie, załóżmy, że n zajmuje k bitów i określmy, jaka jest najmniejsza i największa liczba zajmująca dokładnie k bitów. Największa taka liczba składa się z k bitów równych 1, czyli

$$(111\ldots1)_2 = 2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2^1 + 2^0 = 2^k - 1.$$

Z drugiej strony, najmniejsza liczba reprezentowana dokładnie na k bitach ma tylko jedną jedynkę na najbardziej znaczącej pozycji. Stąd otrzymujemy następujące nierówności:

$$2^{k-1} - 1 < n \leqslant 2^k - 1.$$

Dodajmy 1 do wszystkich stron tych nierówności i weźmy logarytm $\log_2$. Otrzymamy:

$$k - 1 < \log_2(n+1) \leqslant k.$$

Ponieważ liczba bitów k jest liczbą naturalną (całkowitą dodatnią), mamy:

$$k = \lceil \log_2(n+1) \rceil,$$

gdzie $\lceil x \rceil$ jest sufitem liczby x, czyli równa się najmniejszej liczbie naturalnej większej lub równej x. Z tej części rozważań możemy wyciągnąć wniosek, że naturalna liczba n zajmuje w pamięci komputera około $\log_2 n$ bitów – ta liczba jest często przyjmowana za *komputerowy rozmiar liczby* n. Zauważmy, że poszukiwanie binarne w przedziale zawierającym n liczb odpowiada tworzeniu binarnej reprezentacji liczby n, liczba kroków w takim poszukiwaniu wynosi około $\log_2 n$.

Możemy teraz algorytmicznie zdefiniować $\log_2 n$ jako:

Logarytm $\log_2 n$ *jest równy liczbie kroków, jakie potrzebujemy by zmniejszyć* n *do* 1, *dzieląc sukcesywnie przez* 2.

n	$\lceil \log_2 n \rceil$
128	7
1024	10
65 536	16
1 048 576	20
10^{10}	34
10^{50}	167
10^{100}	333
10^{200}	665
10^{300}	997
10^{500}	1661

Tabela 3: Liniowy i logarytmiczny wzrost wartości

O znaczeniu i „potędze" logarytmów i funkcji logarytmicznej w informatyce, a ogólniej – w obliczeniach decyduje szybkość wzrostu jej wartości, nieporównywalnie mała względem szybkości wzrostu jej argumentu, co ilustrujemy w tabeli 3. A zatem dla liczb, które mają około stu cyfr, wartość logarytmu wynosi tylko około 333.

Szybkie podnoszenie do potęgi. Podnoszenie do potęgi jest bardzo prostym, szkolnym zadaniem. Na przykład, aby obliczyć 3^4, wykonujemy trzy mnożenia $3 \cdot 3 \cdot 3 \cdot 3$. A zatem w ogólności, aby w ten sposób obliczyć wartość potęgi x^n, należy wykonać $n-1$ mnożeń: o jedno mniej niż wynosi wykładnik potęgi. Czy ten „szkolny" algorytm jest na tyle szybki, by obliczyć na przykład wartość potęgi:

$$x^{12345678912345678912345678912345},$$

która może pojawić się przy szyfrowaniu metodą RSA informacji przesyłanych w Internecie?

Oszacujmy, ile czasu będzie trwało obliczanie tej potęgi, jeśli zastosujemy szkolny algorytm, czyli ile czasu zabierze wykonanie 12 345 678 912 345 678 912 345 678 912 344 mnożeń. Przypuśćmy, że dysponujemy superkomputerem, który działa z szybkością jednego petaflopa, zatem wykonuje około 10^{15} operacji na sekundę. Obliczenie powyższej potęgi będzie trwało

$$12\,345\,678\,912\,345\,678\,912\,345\,678\,912\,344/10^{15}\ \text{s} \approx 391\,478\,910\ \text{lat} \approx 4 \cdot 10^8\ \text{lat}.$$

Jeśli taki algorytm byłby stosowany do szyfrowania naszej poczty w Internecie, to nigdy nie otrzymalibyśmy żadnego listu. Dodajmy, że w praktycznych sytuacjach konieczne jest obliczanie potęg o wykładnikach, które mają kilkaset cyfr.

Postaramy się teraz tak wykonywać potęgowanie, aby w każdym kroku wykładnik zmniejszył się około o połowę. W tym celu zauważmy, że jeżeli n jest liczba parzystą, czyli na przykład $n = 2k$, to $x^{2k} = (x^k)^2$, a gdy n jest liczbą nieparzystą, czyli $n = 2k+1$, to $x^{2k+1} = (x^{2k}) \cdot x$. Na przykład, by obliczyć wartość x^{23}, postępujemy następująco:

$$\begin{aligned} x^{23} &= (x^{22}) \cdot x = ((x^{11})^2) \cdot x = (((x^{10}) \cdot x)^2) \cdot x = ((((x^5)^2) \cdot x)^2) \cdot x = \\ &= (((((x^4) \cdot x)^2) \cdot x)^2) \cdot x = ((((((x^2)^2) \cdot x)^2) \cdot x)^2) \cdot x. \end{aligned}$$

Zatem wartość potęgi x^{23} może być obliczona przy użyciu 7 mnożeń (podnoszenie do kwadratu to jedno mnożenie).

Oszacujmy, ile mnożeń jest wykonywanych w takim algorytmie dla dowolnego n. W tym celu porównajmy binarną reprezentację liczby $n = 23 = 10111_2$ z kolejnością wykonywania mnożeń w tym algorytmie, patrząc od prawej do lewej. Nietrudno się przekonać, że z wyjątkiem najbardziej znaczącej pozycji w reprezentacji binarnej wykładnika n, każdemu bitowi o wartości 1 odpowiada mnożenie przez x, a każdej pozycji odpowiada podniesienie do kwadratu. A zatem, liczba mnożeń potrzebnych do obliczenia wartości x^n za pomocą powyższego algorytmu jest równa liczbie pozycji w binarnej reprezentacji liczby n minus 1 plus liczba bitów równych 1 w tej reprezentacji także minus 1. Ponieważ, jak wiemy, długość reprezentacji binarnej liczby n wynosi około $\log_2 n$, liczba mnożeń potrzebnych do obliczenia wartości x^n wynosi około $2 \log_2 n$.

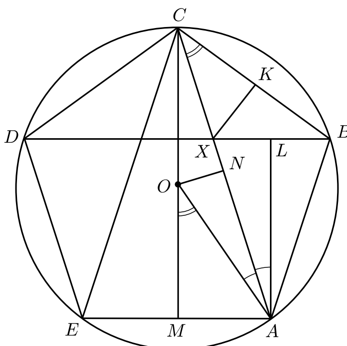
Sprawdźmy, jaki to da efekt, gdy $n = 12\,345\,678\,912\,345\,678\,912\,345\,678\,912\,345$. W tym przypadku $2 \log_2 n < 207$. Zatem zamiast czekać $4 \cdot 10^8$ lat, wynik możemy otrzymać wykonując nie więcej niż 207 mnożeń! To szokujące osiągnięcie naszego algorytmu! Na podstawie tabeli 3 widać, że obliczenie wartości x^n dla n z setkami cyfr wymaga wykonania kilka tysięcy mnożeń, co na zwykłym komputerze trwa ułamek sekundy.

Przedstawiony algorytm można zapisać w postać rekurencyjnej:

$$x^n = \begin{cases} 1 & \text{dla } n = 0, \\ (x^{n/2})^2 & \text{dla parzystego } n, \\ (x^{n-1}) \cdot x & \text{dla nieparzystego } n. \end{cases}$$

Algorytmy potęgowania są dobitną ilustracją słów Ralpha Gomory'ego, naukowego szefa firmy IBM: *Najlepszym sposobem przyspieszania pracy komputerów jest obarczanie ich mniejszą liczba działań.* Czyli prawdziwe przyspieszanie obliczeń osiągamy dzięki efektywnym algorytmom, a nie szybszym komputerom, a w tym konkretnym przypadku, dzięki zastąpieniu algorytmu liniowego przez algorytm o złożoności logarytmicznej.

Rozwiązanie zadania M 1441. Przyjmijmy oznaczenia jak na rysunku (K, L, M, N to środki odcinków).



Niech a oznacza długość boku pięciokąta oraz d – długość przekątnej. Z podobieństwa trójkątów ABC i BXC mamy $a/d = (d-a)/a$, skąd $ad = d^2 - a^2$. Z podobieństwa trójkątów ANO i ALX otrzymujemy

$$(*) \qquad \frac{\sqrt{1-(d/2)^2}}{1} = \frac{(d-a)/2}{a},$$

skąd $a^2(4-d^2) = (d-a)^2 = d^2 + a^2 - 2ad = d^2 + a^2 - 2(d^2 - a^2)$. W takim razie $(ad)^2 = a^2 + d^2$. Wreszcie z podobieństwa trójkątów AMO i XKC i $(*)$ mamy

$$\frac{1}{\sqrt{1-(a/2)^2}} = \frac{d-a}{a/2} = 4\sqrt{1-(d/2)^2},$$

skąd $1 = 16 - 4a^2 - 4d^2 + a^2d^2 = 16 - 3a^2d^2$, więc $(ad)^2 = 5$.

Algorytm Euklidesa a metoda połowienia. Okazuje się, że Euklides był bliski wynalezienia logarytmu, niemal 2000 lat przed tym, jak zrobił to John Napier. Algorytm Euklidesa jest jednym z najstarszych znanych algorytmów. Służy do znajdowania największego wspólnego dzielnika (w skrócie NWD) dwóch liczb. W tabeli 4 są zamieszczone wyniki obliczeń tego algorytmu w trakcie znajdowania NWD(34, 21). W ogólności, algorytm Euklidesa generuje następujący ciąg reszt (trzecia kolumna w tabeli 4):

$$r_{-1} = n, \quad r_0 = m, \quad r_1, \quad r_2, \quad \ldots, \quad r_k = 0,$$

Te reszty są generowane zgodnie z następującym ciągiem równości:

$$\begin{aligned} r_{-1} &= q_1 r_0 + r_1, && \text{gdzie } 0 \leqslant r_1 < r_0, \\ r_0 &= q_2 r_1 + r_2, && \text{gdzie } 0 \leqslant r_2 < r_1, \\ &\vdots \\ r_{k-2} &= q_k r_{k-1} + r_k, && \text{gdzie } 0 \leqslant r_k < r_{k-1}, \end{aligned}$$

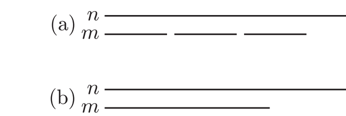
n	m	r_i
34	21	13
21	13	8
13	8	5
8	5	3
5	3	2
3	2	1
2	1	0

Tabela 4. Obliczanie wartości NWD(34, 21)

i ostatecznie $\mathrm{NWD}(n, m) = r_{k-1}$. Pierwsza równość odpowiada pierwszemu wierszowi w tabeli 4, a ostatnia – ostatniemu wierszowi. Ilorazy q_i i reszty r_i w tych równościach spełniają równości:

$$q_i = r_{i-2} \textbf{ div } r_{i-1}, \qquad r_i = r_{i-2} \textbf{ mod } r_{i-1}.$$

Powstaje teraz pytanie, ile kroków wykonuje algorytm Euklidesa, by obliczyć wartość $\mathrm{NWD}(n, m)$. Pewna sugestia może wyniknąć z porównania liczb w pierwszej i trzeciej kolumnie w tabeli 4. Można zauważyć, że w każdym wierszu liczba w trzeciej kolumnie jest co najmniej dwa razy mniejsza niż liczba w pierwszej kolumnie, a zatem reszta r_i w równaniu $r_i = r_{i-2} \textbf{ mod } r_{i-1}$, jest co najmniej dwa razy mniejsza niż r_{i-2}.



Rys. 5. Geometryczne uzasadnienie faktu, że reszta z dzielenia n przez m jest nie większa niż $n/2$.

Uzasadnienie tego faktu jest bardzo proste, jeśli posłużymy się rozumowaniem geometrycznym (rys. 5). A zatem, chcemy pokazać, że reszta r z dzielenia n przez m nie jest większa niż $n/2$. Rozważmy dwa przypadki:

(a) $m \leqslant n/2$ – ponieważ reszta nie jest większa niż m, więc wynosi co najwyżej $n/2$;
(b) $m > n/2$ – reszta wynosi $n - m$, a $n - m$ nie jest większe niż $n/2$.

A zatem, w ciągu reszt generowanym w algorytmie Euklidesa każda reszta jest co najmniej dwa razy mniejsza niż reszta, która występuje w tym ciągu o dwie pozycje wcześniej. Przypomina to ciąg liczb generowany przez algorytm poszukiwania binarnego, z wyjątkiem tego, że generowany ciąg może być w najgorszym przypadku dwa razy dłuższy. Ważnym wnioskiem jest więc stwierdzenie, że:

Algorytm Euklidesa oblicza $\mathrm{NWD}(n, m)$ *dla* $n \geqslant m$ *w co najwyżej* $2 \log_2 n$ *krokach.*

Pewnym wyzwaniem jest pytanie dla jakich liczb n i m algorytm Euklidesa wykonuje największą możliwą liczbę kroków (dwie takie liczby zostały użyte w naszym przykładzie). Odpowiedź może być zaskakująca.

Liczby Fibonacciego. Tendencja do zastępowania algorytmów o złożoności liniowej przez algorytmy o złożoności logarytmicznej, zilustrowana algorytmem szybkiego potęgowaniem, występuje w wielu innych problemach algorytmicznych, np. przy wyznaczaniu wartości liczb Fibonacciego. Klasyczna zależność rekurencyjna, definiująca liczby Fibonacciego, może być wykorzystana do podania algorytmu liniowego, który dodatkowo unika wielokrotnych, takich samych odwołań rekurencyjnych.

Aby otrzymać w tym przypadku algorytm o złożoności logarytmicznej, należy posłużyć się układem dwóch zależności rekurencyjnych, w których indeksy liczb Fibonacciego po prawej stronie są redukowane o około połowę. I ponownie, jak powyżej, by uniknąć wielokrotnych takich samych wywołań rekurencyjnych, należy rekurencję zrealizować jako iterację.

$$F_0 = 0, \quad F_1 = 1, \quad F_{2n-1} = F_{n-1}^2 + F_n^2, \quad F_{2n} = 2F_{n-1}F_n + F_n^2.$$

Więcej na ten temat można znaleźć w książce M.M. Sysło, *Piramidy, szyszki i inne konstrukcje algorytmiczne*, WSiP 1998; Helion 2015.

Metody podziału i ograniczeń. Wszystkie algorytmy zaprezentowane w tej pracy bazują na idei metody *dziel i zwyciężaj*. W informatyce istnieje bardzo wiele algorytmów będących realizacją tej idei. We wszystkich przypadkach to podejście algorytmiczne wprowadza do ogólnego wyrażenia na złożoność obliczeniową rozwiązywanego problemu jedynie logarytmiczny czynnik. Tak jest na przykład w przypadku sortowania przez binarne umieszczanie czy sortowanie przez scalanie.