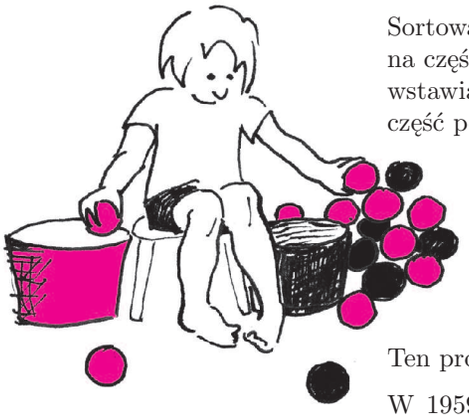
O sortowaniu Shella

Marcin CIURA[*]



Sortowanie przez proste wstawianie polega na tym, że elementy sortowane dzieli się na część posortowaną i nieposortowaną. We właściwe miejsca części posortowanej wstawia się przylegające do niej elementy części nieposortowanej. W ten sposób część posortowana rośnie od jednego do wszystkich elementów. Przykład:

$$\begin{array}{l} 8\,|\,2\ 5\ 9\ 6 \\ \mathbf{2}\ 8\,|\,5\ 9\ 6 \\ 2\ \mathbf{5}\ 8\,|\,9\ 6 \\ 2\ 5\ 8\ \mathbf{9}\,|\,6 \\ 2\ 5\ \mathbf{6}\ 8\ 9\,| \end{array}$$

Ten prosty pomysł stwarza jednak ogromne możliwości.

W 1959 roku Donald Shell opublikował pierwszy algorytm sortowania, który zarazem miał złożoność czasową potencjalnie niższą niż kwadratowa i działał w miejscu, czyli bez osobnej tablicy na wynik. Kolejne przebiegi algorytmu noszącego jego nazwisko polegają na sortowaniu przez proste wstawianie elementów oddalonych o ustaloną liczbę miejsc h, czyli na tak zwanym h-sortowaniu.

Poniżej zilustrowano przebieg sortowania metodą Shella z odstępami 5, 3, 1. Pierwszy przebieg, czyli 5-sortowanie, sortuje osobno zawartość każdego z fragmentów (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}), na przykład fragment (a_1, a_6, a_{11}) zmienia z $(62, 17, 25)$ na $(17, 25, 62)$. Następny przebieg, czyli 3-sortowanie, sortuje zawartość fragmentów (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}). Ostatni przebieg, czyli 1-sortowanie, to zwykłe sortowanie przez proste wstawianie całej tablicy $(a_1, \ldots, a_{12})$.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
dane wejściowe:	62	83	18	53	07	17	95	86	47	69	25	28
po 5-sortowaniu:	17	28	18	47	07	25	83	86	53	69	62	95
po 3-sortowaniu:	17	07	18	47	28	25	69	62	53	83	86	95
po 1-sortowaniu:	07	17	18	25	28	47	53	62	69	83	86	95

Zauważmy, że fragmenty tablicy, na których operuje algorytm Shella, są z początku krótkie, a pod koniec prawie uporządkowane. Oba te przypadki sprzyjają sortowaniu przez proste wstawianie.

Oto intrygująca własność: po h_2-sortowaniu dowolnej h_1-posortowanej tablicy pozostaje ona nadal h_1-posortowana. Każda h_1-posortowana i h_2-posortowana tablica jest też $(a_1h_1 + a_2h_2)$--posortowana dla wszystkich całkowitych nieujemnych a_1 i a_2.

Każdy ciąg odstępów zakończony jedynką prowadzi do poprawnie sortującego algorytmu, ale ich zachowanie na tyle się różni, że właściwiej byłoby mówić o *algorytmach Shella* niż o jednym algorytmie. W tabeli na następnej stronie zestawiono większość dotychczas opublikowanych propozycji ciągów odstępów. Niektóre z tych ciągów mają malejące wyrazy zależne od N, czyli rozmiaru sortowanej tablicy. Inne to rosnące ciągi nieskończone, z których należy wziąć w odwrotnej kolejności wyrazy mniejsze od N.

Jiang, Li i Vitányi podali dolne ograniczenia na rząd średniej złożoności m-przebiegowego sortowania Shella: $\Omega(mN^{1+1/m})$ przy $m \leqslant \log_2 N$ i $\Omega(mN)$ przy $m > \log_2 N$. Zatem algorytm Shella ma szanse działać w średnim czasie rosnącym jak $N \log N$ tylko z ciągami o liczbie odstępów rosnącej proporcjonalnie do logarytmu długości sortowanych tablic. Nie wiadomo jednak, czy taki optymalny rząd złożoności średniej jest w ogóle osiągalny. Wiadomo natomiast, że złożoność pesymistyczna jest wyższego rzędu: Poonen, Plaxton i Suel dowiedli, że rośnie ona co najmniej jak $\Omega(N \log^2 N/(\log \log N)^2)$.

Złożoność pesymistyczna algorytmu Shella wiąże się z *liczbami Frobeniusa*. Dla danych całkowitych $h_1, \ldots, h_n$ o największym wspólnym dzielniku 1 liczba Frobeniusa $g(h_1, \ldots, h_n)$ to największa liczba całkowita nieprzedstawialna w postaci $a_1h_1 + \ldots + a_nh_n$ przy $a_1, \ldots, a_n$ całkowitych nieujemnych. Gdy $n = 2$, $g(h_1, h_2) = (h_1 - 1)(h_2 - 1) - 1$. Wzory na $g(h_1, h_2, h_3)$ są skomplikowane. Dla $n > 3$ nie znamy ogólnych wzorów na liczby Frobeniusa, ale potrafimy je wyznaczać algorytmicznie.

Gdy wniknąć w szczegóły, luki w naszej wiedzy o algorytmie Shella okazują się tak duże, że śmiało można go nazwać najbardziej tajemniczym spośród wydajnych algorytmów sortowania. Złożoność pesymistyczną umiemy określać zaledwie dla kilku klas ciągów odstępów. O średniej liczbie operacji mamy tylko niepraktyczne wyniki: Espelid wyznaczył ją przy odstępach równych kolejnym potęgom dwójki, Knuth rozpracował sortowanie z dwoma przebiegami, Yao, a potem Janson i Knuth – sortowanie z trzema przebiegami. Na podstawie

[*]Instytut Informatyki, Politechnika Śląska

Zestawienie propozycji ciągów odstępów w algorytmie Shella

wyraz ogólny ciągu ($k \geqslant 1$)	konkretne odstępy	rząd złożoności pesymistycznej	autor i rok publikacji
$\lfloor N/2^k \rfloor$	$\lfloor \frac{N}{2} \rfloor, \lfloor \frac{N}{4} \rfloor, \ldots, 1$	$\Theta(N^2)$ [gdy $N = 2^p$]	Shell, 1959
$2\lfloor N/2^{k+1} \rfloor + 1$	$2\lfloor \frac{N}{4} \rfloor + 1, \ldots, 3, 1$	$\Theta(N^{3/2})$	Frank, Lazarus, 1960
$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta(N^{3/2})$	Hibbard, 1961
2^{k+1}, na początku 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta(N^{3/2})$	Papiernow, Stasiewicz, 1965
kolejne liczby postaci $2^p 3^q$	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta(n \log^2 N)$	Pratt, 1971
$(3^k - 1)/2$, nie większe niż $\lceil N/3 \rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta(N^{3/2})$	Knuth, 1973
$4^k + 3 \cdot 2^{k-1} + 1$, na początku 1	$1, 8, 23, 77, 281, \ldots$	$O(N^{4/3})$	Sedgewick, 1982
$\prod_{0 \leqslant q < r} a_q$ gdzie $r = \lfloor \sqrt{2k + \sqrt{2k}} \rfloor$, $q \neq ((r^2 + r)/2) - k$, $a_q = \min\{n \in \mathbb{N} : n \geqslant (5/2)^{q+1}, \forall p : 0 \leqslant p < q \Longrightarrow \mathrm{nwd}(a_p, n) = 1\}$	$1, 3, 7, 21, 48, 112, \ldots$	$O(Ne^{\sqrt{8 \ln(5/2) \ln N}})$	Incerpi, Sedgewick, 1985
$9(4^{k-1} - 2^{k-1}) + 1$, $4^{k+1} - 6 \cdot 2^k + 1$	$1, 5, 19, 41, 109$	$O(N^{4/3})$	Sedgewick, 1986
$h_k = \max\{\lfloor 5h_{k-1}/11 \rfloor, 1\}$, $h_0 = N$	$\lfloor \frac{5N}{11} \rfloor, \lfloor \frac{5}{11} \lfloor \frac{5N}{11} \rfloor \rfloor, \ldots, 1$	?	Gonnet, Baeza-Yates, 1991
$\lceil \frac{9^k - 4^k}{5 \cdot 4^{k-1}} \rceil$	$1, 4, 9, 20, 46, 103, \ldots$	?	Tokuda, 1992
nieznany	$1, 4, 10, 23, 57, 132, \ldots$	?	Ciura, 2001

doświadczeń odgadnięto jeszcze, że średni czas działania algorytmu z ciągami Hibbarda i Knutha jest rzędu $N^{5/4}$, a z ciągiem Gonneta i Baezy-Yatesa – rzędu $N(3 + \ln \ln N) \log N$, ale aproksymacje średniej liczby operacji czynione kiedyś dla innych ciągów zawodzą przy tablicach liczących miliony elementów. Zauważono też, że średnio najmniej porównań elementów potrzeba, gdy ilorazy kolejnych odstępów leżą z grubsza między 2,2 a 2,3 – dlatego ciągi Gonneta i Baezy-Yatesa o ilorazie 2,2 i Tokudy o ilorazie 2,25 sprawdzają się w praktyce – ale nie umiemy wyjaśnić, dlaczego minimum przypada właśnie w tym miejscu.

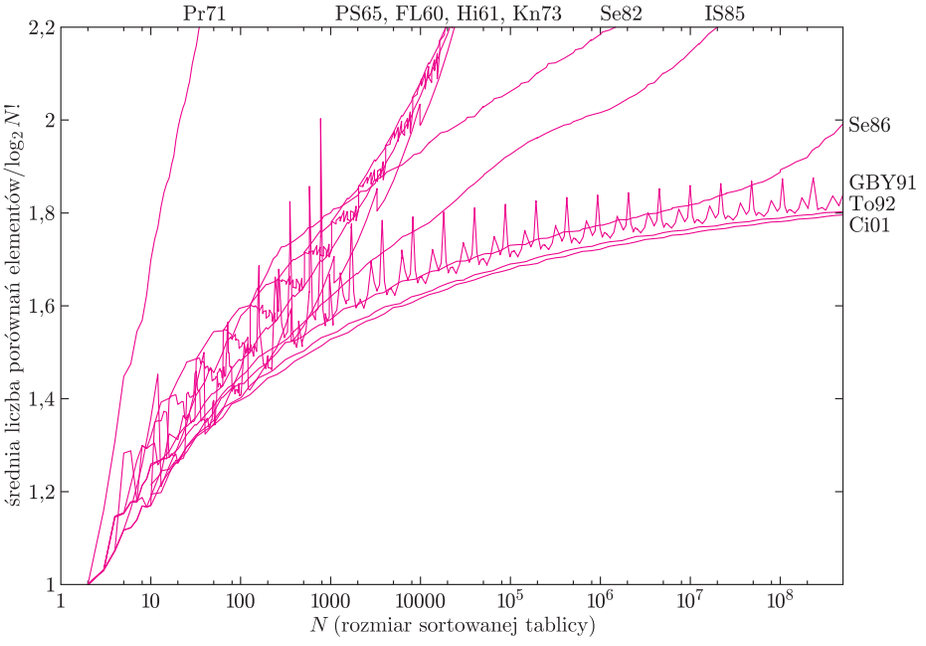
W sortowaniu szybkim można stosować algorytm Shella do krótkich podtablic, a także wtedy, gdy głębokość rekurencji przekroczy zadany limit, aby zapobiec patologicznemu spowolnieniu sortowania. Działa tak, na przykład, popularny kompresor bzip2.

Oczywiście, ciągi odstępów określone tak prostymi wzorami stanowią znikomą część wszystkich możliwości. Ze względu na liczbę prób, które trzeba by wykonać, empiryczne znalezienie po prostu najlepszego ciągu wydawało się niemożliwe nawet dla niewielkich tablic. Poszukiwanie ciągów optymalizujących średnią liczbę operacji można jednak znacznie przyspieszyć dzięki *analizie sekwencyjnej*, gdzie, inaczej niż w zwykłych metodach statystycznych, liczba prób z danym ciągiem odstępów – czyli zliczeń operacji wykonanych przy sortowaniu losowych permutacji – nie jest ustalona z góry. Wykonuje się ich tym mniej, im bardziej jednoznaczne dają wyniki. Dzięki temu można szybciej wykryć zarówno ciągi, z którymi sortowanie działa szybko, jak i te, z którymi działa wolno.

Z testów dla niedużych tablic wynika, że średnią liczbę przestawień elementów minimalizuje ciąg Pratta $\{2^p 3^q\}$, a średnią liczbę ich porównań – ciągi w przybliżeniu geometryczne. Skupimy się na porównaniach, bo to ich liczba jest proporcjonalna do czasu działania rzeczywistych programów. Od rozmiaru tablicy zależą większe odstępy najlepszych ciągów, ale nie ich początki – wszystkie dobre ciągi zaczynają się odstępami $(1, 4, 9)$, $(1, 4, 10)$, $(1, 4, 13)$ lub $(1, 5, 11)$, a optymalne wartości kolejnych odstępów ustalają się w miarę wydłużania sortowanych tablic. Dlatego poszukiwania można jeszcze skrócić przez rozpatrywanie tylko ciągów o niektórych początkach. W testach prowadzonych dla tablic liczących do kilku tysięcy elementów znaleziono wiele wydajnych ciągów o zbliżonej średniej liczbie porównań elementów, spośród których można polecić na przykład ciąg 1, 4, 10, 23, 57, 132, 301, 701.

Poniższy wykres przedstawia średnią liczbę porównań elementów w różnych wariantach sortowania Shella, dzieloną przez teoretyczne minimum, czyli $\log_2 N!$.

Aby wykonać ten wykres, do ciągu 1, 4, 10, 23, 57, 132, 301, 701 dodano wyrazy $h_k = \left\lfloor 2\frac{17}{75} h_{k-1} \right\rfloor$.



Jak widać, stosowanie tak zoptymalizowanych ciągów odstępów daje znikomy zysk, ale ciekawą i niewyjaśnioną kwestią pozostaje, dlaczego ich wyrazy przyjmują takie, a nie inne wartości oraz czy można je określić jawnym wzorem.

Łamigłówki bitowe

Jak pamiętamy, komputery operują na bitach (przyjmujących wartości 0, 1), ale zwykle nie przetwarzają ich pojedynczo, lecz większymi grupami, np. słowami 8-, 16-, 32- czy 64-bitowymi. Procesory i języki programowania, poza standardowymi operacjami arytmetycznymi na liczbach całkowitych ($+$, $-$, $\cdot$ itp.), udostępniają też operacje bitowe. Działają one tak, jakby 0 było wartością logiczną *fałsz*, a 1 – *prawda*. Na przykład *AND* (koniunkcja) działa tak:

$$0\ AND\ 0 = 0\ AND\ 1 = 1\ AND\ 0 = 0, \quad 1\ AND\ 1 = 1.$$

Z kolei *OR* (alternatywa) daje wynik 1, gdy choć jeden z argumentów jest jedynką. Negacja *NOT* zamienia zera na jedynki i odwrotnie, zaś *XOR* (od *eXclusive OR* – alternatywa wykluczająca) daje wartość 1, gdy jej argumenty są *różne*:

$$0\ XOR\ 0 = 1\ XOR\ 1 = 0, \quad 0\ XOR\ 1 = 1\ XOR\ 0 = 1.$$

Operacje te stosuje się zwykle nie do pojedynczych bitów, ale do całych słów, bit po bicie. Jeśli, na przykład, pracujemy na liczbach ośmiobitowych, to:

$$\begin{aligned} 28\ XOR\ 11 &= 00011100\ XOR\ 00001011 \\ &= 00010111 = 23. \end{aligned}$$

Te wszystkie wyjaśnienia są po to, aby Czytelnik mógł przystąpić do zadań, w których przydadzą się zarówno operacje bitowe, jak i arytmetyczne.

rozwiązania w numerze

1. n-bitowa liczba całkowita może reprezentować podzbiór zbioru n-elementowego: i-ty bit jest równy 1, gdy i-ty element należy do podzbioru, a 0 w przeciwnym razie. Jak w tej reprezentacji obliczać część wspólną i sumę dwóch podzbiorów? A dopełnienie zbioru?

2. Sprawdź, że $(a\ XOR\ b)\ XOR\ b = a$ dla dowolnych liczb a i b (to niezwykle użyteczna własność!). Kiedy $a\ XOR\ b = 0$?

3. Mamy dwie zmienne a i b, których wartości chcemy zamienić miejscami. Zwykle napisalibyśmy

$$\texttt{tmp} := \texttt{a},\ \texttt{a} := \texttt{b},\ \texttt{b} := \texttt{tmp}.$$

A jak poradzić sobie bez pomocniczej zmiennej?

4. Autor Teorii Wszystkiego zapisał swoją teorię w pliku i chce go przekazać dwóm osobom tak, aby żadna z nich nie mogła poznać ani rąbka tajemnicy, ale aby obie razem mogły odtworzyć cały plik i poznać sekret. Jak to zrobić? Wskazówka: można zacząć od wygenerowania losowego ciągu bitów o długości takiej jak cały plik.

5. Napisz pojedynczą instrukcję, która sprawdzi, czy dana liczba a jest potęgą dwójki.

6. Napisz instrukcję obliczającą maksymalną potęgę dwójki, która dzieli daną liczbę a.

7. Jak wyznaczyć liczbę elementów zbioru, reprezentowanego jak w zadaniu 1, wykonując liczbę operacji proporcjonalną do tej liczby elementów?

Zebrali M.A. i F.W.