Wydanie IV

Algorytmy

ROBERT SEDGEWICK KEVIN WAYNE

Helion

Tytuł oryginału: Algorithms (4th Edition)

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-246-3536-8

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/algor4.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/algor4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

Sortowanie to proces porządkowania obiektów w logiczny sposób. Przykładowo, na wydruku dla użytkownika karty kredytowej transakcje są uporządkowane chronologicznie. Kolejność ta została prawdopodobnie wyznaczona przez algorytm sortowania. W początkowym okresie rozwoju informatyki szacowano, że do 30% wszystkich cykli procesora poświęcanych jest na sortowanie. To, że obecnie odsetek ten jest niższy, wynika z tego, iż algorytmy sortowania są stosunkowo wydajne, a nie ze zmniejszenia znaczenia tej operacji. Wszechobecność komputerów sprawia, że dostępnych jest mnóstwo danych, a pierwszym krokiem przy ich organizowaniu jest często sortowanie. We wszystkich systemach komputerowych istnieją implementacje algorytmów sortowania dostępne dla systemu i użytkowników.

Są trzy praktyczne powody, dla których warto poznać algorytmy sortowania (mimo że można zastosować sortowanie systemowe).

- Analiza algorytmów sortowania jest solidnym wprowadzeniem do podejścia używanego przy porównywaniu wydajności algorytmów w tej książce.
- Podobne techniki są skuteczne w rozwiązywaniu innych problemów.
- Algorytmy sortowania często służą za punkt wyjścia przy rozwiązywaniu innych problemów.

Ważniejsze od tych praktycznych powodów jest to, że algorytmy sortowania są eleganckie, klasyczne i skuteczne.

Sortowanie odgrywa kluczową rolę w komercyjnym przetwarzaniu danych i współczesnych obliczeniach naukowych. Istnieje wiele zastosowań takich algorytmów w obszarze przetwarzania transakcji, optymalizacji kombinatorycznej, astrofizyki, dynamiki molekularnej, lingwistyki, badań nad genomem, prognozowania pogody itd. Jeden z algorytmów sortowania (sortowanie szybkie, opisane w PODROZDZIALE 2.3) został uznany za jeden z 10 najważniejszych algorytmów XX wieku w dziedzinie nauki i inżynierii.

W tym rozdziale omówiono kilka klasycznych metod sortowania i wydajną implementację ważnego typu danych — kolejki priorytetowej. Opisano teoretyczne podstawy porównywania algorytmów sortowania, a rozdział zakończono analizą zastosowań sortowania i kolejek priorytetowych.

2.1. PODSTAWOWE METODY SORTOWANIA

W RAMACH PIERWSZEJ WYPRAWY do krainy algorytmów sortowania analizujemy dwie podstawowe metody sortowania i odmianę jednej z nich. Oto niektóre powody do zapoznania się z tymi stosunkowo prostymi algorytmami. Po pierwsze, zapewniają one kontekst, w którym można poznać terminologię i podstawowe mechanizmy. Po drugie, te proste algorytmy są w niektórych zastosowaniach wydajniejsze od zaawansowanych algorytmów omówionych dalej. Po trzecie, jak się okaże, pozwalają poprawić wydajność bardziej skomplikowanych rozwiązań.

Reguły Zajmujemy się przede wszystkim algorytmami do zmiany kolejności w *tablicach elementów*, w których każdy element posiada *klucz*. Zadaniem algorytmu sortowania jest zmiana kolejności elementów, tak aby klucze były uporządkowane według dobrze zdefiniowanej reguły (zwykle w porządku liczbowym lub alfabetycznym). Należy uporządkować tablicę, żeby klucz każdego elementu był nie mniejszy niż klucz na każdej pozycji o niższym indeksie i nie większy niż klucz w elementach o większych indeksach. Specyficzne cechy kluczy i elementów mogą być bardzo różne w poszczególnych zastosowaniach. W Javie elementy są obiektami, a abstrakcyjne pojęcie „klucz" jest ujęte we wbudowanym mechanizmie — opisanym na stronie 259 interfejsie `Comparable`.

Klasa `Example`, przedstawiona na następnej stronie, to ilustracja zastosowanych konwencji. Kod sortujący umieszczono w metodzie `sort()` w tej samej klasie, co prywatne funkcje pomocnicze `less()` i `exch()` (a czasem także kilka innych) oraz przykładowego klienta `main()`. W klasie `Example` znajduje się też kod, który może być przydatny przy wstępnym diagnozowaniu. Klient testowy `main()` sortuje łańcuchy znaków ze standardowego wejścia i używa prywatnej metody `show()` do wyświetlenia zawartości tablicy. W dalszej części rozdziału zbadano różne klienty testowe, służące do porównywania algorytmów i analizowania ich wydajności. Aby rozróżnić metody sortowania, różnym klasom nadano inne nazwy. W klientach można wywoływać różne implementacje za pomocą specyficznych nazw: `Insertion.sort()`, `Merge.sort()`, `Quick.sort()` itd.

Kod sortujący przeważnie korzysta z danych za pomocą tylko dwóch operacji: metody `less()`, która porównuje elementy, oraz metody `exch()`, zamieniającej je miejscami. Implementowanie metody `exch()` jest łatwe, a interfejs `Comparable` ułatwia implementowanie metody `less()`. Ponieważ dostęp do danych mają tylko te dwie operacje, kod jest czytelny i przenośny, a ponadto łatwo jest sprawdzać poprawność algorytmów, badać ich wydajności oraz porównywać je. Przed przejściem do implementacji sortowania omówiono liczne ważne kwestie, które trzeba starannie przemyśleć dla każdej techniki sortowania.

Szablon klas sortujących

```
public class Example
{
   public static void sort(Comparable[] a)
   { /* Zobacz algorytmy 2.1, 2.2, 2.3, 2.4, 2.5 lub 2.7. */ }

   private static boolean less(Comparable v, Comparable w)
   { return v.compareTo(w) < 0; }

   private static void exch(Comparable[] a, int i, int j)
   { Comparable t = a[i]; a[i] = a[j]; a[j] = t; }

   private static void show(Comparable[] a)
   {  // Wyświetla tablicę w jednym wierszu.
      for (int i = 0; i < a.length; i++)
         StdOut.print(a[i] + " ");
      StdOut.println();
   }

   public static boolean isSorted(Comparable[] a)
   {  // Sprawdza, czy elementy tablicy mają odpowiednią kolejność.
      for (int i = 1; i < a.length; i++)
         if (less(a[i], a[i-1])) return false;
      return true;
   }

   public static void main(String[] args)
   {  // Wczytuje łańcuchy znaków ze standardowego wejścia,
      // sortuje je i wyświetla.
      String[] a = In.readStrings();
      sort(a);
      assert isSorted(a);
      show(a);
   }
}
```

W klasie tej przedstawiono konwencje używane dalej do implementowania technik sortowania tablic. Dla każdego algorytmu sortowania pokazano metodę `sort()` z podobnej klasy, przy czym nazwę `Example` zmieniono na nazwę odpowiednią dla algorytmu. Klient testowy sortuje łańcuchy znaków ze standardowego wejścia, jednak metody sortowania zadziałają dla dowolnego typu danych implementującego interfejs `Comparable`.

```
% more tiny.txt
S O R T E X A M P L E

% java Example < tiny.txt
A E E L M O P R S T X
```

```
% more words3.txt
bed bug dad yes zoo ... all bad yet

% java Example < words.txt
all bad bed bug dad ... yes yet zoo
```

Sprawdzanie poprawności Czy implementacja sortowania zawsze umieszcza elementy tablicy we właściwej kolejności, niezależnie od ich początkowego uporządkowania? Stosujemy konserwatywne podejście i umieszczamy w kliencie testowym instrukcję `assert isSorted(a);`, aby sprawdzić, czy elementy tablicy są po sortowaniu odpowiednio uporządkowane. Warto umieścić tę instrukcję w *każdej* implementacji sortowania, choć zwykle testujemy kod i opracowujemy matematyczne dowody poprawności algorytmów. Warto zauważyć, że test jest wystarczający tylko wtedy, jeśli do zmiany pozycji elementów tablicy używamy wyłącznie metody `exch()`. Przy stosowaniu kodu zapisującego wartości bezpośrednio w tablicy test nie gwarantuje poprawności (za prawidłowy uznany zostanie na przykład kod niszczący pierwotną tablicę wejściową przez ustawienie wszystkich elementów na tę samą wartość).

Model kosztów dla sortowania. Przy analizowaniu algorytmów sortowania liczone są *porównania* i *przestawienia*. Dla algorytmów, które nie przestawiają elementów, liczone są *dostępy do tablicy*.

Czas wykonania Testujemy też *wydajność* algorytmów. Zaczynamy od udowodnienia faktów na temat liczby podstawowych operacji (porównań i przestawień oraz czasem liczby dostępów tablicy w celu odczytu lub zapisu), które różne algorytmy sortowania wykonują dla rozmaitych naturalnych modeli danych wejściowych. Następnie używamy tych faktów do opracowania hipotez dotyczących względnej wydajności algorytmów. Prezentujemy też narzędzia do eksperymentalnego sprawdzania hipotez. Używamy spójnego stylu kodowania, aby ułatwić tworzenie prawidłowych hipotez na temat wydajności, prawdziwych dla typowych implementacji.

Dodatkowa pamięć Ilość dodatkowej pamięci używanej przez algorytm sortowania jest często równie ważnym czynnikiem jak czas wykonania. Algorytmy sortowania dzielą się na dwa podstawowe rodzaje — sortujące *w miejscu*, które nie potrzebują dodatkowej pamięci (za wyjątkiem małego stosu wywołań funkcji lub stałej liczby zmiennych egzemplarza), oraz algorytmy wymagające dodatkowej pamięci na drugą kopię sortowanej tablicy.

Typy danych Kod sortujący działa dla elementów każdego typu obsługującego interfejs `Comparable`. Stosowanie się do konwencji Javy jest tu wygodne, ponieważ wiele typów danych obsługuje ten interfejs. Dotyczy to na przykład nakładkowych typów numerycznych Javy, takich jak `Integer` i `Double`, a także typu `String` i różnych zaawansowanych typów w rodzaju `File` lub `URL`. Wystarczy wywołać jedną z metod sortowania, podając jako argument tablicę wartości dowolnego z tych typów. Przykładowo, w kodzie po prawej stronie użyto sortowania szybkiego (zobacz PODROZDZIAŁ 2.3) do posortowania `N` losowych wartości typu `Double`. Przy samodzielnym tworzeniu typów można umożliwić w kodzie klienta sortowanie danych określonego typu, implementując inter-

```
Double a[] = new Double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.uniform();
Quick.sort(a);
```

Sortowanie tablicy losowych wartości

fejs `Comparable`. W tym celu wystarczy zaimplementować metodę `compareTo()`, która wyznacza uporządkowanie obiektów typu w tak zwanym *porządku naturalnym*, co pokazano tu dla typu danych `Date` (zobacz stronę 103). Zgodnie z konwencjami Javy wywołanie `v.compareTo(w)` zwraca liczbę całkowitą — ujemną (zwykle `-1`) dla `v<w`, zero dla `v=w` i dodatnią (zwykle `+1`) dla `v>w`. Z uwagi na zwięzłość w dalszej części akapitu używamy standardowego zapisu w rodzaju `v>w` jako skrótu dla kodu `v.compareTo(w)>0`. Wywołanie `v.compareTo(w)` powoduje wyjątek, jeśli `v` i `w` mają niezgodne typy lub jedna z tych wartości to `null`. Ponadto metoda `compareTo()` musi wyznaczać *porządek liniowy*. Musi więc być:

- *zwrotna* (`v=v` dla każdego `v`),
- *antysymetryczna* (dla wszystkich `v` i `w` jeśli `v<w`, to `w>v`, a jeżeli `v=w`, to `w=v`),
- *przechodnia* (dla wszystkich `v`, `w` i `x` jeśli `v<=w` i `w<=x`, to `v<=x`).

W matematyce reguły te są intuicyjne i standardowe. Nietrudno się do nich dostosować. Ujmijmy to krótko — metoda `compareTo()` to implementacja abstrakcyjnego *klucza*. Definiuje uporządkowanie sortowanych elementów (obiektów), które mogą być dowolnego typu obsługującego interfejs `Comparable`. Zauważmy, że w metodzie `compareTo()` nie trzeba używać wszystkich zmiennych egzemplarza. Klucz może być małą częścią każdego elementu.

```
public class Date implements Comparable<Date>
{
   private final int day;
   private final int month;
   private final int year;

   public Date(int d, int m, int y)
   { day = d; month = m; year = y; }

   public int day() { return day; }
   public int month() { return month; }
   public int year() { return year; }

   public int compareTo(Date that)
   {
      if (this.year > that.year ) return +1;
      if (this.year < that.year ) return -1;
      if (this.month > that.month) return +1;
      if (this.month < that.month) return -1;
      if (this.day > that.day ) return +1;
      if (this.day < that.day ) return -1;
      return 0;
   }

   public String toString()
   { return month + "/" + day + "/" + year; }
}
```

Definiowanie typu umożliwiającego porównywanie

W DALSZEJ CZĘŚCI ROZDZIAŁU omówiono liczne algorytmy do sortowania tablic obiektów mających porządek naturalny. Aby porównać algorytmy i przedstawić różnice między nimi, zbadano wiele ich cech, w tym liczbę porównań i przestawień dla różnego rodzaju danych wejściowych oraz ilość potrzebnej dodatkowej pamięci. Cechy te prowadzą do opracowania hipotez na temat wydajności. Wiele właściwości algorytmów sprawdzono w ostatnich dziesięcioleciach na niezliczonych komputerach. Zawsze trzeba badać specyficzne implementacje, dlatego omówiono służące do tego narzędzia. Po rozważeniu klasycznego sortowania przez wybieranie, sortowania przez wstawianie, sortowania Shella, sortowania przez scalanie, sortowania szybkiego i sortowania przez kopcowanie, W PODROZDZIALE 2.5 omówiono praktyczne zagadnienia i zastosowania.

Sortowanie przez wybieranie Jeden z najprostszych algorytmów sortowania działa tak — najpierw należy znaleźć najmniejszy element tablicy i przestawić go z pierwszym elementem (z nim samym, jeśli to obiekt na pierwszej pozycji jest najmniejszy). Następnie trzeba znaleźć kolejny najmniejszy element i przestawić go z drugim elementem. Proces jest kontynuowany do momentu posortowania całej tablicy. Metoda ta nosi nazwę *sortowanie przez wybieranie*, ponieważ oparta jest na wielokrotnym wybieraniu najmniejszego z pozostałych elementów.

Jak widać na podstawie implementacji w ALGORYTMIE 2.1, pętla wewnętrzna w sortowaniu przez wybieranie jedynie porównuje bieżący element z najmniejszym ze znalezionych do tej pory (dodatkowy kod zwiększa bieżący indeks i sprawdza, czy jego wartość nie wyszła poza granice tablicy). Trudno napisać prostszy kod. Operacja przenoszenia elementów znajduje się poza pętlą wewnętrzną. Każde przestawienie prowadzi do umieszczenia elementu na ostatecznej pozycji, dlatego liczba przestawień wynosi N. Tak więc czas wykonania jest zależny od liczby porównań.

Twierdzenie A. Sortowanie przez wybieranie wymaga $\sim N^2/2$ porównań i N przestawień.

Dowód. Można to udowodnić, analizując ślad działania algorytmu. Jest nim tabela o wymiarach N na N, w której litery w kolorze innym niż szary odpowiadają porównaniom. Około połowa elementów tablicy (te na przekątnej i nad nią) jest w kolorze innym niż szary. Każdy element na przekątnej odpowiada przestawieniu. Ujmijmy to dokładniej — na podstawie analizy kodu można stwierdzić, że dla każdego i między 0 a $N - 1$ potrzeba jednego przestawienia i $N - 1 - i$ porównań, co daje w sumie N przestawień i $(N - 1) + (N - 2) + \ldots + 2 + 1 + 0 = N(N - 1) / 2 \sim N^2 / 2$ porównań.

PODSUMUJMY — sortowanie przez wybieranie to prosta metoda sortowania, łatwa do zrozumienia i zaimplementowania. Oto dwie specyficzne dla niej cechy.

Czas wykonania jest niezależny od danych wejściowych Proces wyszukiwania najmniejszego elementu w jednym przejściu przez tablicę nie zapewnia informacji o tym, gdzie może znajdować się najmniejszy element w następnym przejściu. Cecha ta w niektórych sytuacjach jest wadą. Przykładowo, osoba używająca klienta do sortowania może być zaskoczona, kiedy stwierdzi, że sortowanie przez wybieranie działa równie długo dla już uporządkowanej tablicy lub dla tablicy, w której wszystkie klucze są takie same, jak dla losowo uporządkowanej tablicy! Jak się okaże, inne algorytmy lepiej wykorzystują początkowe uporządkowanie danych wejściowych.

Potrzebna jest minimalna liczba przestawień Każde z N przestawień zmienia wartość dwóch elementów tablicy, dlatego sortowanie przez wybieranie wymaga N przestawień. Liczba dostępów do tablicy rośnie *liniowo* wraz z wielkością tablicy. Żaden inny z omawianych algorytmów sortowania nie posiada tej cechy (w większości wzrost jest liniowo-logarytmiczny lub kwadratowy).

ALGORYTM 2.1. Sortowanie przez wybieranie

```
public class Selection
{
   public static void sort(Comparable[] a)
   {  // Sortowanie a[] w porządku rosnącym.
      int N = a.length; // Długość tablicy.
      for (int i = 0; i < N; i++)
      {  // Przestawianie a[i] z najmniejszym elementem z a[i+1...N).
         int min = i; // Indeks minimalnego elementu.
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min])) min = j;
         exch(a, i, min);
      }
   }
   // Metody less(), exch(), isSorted() i main() przedstawiono na stronie 257.
}
```

Dla każdego `i` implementacja umieszcza `i`-ty najmniejszy element w `a[i]`. Elementy na lewo od `i` to `i` najmniejszych elementów. Nie są one ponownie sprawdzane.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Czarne elementy są sprawdzane w celu znalezienia minimum

Czerwone elementy to `a[min]`

Szare elementy znajdują się na ostatecznej pozycji

Ślad działania sortowania przez wybieranie (zawartość tablicy po każdym przestawieniu)

Sortowanie przez wstawianie Algorytm często stosowany do sortowania kart w czasie gry w brydża polega na sprawdzaniu kolejnych kart i umieszczaniu ich w odpowiednim miejscu wśród wcześniej ułożonych (przy zachowaniu uporządkowania w tej grupie). W implementacji komputerowej trzeba zrobić miejsce na wstawienie bieżącego elementu, przenosząc większe elementy o jedno miejsce w prawo przed wstawieniem danego na wolną pozycję. ALGORYTM 2.2 to implementacja tej techniki, nazywanej *sortowaniem przez wstawianie*.

Tu, podobnie jak w sortowaniu przez wybieranie, elementy na lewo od bieżącego indeksu są posortowane, jednak nie znajdują się na ostatecznej pozycji, ponieważ konieczne może być ich przeniesienie w celu zrobienia miejsca na mniejsze, później napotkane elementy. Jednak po dojściu indeksu do prawego końca tablica jest w pełni posortowana.

Czas wykonania sortowania przez wstawianie zależy od początkowego układu elementów w danych wejściowych (inaczej niż w sortowaniu przez wybieranie). Przykładowo, jeśli tablica jest duża, a elementy są już uporządkowane (lub prawie posortowane), sortowanie jest dużo szybsze niż dla elementów rozmieszczonych losowo albo w odwrotnej kolejności.

Twierdzenie B. Sortowanie przez wstawianie wymaga średnio $\sim N^2/4$ porównań i $\sim N^2/4$ przestawień dla losowo uporządkowanej tablicy o długości N i niepowtarzalnych kluczach. W najgorszym przypadku potrzeba $\sim N^2/2$ porównań i $\sim N^2/2$ przestawień, a w najlepszym przypadku jest to $N - 1$ porównań i 0 przestawień.

Dowód. Podobnie jak w TWIERDZENIU A, tak i tu liczbę porównań i przestawień łatwo jest zwizualizować w tabeli o wymiarach N na N używanej do ilustrowania sortowania. Należy policzyć elementy pod przekątną. W najgorszym przypadku należy uwzględnić wszystkie elementy, a w najlepszym zbiór nie obejmuje żadnego elementu. Dla losowo uporządkowanych tablic można oczekiwać, że każdy element trzeba średnio przesunąć o mniej więcej połowę, dlatego uwzględniamy połowę elementów pod przekątną.

Liczba porównań to liczba przestawień plus dodatkowa wartość równa N minus liczba sytuacji, w których wstawiany element jest najmniejszy spośród dotychczas znalezionych. W najgorszym przypadku (tablica w odwrotnej kolejności) wartość ta jest nieistotna w stosunku do łącznej liczby porównań. W najlepszym przypadku (tablica posortowana) porównań jest $N - 1$.

Sortowanie przez wstawianie działa dobrze dla pewnego rodzaju nielosowych tablic, które często powstają w praktyce (nawet jeśli tablice są bardzo duże). Rozważmy na przykład, co się stanie po zastosowaniu sortowania przez wstawianie dla już posortowanej tablicy. Algorytm natychmiast stwierdzi, że każdy element znajduje się we właściwym miejscu tablicy, a łączny czas wykonania rośnie liniowo (czas wykonania sortowania przez wybieranie dla takich tablic jest kwadratowy). To samo dotyczy tablic, w których wszystkie klucze są równe (stąd warunek niepowtarzalności kluczy w TWIERDZENIU B).

ALGORYTM 2.2. Sortowanie przez wstawianie

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {  // Sortowanie a[] w porządku rosnącym.
      int N = a.length;
      for (int i = 1; i < N; i++)
      {  // Wstawianie a[i] między a[i-1], a[i-2], a[i-3] itd.
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
      }
   }
   // Metody less(), exch(), isSorted() i main() przedstawiono na stronie 257.
}
```

Dla każdego `i` z przedziału od 0 do `N-1` należy przestawić `a[i]` z mniejszymi elementami z przedziału od `a[0]` do `a[i-1]`. Przy przesuwaniu indeksu `i` od lewej do prawej elementy po lewej stronie są posortowane, dlatego po dotarciu `i` do prawego końca tablica jest posortowana.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Szare elementy pozostają w miejscu

Czerwony element to `a[j]`

Czarne elementy należy przenieść o jedno miejsce w prawo przy wstawianiu

Ślad działania sortowania przez wstawianie (zawartość tablicy po każdym wstawianiu)

Rozważmy bardziej ogólne zagadnienie, związane z *częściowo posortowanymi* tablicami. *Inwersja* to para elementów tablicy uporządkowanych w niewłaściwy sposób. W słowie `E X A M P L E` występuje 11 inwersji: `E-A`, `X-A`, `X-M`, `X-P`, `X-L`, `X-E`, `M-L`, `M-E`, `P-L`, `P-E` i `L-E`. Jeśli liczba inwersji w tablicy jest mniejsza niż pewna stała wielokrotność wielkości tablicy, mówimy, że tablica jest *częściowo posortowana*. Oto typowe przykłady częściowo posortowanych tablic:

- Tablica, w której każdy element znajduje się niedaleko ostatecznej pozycji.
- Krótka tablica dołączona do długiej posortowanej tablicy.
- Tablica, w której niewielka liczba elementów znajduje się nie na swoim miejscu.

Sortowanie przez wstawianie (w przeciwieństwie do sortowania przez wybieranie) jest wydajną metodą dla takich tablic. Jeśli liczba inwersji jest niska, sortowanie przez wstawianie jest często szybsze niż jakakolwiek inna metoda sortowania omówiona w rozdziale.

Twierdzenie C. Liczba przestawień w sortowaniu przez wstawianie jest równa liczbie inwersji w tablicy, a liczba porównań wynosi przynajmniej liczbę inwersji, a najwyżej liczbę inwersji plus wielkość tablicy minus 1.

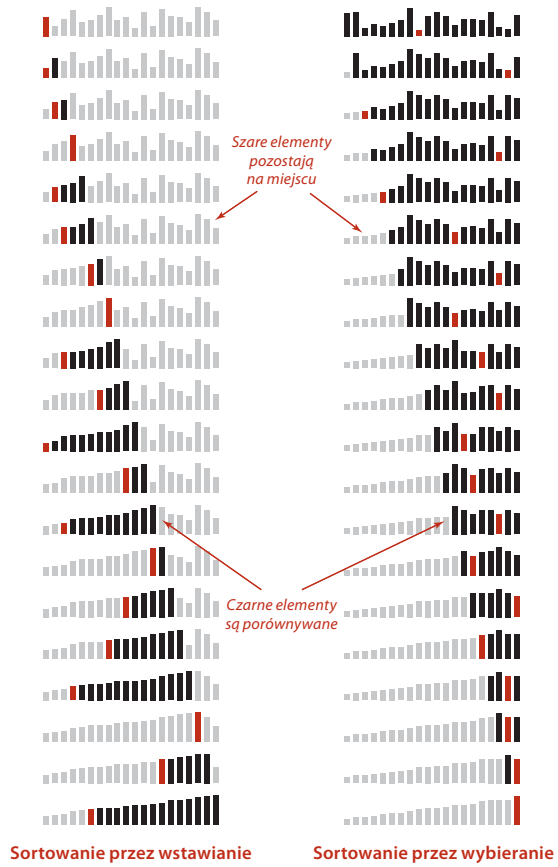
Dowód. Każde przestawienie dotyczy dwóch przyległych elementów ustawionych w złej kolejności, a tym samym zmniejsza liczbę inwersji o jeden, a tablica jest posortowana, kiedy liczba inwersji dochodzi do zera. Każde przestawienie wymaga porównania. Ponadto mogą mieć miejsce dodatkowe porównania dla każdej wartości `i` z przedziału od `1` do `N-1` (jeśli `a[i]` nie dociera do lewego końca tablicy).

Można łatwo znacznie przyspieszyć sortowanie przez wstawianie, skracając wewnętrzną pętlę tak, aby przenosiła większe elementy o jedną pozycję w prawo, zamiast wykonywać pełne przestawianie (pozwala to zmniejszyć liczbę dostępów do tablicy o połowę). Wprowadzenie tego usprawnienia pozostawiamy jako ćwiczenie (zobacz ĆWICZENIE 2.1.25).

PODSUMOWANIE — sortowanie przez wstawianie to doskonała metoda dla częściowo posortowanych tablic. Jest też dobrą techniką dla krótkich tablic. Ma to znaczenie nie tylko z uwagi na to, że takie tablice często występują w praktyce, ale też dlatego, iż tablice obu rodzajów powstają na etapach pośrednich w zaawansowanych algorytmach sortujących. Dlatego sortowanie przez wstawianie omówiono ponownie w kontekście takich algorytmów.

Wizualizacja działania algorytmów sortujących W tym rozdziale używamy prostej reprezentacji wizualnej do opisywania algorytmów sortujących. Zamiast śledzić postępy sortowania za pomocą wartości kluczy, na przykład liter, liczb lub słów, używamy pionowych słupków sortowanych według wysokości. Zaletą takiej reprezentacji jest to, że pozwala zrozumieć działanie metody.

Po prawej stronie, w wizualnych śladach działania, od razu widać, że w sortowaniu przez wstawianie elementy na prawo od indeksu nie są uwzględniane, natomiast w sortowaniu przez wybieranie nie są sprawdzane elementy na lewo od indeksu. Ponadto wyraźnie widać, że sortowanie przez wstawianie nie wymaga przenoszenia elementów mniejszych od wstawianego i wykonuje średnio około połowy porównań potrzebnych w sortowaniu przez wybieranie.



Wizualny ślad działania podstawowych algorytmów sortujących

Za pomocą opracowanej przez nas biblioteki `StdDraw` tworzenie wizualnego śladu nie jest trudniejsze od generowania zwykłego śladu. Należy posortować wartości typu `Double`, dopracować algorytm tak, aby wywoływał metodę `show()` w odpowiedni sposób (tak jak dla standardowego śladu), i opracować wersję metody `show()`, żeby korzystała z biblioteki `StdDraw` do rysowania słupków, zamiast wyświetlać wyniki. Najbardziej skomplikowanym zadaniem jest określenie skali dla osi *y* tak, aby kolejne rysunki pojawiły się w oczekiwanej kolejności. Zachęcamy do wykonania ĆWICZENIA 2.1.18. Pozwoli to docenić wartość wizualnego śladu i ułatwi jego tworzenie.

Jeszcze łatwiejszym zadaniem jest utworzenie *animacji* na podstawie śladu działania, co pozwoli zobaczyć dynamiczne sortowanie tablicy. Animacja oparta jest na procesie opisanym w poprzednim akapicie, jednak nie trzeba tu martwić się o oś *y* (wystarczy za każdym razem wyczyścić zawartość okna i ponownie wyświetlić słupki). Choć nie można tego pokazać na kartach książki, animowane reprezentacje także pomagają zrozumieć działanie algorytmów. Zachęcamy do wykonania ĆWICZENIA 2.1.17, co pozwoli się o tym przekonać.

Porównywanie dwóch algorytmów sortujących Mamy już dwie implementacje i oczywiście ciekawe jest, która z nich jest szybsza — sortowanie przez wybieranie (ALGORYTM 2.1) czy sortowanie przez wstawianie (ALGORYTM 2.2). Pytania tego rodzaju pojawiają się wielokrotnie w czasie badań algorytmów i są głównym tematem tej książki. Pewne podstawowe kwestie omówiono w ROZDZIALE 1., jednak ten pierwszy przykład wykorzystamy do przedstawienia podstawowego podejścia do udzielania odpowiedzi na podobne pytania. Ogólnie, stosując podejście wprowadzone w PODROZDZIALE 1.4, porównujemy algorytmy przez:

- ich zaimplementowanie i zdiagnozowanie,
- przeanalizowanie podstawowych cech,
- sformułowanie hipotez na temat względnej wydajności,
- przeprowadzenie eksperymentów w celu sprawdzenia hipotez.

Kroki te są ni mniej, nie więcej jak sprawdzoną *metodą naukową* zastosowaną do badania algorytmów.

W tym kontekście ALGORYTM 2.1 i ALGORYTM 2.2 dotyczą pierwszego kroku. TWIERDZENIA A, B i C stanowią drugi krok. CECHA D ze strony 267 to krok trzeci, a klasa `SortCompare` ze strony 268 umożliwia wykonanie czwartego kroku. Wszystkie etapy są ze sobą powiązane.

Krótkie opisy powodują, że nie widać dużej ilości pracy potrzebnej do poprawnego zaimplementowania, przeanalizowania i przetestowania algorytmów. Każdy programista wie, że kod jest efektem długiego diagnozowania i usprawniania; każdy matematyk zdaje sobie sprawę, iż poprawne analizy bywają bardzo skomplikowane; każdy naukowiec wie, że formułowanie hipotez oraz projektowanie i wykonywanie eksperymentów w celu ich sprawdzenia wymaga olbrzymiej staranności. Kompletne opracowanie wyników pozostawiamy ekspertom badającym najważniejsze algorytmy, jednak każdy programista stosujący algorytm powinien znać naukowy kontekst, który pozwolił ustalić cechy algorytmu w obszarze wydajności.

Po opracowaniu implementacji następny krok polega na ustaleniu odpowiedniego modelu danych wejściowych. Dla sortowania naturalnym modelem, który wykorzystano w TWIERDZENIACH A, B i C, jest uznanie, że tablice są losowo uporządkowane *oraz* że wartości kluczy są niepowtarzalne. W zastosowaniach, w których pojawia się duża liczba kluczy o tej samej wartości, potrzebny jest bardziej skomplikowany model.

Jak można sformułować hipotezę dotyczącą czasu wykonania sortowania przez wstawianie i wybieranie dla losowo uporządkowanych tablic? Z analizy ALGORYTMÓW 2.1 i 2.2 oraz TWIERDZEŃ A i B bezpośrednio wynika, że dla losowych danych czas wykonania obu algorytmów powinien być kwadratowy. Oznacza to, że czas sortowania przez wstawianie jest proporcjonalny do małej stałej razy N^2, a sortowania przez wybieranie — do innej małej stałej razy N^2. Wartości obu stałych zależą od kosztów porównań i przestawień na danym komputerze. Dla wielu typów danych i standardowych komputerów sensowne jest założenie, że koszty te są zbliżone (choć istnieje kilka ważnych wyjątków). Bezpośrednio wynikają z tego następujące hipotezy.

Cecha D. Dla losowo uporządkowanych tablic niepowtarzalnych wartości czas sortowania przez wstawianie i sortowania przez wybieranie jest kwadratowy, a szybkość tych algorytmów różni się o niewielką stałą.

Dowód. Stwierdzenie to przez ostatnie pół wieku potwierdzono na wielu komputerach. Sortowanie przez wstawianie było około dwukrotnie szybsze od sortowania przez wybieranie w czasie pisania pierwszego wydania tej książki (w roku 1980) i nadal tak jest, choć wtedy posortowanie 100 000 elementów za pomocą tych algorytmów zajmowało kilka godzin, a obecnie dzieje się to w kilka sekund. Czy na Twoim komputerze sortowanie przez wstawianie jest nieco szybsze od sortowania przez wybieranie? Aby to sprawdzić, możesz użyć klasy `SortCompare` z następnej strony. W klasie używana jest metoda `sort()` z klas o nazwach podanych jako argumenty wiersza poleceń do wykonania określonej liczby eksperymentów (sortowania tablic o danym rozmiarze). Program wyświetla stosunek odnotowanych czasów wykonania algorytmów.

Aby sprawdzić hipotezę, przeprowadzono eksperymenty za pomocą klasy `SortCompare` (zobacz stronę 268). Jak zwykle do ustalenia czasu wykonania służy klasa `Stopwatch`. Pokazana tu implementacja metody `time()` działa dla podstawowych technik sortowania opisanych w rozdziale. Metoda `timeRandomInput()` z klasy `SortCompare` działa zgodnie z modelem losowo uporządkowanych danych wejściowych — generuje losowe wartości typu `Double`, sortuje je i zwraca łączny czas sortowania dla określonej liczby prób. Wykorzystanie losowych wartości typu `Double` z przedziału od `0.0` do `1.0` jest dużo prostsze niż użycie funkcji bibliotecznej w rodzaju `StdRandom.shuffle()`. Jest to też skuteczne podejście, ponieważ wystąpienie kluczy o równej wartości jest bardzo mało prawdopodobne (zobacz ĆWICZENIE 2.5.31). Jak opisano to W ROZDZIALE 1., liczba prób jest pobierana jako argument, co pozwala wykorzystać prawo wielkich liczb (im więcej prób, tym podzielenie łącznego czasu pracy przez liczbę powtórzeń daje dokładniejsze szacunki rzeczywistego średniego czasu wykonania) i zniwelować efekty obciążenia systemu. Zachęcamy do eksperymentów z programem `SortCompare` na własnym komputerze. Pomaga to poznać stopień, w jakim wnioski na temat sortowania przez wstawianie i wybieranie są prawdziwe.

```
public static double time(String alg, Comparable[] a)
{
   Stopwatch timer = new Stopwatch();
   if (alg.equals("Insertion")) Insertion.sort(a);
   if (alg.equals("Selection")) Selection.sort(a);
   if (alg.equals("Shell"))     Shell.sort(a);
   if (alg.equals("Merge"))     Merge.sort(a);
   if (alg.equals("Quick"))     Quick.sort(a);
   if (alg.equals("Heap"))      Heap.sort(a);
   return timer.elapsedTime();
}
```

Pomiar czasu pracy jednego z algorytmów sortujących z tego rozdziału dla określonych danych

Porównywanie dwóch algorytmów sortujących

```
public class SortCompare
{
   public static double time(String alg, Double[] a)
   { /* Zobacz tekst. */ }

   public static double timeRandomInput(String alg, int N, int T)
   {  // Użycie algorytmu alg do posortowania T losowych tablic
      // o długości N.
      double total = 0.0;
      Double[] a = new Double[N];
      for (int t = 0; t < T; t++)
      {  // Przeprowadzenie jednego eksperymentu (generowanie
         // i sortowanie tablicy).
         for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
         total += time(alg, a);
      }
      return total;
   }

   public static void main(String[] args)
   {
      String alg1 = args[0];
      String alg2 = args[1];
      int N = Integer.parseInt(args[2]);
      int T = Integer.parseInt(args[3]);
      double t1 = timeRandomInput(alg1, N, T); // Suma dla alg1.
      double t2 = timeRandomInput(alg2, N, T); // Suma dla alg2.
      StdOut.printf("Dla %d losowych wartości Double\n technika %s jest",
N, alg1);
      StdOut.printf(" %.1f razy szybsza od %s\n", t2/t1, alg2);
   }
}
```

Ten klient uruchamia dwie techniki sortowania (ich nazwy podano w pierwszych dwóch argumentach wiersza poleceń) dla tablicy zawierającej `N` (trzeci argument) losowych wartości typu `Double` z przedziału od `0.0` do `1.0`, ponawia eksperyment `T` razy (czwarty argument wiersza poleceń), a następnie wyświetla stosunek łącznych czasów działania.

```
% java SortCompare Insertion Selection 1000 100
Dla 1000 losowych wartości Double
   technika Insertion jest 1.7 razy szybsza od Selection
```

CECHA D celowo jest nieco niejasna (wartość małej stałej jest nieokreślona, a ponadto nie ma założenia o podobnych kosztach porównań i przestawień), dlatego okazuje się prawdziwa w wielu sytuacjach. Kiedy to możliwe, kluczowe aspekty wydajności każdego z analizowanych algorytmów staramy się ująć w stwierdzeniach tego rodzaju. Jak opisano to w ROZDZIALE 1., każda omawiana *Cecha* wymaga naukowego przetestowania w danej sytuacji, czasem z wykorzystaniem bardziej dopracowanych hipotez opartych na powiązanym *Twierdzeniu* (matematycznej prawdzie).

W kontekście praktycznych zastosowań jest jeszcze jeden kluczowy krok — *przeprowadzenie eksperymentów w celu walidacji hipotez dla używanych danych*. Omawianie tego etapu odkładamy do PODROZDZIAŁU 2.5 i ćwiczeń. Jeśli w omawianym przykładzie klucze sortujące nie są unikatowe i (lub) losowo uporządkowane, CECHA D może nie być prawdziwa. Tablicę można losowo uporządkować za pomocą metody `StdRandom.shuffle()`, jednak aplikacje z dużą liczbą równych kluczy wymagają dokładnych analiz.

Omówienie analiz algorytmów ma stanowić punkt wyjścia — nie mają to być ostateczne wnioski. Jeśli zainteresują Cię inne kwestie dotyczące wydajności algorytmów, możesz je zbadać za pomocą narzędzia w rodzaju `SortCompare`. Ćwiczenia dają wiele okazji do przeprowadzenia takich badań.

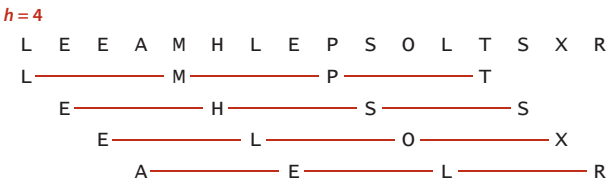
NIE ZAGŁĘBIAMY się bardziej w porównywanie wydajności sortowania przez wstawianie i wybieranie, ponieważ o wiele bardziej interesują nas algorytmy działające od nich setki, tysiące, a nawet miliony razy szybciej. Jest jednak kilka powodów, dla których warto zrozumieć podstawowe algorytmy. Algorytmy te:

- Pomagają poznać podstawowe zasady.
- Zapewniają punkt odniesienia w obszarze wydajności.
- Są stosowane w pewnych specjalnych sytuacjach.
- Mogą być podstawą do rozwijania lepszych algorytmów.

Z tych powodów stosujemy to samo podejście i rozważamy podstawowe algorytmy dla każdego problemu omawianego w książce — nie tylko do sortowania. Programy w rodzaju `SortCompare` odgrywają kluczową rolę w technice stopniowego rozwijania algorytmów. Na każdym etapie można użyć takiego programu do ocenienia, czy nowy algorytm lub usprawniona wersja znanego zapewnia oczekiwane zyski wydajności.

Sortowanie Shella Aby pokazać znaczenie znajomości podstawowych metod sortowania, omawiamy szybki algorytm oparty na sortowaniu przez wstawianie. Sortowanie przez wstawianie jest wolne dla dużych nieuporządkowanych tablic, ponieważ jedyne przestawienia dotyczą tu przyległych elementów, dlatego wartości można przenosić w tablicy tylko po jednym miejscu. Jeśli element o najmniejszym kluczu znajduje się na końcu tablicy, potrzeba $N - 1$ przestawień, aby umieścić go na docelowej pozycji. *Sortowanie Shella* to proste rozwinięcie sortowania przez wstawianie. Przyspieszenie działania wynika tu z możliwości przestawiania oddalonych elementów tablicy. Prowadzi to do powstawania częściowo posortowanych tablic, które można ostatecznie wydajnie posortować za pomocą sortowania przez wstawianie.

Pomysł polega na uporządkowaniu tablicy w taki sposób, aby co h-te elementy (rozpoczynając od dowolnego miejsca) były posortowanymi podciągami. Mówimy, że taka tablica jest po h-sortowaniu. Ujmijmy to inaczej — tablica po h-sortowaniu to h niezależnie posortowanych i wymieszanych ze sobą podciągów. Przeprowadzając h-sortowanie dla dużych wartości h można przenosić elementy tablicy na duże odległości, co ułatwia h-sortowanie dla mniejszych wartości h. Zastosowanie takiej procedury dla dowolnego ciągu wartości h kończącego się wartością 1 daje posortowaną tablicę. Tak działa sortowanie Shella. W implementacji w ALGORYTMIE 2.3, pokazanym na następnej stronie, użyto ciągu malejących wartości $\frac{1}{2}(3^k - 1)$. Rozpoczęto od największego przyrostu mniejszego od $N/3$, po czym jest on zmniejszany o 1. Taki ciąg nazywany jest *ciągiem odstępów*. ALGORYTM 2.3 sam oblicza ciąg odstępów. Inna możliwość to zapisanie takiego ciągu w tablicy.



Ciąg po h-sortowaniu to h wymieszanych posortowanych podciągów

Jednym ze sposobów na zaimplementowanie sortowania Shella jest użycie — dla każdego h — sortowania przez wstawianie niezależnie dla każdego z h podciągów. Ponieważ podciągi są niezależne, można użyć jeszcze prostszego podejścia. Przy h-sortowaniu tablicy wystarczy wstawić każdy element między poprzednie w podciągu dla danego h, przestawiając go z elementami o wyższych kluczach (przenosząc te ostatnie o jedną pozycję w prawo w podciągu). Do wykonania tego zadania używamy kodu sortowania przez wstawianie, zmodyfikowanego tak, aby dekrementacja wynosiła h zamiast 1 przy poruszaniu się po tablicy. Ta obserwacja pozwala zredukować implementację sortowania Shella do procesu podobnego do sortowania przez wstawianie dla każdego odstępu.

Sortowanie Shella zapewnia wydajność przez równoważenie rozmiaru i częściowego uporządkowania (w podciągach). Początkowo podciągi są krótkie. Na dalszych etapach podciągi są częściowo posortowane. W obu sytuacjach uruchamiane jest sortowanie przez wstawianie. Stopień częściowego posortowania podciągów jest zmienny i zależy w dużym stopniu od ciągu odstępów. Określenie wydajności sortowania Shella nie jest proste. ALGORYTM 2.3 to jedyna z omawianych tu metod sortowania, dla której nie scharakteryzowano dokładnie wydajności dla losowo uporządkowanych tablic.

ALGORYTM 2.3. Sortowanie Shella

```
public class Shell
{
   public static void sort(Comparable[] a)
   {  // Sortowanie a[] w kolejności rosnącej.
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...
      while (h >= 1)
      {  // h-sortowanie tablicy.
         for (int i = h; i < N; i++)
         {  // Wstawianie a[i] między a[i-h], a[i-2*h], a[i-3*h] itd.
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }
         h = h/3;
      }
   }
   // Metody less(), exch(), isSorted() i main() opisano na stronie 257.
}
```

Oto zwięzła implementacja sortowania Shella. Należało zmodyfikować wstawianie przez sortowanie (ALGORYTM 2.2) pod kątem *h*-sortowania tablicy i dodać pętlę zewnętrzną do zmniejszania wartości *h* w ciągu odstępów, który zaczyna się od stałej części tablicy, a kończy wartością 1.

```
% java SortCompare Shell Insertion 100000 100
Dla 100000 losowych wartości Double
   technika Shell jest 600 razy szybsza od Insertion
```

Dane wejściowe	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sortowanie	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sortowanie	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sortowanie	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Ślad działania sortowania Shella (zawartość tablicy po każdym przejściu)

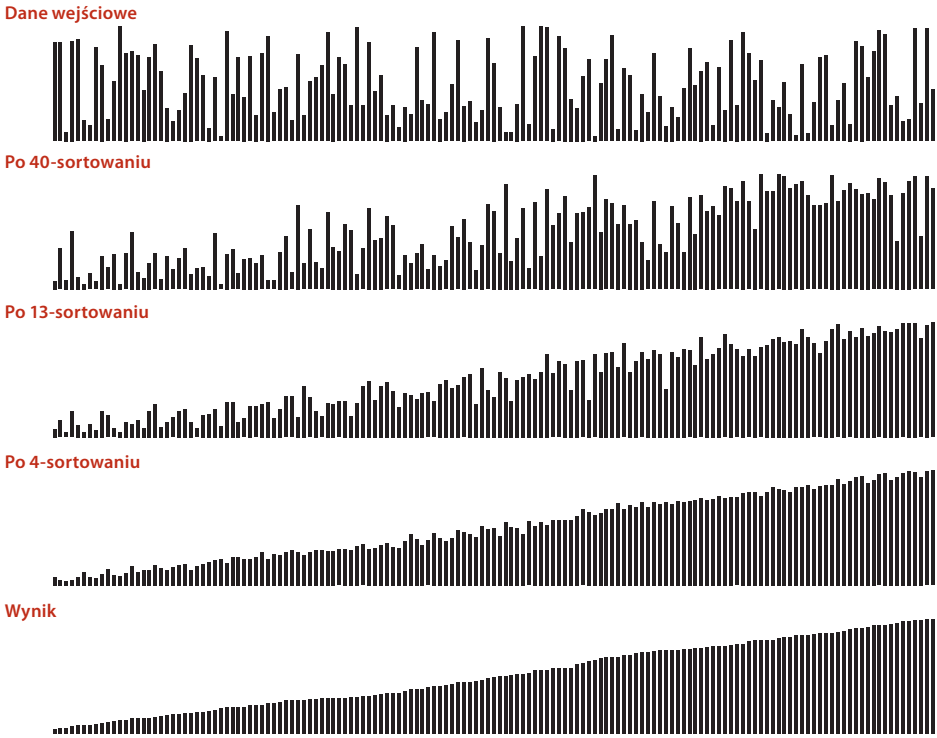
Jak ustalić, który ciąg odstępów należy zastosować? Zwykle trudno jest odpowiedzieć na to pytanie. Wydajność algorytmu zależy nie tylko od wartości odstępów, ale też od arytmetycznych zależności między nimi, na przykład ich wspólnymi dzielnikami i innymi cechami. Przebadano wiele różnych ciągów odstępów, jednak nie udowodniono, że któryś z nich jest najlepszy. Ciąg odstępów zastosowany w ALGORYTMIE 2.3 jest łatwy do obliczenia i w użyciu oraz zapewnia wydajność niemal tak wysoką, jak bardziej zaawansowane ciągi odstępów, dla których udowodniono wyższą wydajność dla najgorszego przypadku. Możliwe, że ciągi odstępów o znacząco wyższej wydajności wciąż czekają na odkrycie.

Dane wejściowe	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sortowanie	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sortowanie	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	A	P	H	O	L	T	S	X	R	M	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	L	L	P	S	O	R	T	S	X	E
	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sortowanie	E	L	E	A	M	H	L	E	P	S	O	L	T	S	X	R
	E	E	L	A	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	M	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	L	M	E	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	O	P	S	L	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X
Wynik	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Szczegółowy ślad działania sortowania Shella (wstawianie)

Sortowanie Shella jest przydatne nawet dla dużych tablic, zwłaszcza w porównaniu z sortowaniem przez wybieranie i wstawianie. Działa też dobrze dla dowolnie (niekoniecznie losowo) uporządkowanych tablic. Utworzenie tablicy, dla której sortowanie Shella działa powoli dla określonego ciągu odstępów, jest zwykle trudne.

Za pomocą programu `SortCompare` można się przekonać, że sortowanie Shella jest znacznie szybsze od sortowania przez wstawianie lub wybieranie, a przewaga szybkości rośnie wraz z rozmiarem tablicy. Przed dalszą lekturą zastosuj na swoim komputerze program `SortCompare` do porównania sortowania Shella z sortowaniem przez wstawianie i wybieranie dla tablic o rozmiarach będących potęgami dwójki (zobacz ĆWICZENIE 2.1.27). Przekonasz się, że sortowanie Shella umożliwia rozwiązanie problemów, z którymi nie radzą sobie prostsze algorytmy. Ten przykład to pierwsza praktycz-



Wizualny ślad działania sortowania Shella

na ilustracja ważnej zasady pojawiającej się na kartach książki — *osiągnięcie zysków w szybkości umożliwiających rozwiązanie problemów, z którymi nie można poradzić sobie w inny sposób, jest jedną z głównych przyczyn prowadzenia badań nad wydajnością i projektowaniem algorytmów*.

Zbadanie cech z obszaru wydajności sortowania Shella wymaga matematycznych analiz wykraczających poza zakres tej książki. Jeśli chcesz się o tym przekonać, zastanów się nad tym, jak udowodnić następujący fakt — *tablica posortowana według h-sortowania pozostaje taka po k-sortowaniu*. Jeśli chodzi o wydajność ALGORYTMU 2.3, najważniejsza jest tu wiedza o tym, że *czas wykonania sortowania Shella nie musi być kwadratowy*. Wiadomo na przykład, że dla najgorszego przypadku liczba porównań w ALGORYTMIE 2.3 jest proporcjonalna do $N^{3/2}$. To, że prosta modyfikacja pozwala złamać barierę kwadratowego czasu wykonania, jest ciekawym spostrzeżeniem, zwłaszcza że uzyskanie tego efektu jest głównym celem w wielu problemach z obszaru projektowania algorytmów.

Nie istnieją matematyczne dane dotyczące średniej liczby porównań w sortowaniu Shella dla losowo uporządkowanych danych wejściowych. Opracowano ciągi odstępów, które pozwalają zmniejszyć asymptotyczny wzrost liczby porównań dla najgorszego przypadku do $N^{4/3}$, $N^{5/4}$, $N^{6/5}$ i tak dalej, jednak wiele z tych badań ma znaczenie akademickie, ponieważ dla stosowanych w praktyce wartości N poszczególne funkcje prawie nie różnią się od siebie (i od stałego czynnika N).

W praktyce można bezpiecznie wykorzystać dawne badania naukowe nad sortowaniem Shella, stosując ciąg odstępów z ALGORYTMU 2.3 (lub jeden z ciągów odstępów przedstawionych w ćwiczeniach w końcowej części podrozdziału; ciągi te pozwalają zwiększyć wydajność o 20 – 40%). Ponadto można łatwo przeprowadzić walidację przedstawionych poniżej hipotez.

Cecha E. Liczba porównań w sortowaniu Shella o odstępach 1, 4, 13, 40, 121, 364 i tak dalej jest ograniczona przez mały mnożnik N razy liczba użytych odstępów.

Dowód. Zmodyfikowanie ALGORYTMU 2.3 tak, aby zliczał porównania i dzielił je przez liczbę odstępów, to proste ćwiczenie (zobacz ĆWICZENIE 2.1.12). Według rozbudowanych eksperymentów średnia liczba porównań na odstęp może wynosić $N^{1/5}$, jednak dość trudno jest określić tempo wzrostu tej funkcji dla niedużych N. Cecha ta wydaje się dość mało zależna od modelu danych wejściowych.

DOŚWIADCZENI PROGRAMIŚCI czasem stosują sortowanie Shella, ponieważ zapewnia akceptowalny czas wykonania nawet dla stosunkowo dużych tablic, wymaga małej ilości kodu i nie zajmuje dodatkowej pamięci. W kilku następnych podrozdziałach opisano metody, które są wydajniejsze, ale — za wyjątkiem bardzo dużych N — tylko dwukrotnie (lub nawet mniej), a ponadto są bardziej skomplikowane. Jeśli potrzebujesz metody sortowania, a sortowanie systemowe jest niedostępne (kod ma działać na przykład na sprzęcie lub w systemie zagnieżdżonym), możesz swobodnie zastosować sortowanie Shella, a później ustalić, czy warto zastąpić je bardziej zaawansowanym rozwiązaniem.

Pytania i odpowiedzi

P. Sortowanie wydaje się sztucznym problemem. Czy nie istnieje wiele innych, dużo ciekawszych zadań wykonywanych za pomocą komputerów?

O. Możliwe, jednak liczne z tych ciekawych operacji *są możliwe* dzięki szybkim algorytmom sortowania. Wiele przykładów znajdziesz w PODROZDZIALE 2.5 i w dalszych fragmentach książki. Warto teraz zapoznać się z sortowaniem, ponieważ problem ten jest łatwy do zrozumienia i pozwala docenić pomysłowość twórców szybszych algorytmów.

P. Dlaczego istnieje tak wiele algorytmów sortowania?

O. Jednym z powodów jest to, że wydajność wielu algorytmów zależy od danych wejściowych, dlatego poszczególne algorytmy mogą być odpowiednie dla różnych zastosowań i określonych rodzajów danych. Przykładowo, sortowanie przez wstawianie jest metodą wybieraną dla częściowo posortowanych lub krótkich tablic. Ważne są też inne ograniczenia, takie jak pamięć i sposób traktowania równych kluczy. Do tego pytania wracamy w PODROZDZIALE 2.5.

P. Po co stosować krótkie metody pomocnicze w rodzaju `less()` i `exch()`?

O. Są to podstawowe abstrakcyjne operacje potrzebne w każdym algorytmie sortowania, a kod jest bardziej zrozumiały dzięki zastosowaniu tych operacji. Ponadto metody te pozwalają przenosić kod bezpośrednio do innych środowisk. Duża część kodu ALGORYTMÓW 2.1 i 2.2 to kod prawidłowy także w kilku innych językach programowania. Nawet w Javie można wykorzystać ten kod jako podstawę do sortowania typów prostych (bez interfejsu `Comparable`). Wystarczy zaimplementować metodę `less()` za pomocą kodu `v < w`.

P. Kiedy uruchamiam program `SortCompare`, za każdym razem otrzymuję inne wyniki (różne od tych z książki). Dlaczego tak się dzieje?

O. Zacznijmy od tego, że masz inny komputer od używanego przez nas; dotyczy to też systemu operacyjnego, środowiska Javy itd. Wszystkie te różnice mogą prowadzić do drobnych różnic w kodzie maszynowym odpowiadającym algorytmom. Różnice między kolejnymi uruchomieniami mogą wynikać z działania różnych aplikacji i wielu innych czynników. Przeprowadzenie bardzo dużej liczby prób powinno zniwelować problem. Warto zauważyć, że małe różnice w wydajności algorytmów są współcześnie trudne do zauważenia. Jest to główna przyczyna tego, że koncentrujemy się na dużych różnicach!

ĆWICZENIA

2.1.1. Przedstaw (jako ślad działania kodu w stylu zastosowanym dla ALGORYTMU 2.1), jak przebiega porządkowanie tablicy `E A S Y Q U E S T I O N` przy sortowaniu przez wybieranie.

2.1.2. Jaka jest maksymalna liczba przestawień elementu w czasie sortowania przez wybieranie? Jaka jest średnia liczba przestawień elementu?

2.1.3. Podaj przykładową *N*-elementową tablicę, która prowadzi do maksymalnej liczby udanych testów `a[j] < a[min]` (co prowadzi do aktualizacji wartości `min`) w czasie sortowania przez wybieranie (ALGORYTM 2.1).

2.1.4. Przedstaw (jako ślad działania kodu w stylu zastosowanym dla ALGORYTMU 2.2), jak przebiega porządkowanie tablicy `E A S Y Q U E S T I O N` przy sortowaniu przez wstawianie.

2.1.5. Dla każdego z dwóch warunków z wewnętrznej pętli `for` sortowania przez wstawianie (ALGORYTM 2.2) opisz tablicę *N* elementów, dla której dany warunek jest zawsze fałszywy po zakończeniu działania pętli.

2.1.6. Która metoda, sortowanie przez wybieranie czy sortowanie przez wstawianie, działa szybciej dla tablicy, w której wszystkie klucze są takie same?

2.1.7. Która metoda, sortowanie przez wybieranie czy sortowanie przez wstawianie, działa szybciej dla tablicy, w której elementy mają kolejność odwrotną względem docelowej?

2.1.8. Załóżmy, że sortowanie przez wstawianie zastosowano dla losowo uporządkowanej tablicy, w której elementy przyjmują jedną z trzech wartości. Czy czas wykonania jest liniowy, kwadratowy, czy pośredni?

2.1.9. Przedstaw (jako ślad działania kodu w stylu zastosowanym dla ALGORYTMU 2.3), jak przebiega porządkowanie tablicy `E A S Y S H E L L S O R T Q U E S T I O N` przy sortowaniu Shella.

2.1.10. Dlaczego nie stosuje się sortowania przez wybieranie przy *h*-sortowaniu w sortowaniu Shella?

2.1.11. Zaimplementuj wersję sortowania Shella, która przechowuje ciąg odstępów w tablicy, zamiast go obliczać.

2.1.12. Zmodyfikuj sortowanie Shella tak, aby dla każdego odstępu wyświetlało liczbę porównań podzieloną przez rozmiar tablicy. Napisz klienta testowego do sprawdzania hipotezy, wedle której liczba ta jest niewielką stałą. Klient ma sortować tablice losowych wartości typu `Double`. Tablice mają mieć rozmiary będące potęgami 10 (zacznij od długości 100).

PROBLEMY DO ROZWIĄZANIA

2.1.13. *Sortowanie talii kart*. Wyjaśnij, jaką metodą uporządkowałbyś talię kart według kolorów (w kolejności piki, kiery, trefle, kara) i według wartości kart w ramach każdego koloru. Uwzględnij następujące warunki — karty są ułożone w rzędzie przednią częścią do dołu, a jedyne dozwolone operacje to sprawdzenie wartości dwóch kart i ich przestawienie (obróconych przednią częścią do dołu).

2.1.14. *Sortowanie struktury dequeue*. Wyjaśnij, jak posortowałbyś talię kart, jeśli jedyne dozwolone operacje to sprawdzanie wartości dwóch pierwszych kart, przedstawianie dwóch pierwszych kart i przenoszenie pierwszej karty na koniec talii.

2.1.15. *Kosztowne przestawienia*. Pracownik firmy spedycyjnej ma za zadanie zmienić uporządkowanie dużej liczby skrzyń według czasu ich wysyłki. Koszty porównań są tu więc bardzo niskie (wystarczy sprawdzić nalepki) w porównaniu z kosztem przestawień (trzeba przenieść skrzynie). Magazyn jest prawie pełny. Dostępne jest dodatkowe miejsce na tylko jedną skrzynię. Jaką metodę sortowania powinien zastosować pracownik?

2.1.16. *Sprawdzanie poprawności*. Napisz metodę `check()`, która wywołuje metodę `sort()` dla danej tablicy i zwraca `true`, jeśli metoda `sort()` sortuje tablicę *oraz* zachowuje w tablicy te same elementy, co początkowo. W przeciwnym razie `check()` ma zwracać `false`. Metoda `sort()` może przestawiać dane nie tylko za pomocą metody `exch()`. Możesz użyć metody `Arrays.sort()` i przyjąć, że działa poprawnie.

2.1.17. *Animacja*. Dodaj do klas `Insertion` i `Selection` kod, aby rysowały zawartość tablicy w formie pionowych słupków, tak jak na wizualnych śladach z tego podrozdziału. Kod ma wyświetlać słupki po każdym przebiegu, co prowadzi do powstania animacji kończącej się obrazem posortowanej tablicy, na którym słupki rozmieszczone są według wysokości. *Wskazówka*: użyj klienta podobnego do tego z tekstu, generującego losowe wartości typu `Double`, wstaw w odpowiednich miejscach wywołania `show()` w kodzie sortującym i zaimplementuj metodę `show()`, która czyści zawartość obrazu i rysuje słupki.

2.1.18. *Wizualny ślad*. Zmodyfikuj rozwiązanie poprzedniego ćwiczenia tak, aby klasy `Insertion` i `Selection` tworzyły wizualne ślady, takie jak te pokazane w tym podrozdziale. *Wskazówka*: przemyślane zastosowanie metody `setYscale()` pozwala łatwo rozwiązać problem. *Dodatkowe zadanie*: dodaj kod potrzebny do utworzenia czerwonych i szarych elementów, takich jak na rysunkach z podrozdziału.

2.1.19. *Najgorszy przypadek dla sortowania Shella*. Utwórz tablicę o 100 elementach, zawierającą wartości od 1 do 100, dla której sortowanie Shella z odstępami `1 4 13 40` wymaga możliwie dużej liczby porównań.

2.1.20. *Najlepszy przypadek dla sortowania Shella*. Jaki jest *najlepszy* przypadek dla sortowania Shella? Wyjaśnij odpowiedź.

PROBLEMY DO ROZWIĄZANIA *(ciąg dalszy)*

2.1.21. *Transakcje z możliwością porównywania.* Używając jako modelu kodu klasy `Date` (strona 259), rozwiń implementację klasy `Transaction` (ĆWICZENIE 1.2.13) o obsługę interfejsu `Comparable`, tak aby kolejność transakcji była wyznaczana przez ich wartość.

Rozwiązanie:

```
public class Transaction implements Comparable<Transaction>
{
   ...
   private final double amount;
   ...
   public int compareTo(Transaction that)
   {
      if (this.amount > that.amount) return +1;
      if (this.amount < that.amount) return -1;
      return 0;
   }
   ...
}
```

2.1.22. *Klient testowy do sortowania transakcji.* Napisz klasę `SortTransactions` zawierającą metodę statyczną `main()`, która wczytuje ciąg transakcji ze standardowego wejścia, sortuje je i wyświetla wynik w standardowym wyjściu (zobacz ĆWICZENIE 1.3.17).

Rozwiązanie:

```
public class SortTransactions
{
   public static Transaction[] readTransactions()
   {  // Zobacz ćwiczenie 1.3.17. }

   public static void main(String[] args)
   {
      Transaction[] transactions = readTransactions();
      Shell.sort(transactions);
      for (Transaction t : transactions)
         StdOut.println(t);
   }
}
```

EKSPERYMENTY

2.1.23. *Sortowanie talii*. Poproś kilku znajomych, aby posortowali talię kart (zobacz ĆWICZENIE 2.1.13). Obserwuj ich starannie i zapisz stosowane przez nich metody.

2.1.24. *Sortowanie przez wstawianie z wartownikiem*. Opracuj implementację sortowania przez wstawianie, w której nie występuje test `j>0` w pętli wewnętrznej. W tym celu najpierw umieść najmniejszy element na odpowiedniej pozycji. Użyj metody `SortCompare` do sprawdzenia skuteczności rozwiązania. *Uwaga*: technika ta często pozwala uniknąć sprawdzania wyjścia indeksu poza przedział. Element umożliwiający uniknięcie testu to *wartownik*.

2.1.25. *Sortowanie przez wstawianie bez przestawień*. Opracuj implementację sortowania przez wstawianie, w której większe elementy przenoszone są w prawo o jedną pozycję za pomocą jednego dostępu do tablicy na element (a nie przy użyciu metody `exch()`). Użyj programu `SortCompare` do oceny skuteczności rozwiązania.

2.1.26. *Typy proste*. Opracuj wersję sortowania przez wstawianie, która sortuje tablice wartości typu `int`. Porównaj wydajność tej wersji i implementacji podanej w tekście (która sortuje wartości typu `Integer` oraz niejawnie stosuje autoboxing i autounboxing do przekształcania danych).

2.1.27. *Sortowanie Shella ma złożoność poniżej kwadratowej*. Użyj programu `SortCompare` do porównania na swoim komputerze sortowania Shella z sortowaniem przez wstawianie i sortowaniem przez wybieranie. Użyj tablic o rozmiarach będących potęgami dwójki (zacznij od długości 128).

2.1.28. *Równe klucze*. Sformułuj i sprawdź hipotezę dotyczącą czasu wykonania sortowania przez wstawianie i sortowania przez wybieranie dla tablic, które zawierają tylko dwie wartości klucza. Załóż, że wystąpienie każdej z obu wartości jest równie prawdopodobne.

2.1.29. *Odstępy w sortowaniu Shella*. Przeprowadź eksperymenty, aby porównać ciąg odstępów z ALGORYTMU 2.3 z ciągiem 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289, 64769, 146305, 260609 (utworzonym przez złączenie ciągów $9\times4k - 9\times2k + 1$ i $4k - 3\times2k + 1$). Zobacz ĆWICZENIE 2.1.11.

2.1.30. *Odstępy geometryczne*. Przeprowadź eksperymenty, aby ustalić wartość t prowadzącą do najkrótszego czasu wykonania sortowania Shella dla losowych tablic dla ciągu odstępów $1, \lfloor t \rfloor, \lfloor t2 \rfloor, \lfloor t3 \rfloor, \lfloor t4 \rfloor$ i tak dalej dla $N = 10^6$. Podaj wartości t i ciągi odstępów dla trzech najlepszych znalezionych wartości.

EKSPERYMENTY *(ciąg dalszy)*

W dalszych ćwiczeniach opisano różne klienty pomocne w ocenie metod sortowania. Programy te mają być punktem wyjścia do zrozumienia cech związanych z wydajnością na podstawie losowych danych. We wszystkich programach użyj metody `time()` *(tak jak w programie* `SortCompare`*), co pozwala uzyskać dokładniejsze wyniki przez określenie większej liczby prób w drugim argumencie wiersza poleceń. Do ćwiczeń tych wracamy w dalszych podrozdziałach przy ocenianiu bardziej zaawansowanych metod.*

2.1.31. *Test podwajania*. Napisz klienta, który wykonuje test podwajania dla algorytmów sortowania. Zacznij od `N` równego `1000` i wyświetl `N`, prognozowaną liczbę sekund, rzeczywistą liczbę sekund i stosunek czasu dla podwojonych wartości `N`. Użyj tego programu do walidacji stwierdzenia, że sortowanie przez wstawianie i sortowanie przez wybieranie działają w czasie kwadratowym dla losowych danych wejściowych. Sformułuj i przetestuj hipotezę dla sortowania Shella.

2.1.32. *Wykresy czasów wykonania*. Napisz klienta, który używa biblioteki `StdDraw` do rysowania wykresów czasów wykonania algorytmu dla losowych danych wejściowych i różnych rozmiarów tablicy. Możesz dodać jeden lub dwa argumenty wiersza poleceń. Postaraj się zaprojektować przydatne narzędzie.

2.1.33. *Rozkład*. Napisz klienta, który wchodzi w nieskończoną pętlę i uruchamia metodę `sort()` dla tablic o rozmiarze podanym jako trzeci argument wiersza poleceń, mierzy czas każdego wykonania metody i używa biblioteki `StdDraw` do rysowania wykresu średnich czasów wykonania. Powinien powstać *rozkład* czasów wykonania.

2.1.34. *Przypadki skrajne*. Napisz klienta, który uruchamia metodę `sort()` dla trudnych lub „patologicznych” przypadków, które mogą wystąpić w praktycznych zastosowaniach. Oto kilka przykładów: już uporządkowane tablice, tablice o odwróconej kolejności, tablice, w których wszystkie klucze mają tę samą wartość, tablice składające się z tylko dwóch różnych wartości i tablice o wielkości 0 lub 1.

2.1.35. *Rozkłady nierównomierne*. Napisz klienta, który generuje dane testowe, losowo porządkując obiekty za pomocą rozkładów innych niż równomierny. Oto kilka takich rozkładów:

- Gaussa,
- Poissona,
- geometryczny,
- dyskretny (w ĆWICZENIU 2.1.28 opisano specjalny przypadek).

Opracuj i przetestuj hipotezę dotyczącą wpływu takich danych wejściowych na wydajność algorytmów opisanych w podrozdziale.

2.1.36. *Dane nierównomierne*. Napisz klienta generującego *dane* testowe, które nie są równomierne. Oto przykłady:

- jedna połowa danych to zera, a druga — jedynki;
- połowa danych to zera, połowa z reszty to jedynki, połowa pozostałych to dwójki i tak dalej;
- jedna połowa danych to zera, a druga — losowe wartości typu `int`.

Sformułuj i przetestuj hipotezy dotyczące wpływu takich danych wejściowych na wydajność algorytmów z tego podrozdziału.

2.1.37. *Częściowo posortowane*. Napisz klienta, który generuje częściowo posortowane tablice, takie jak:

- posortowana w 95% z losowymi wartościami w ostatnich 5%;
- z wszystkimi elementami znajdującymi się nie dalej niż 10 miejsc od ostatecznej lokalizacji;
- posortowana oprócz 5% elementów losowo rozrzuconych po tablicy.

Sformułuj i przetestuj hipotezę dotyczącą wpływu takich danych wejściowych na wydajność algorytmów opisanych w tym podrozdziale.

2.1.38. *Różne typy elementów*. Napisz klienta, który generuje tablice elementów różnych typów o losowych wartościach kluczy. Przykładowe typy mogą obejmować:

- klucz typu `String` (o przynajmniej 10 znakach) i jedną wartość typu `double`;
- klucz typu `double` i 10 wartości typu `String` (o przynajmniej 10 znakach);
- klucz typu `int` i jedną wartość typu `int[20]`.

Sformułuj i przetestuj hipotezę na temat wpływu takich danych wejściowych na wydajność algorytmów z tego podrozdziału.

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Nie trać czasu i energii
— korzystaj ze sprawdzonych rozwiązań!

Algorytmy od zawsze porównywane były do przepisów kucharskich. Z celnością tego porównania trudno dyskutować, na pewno jednak przesolenie zupy ma zupełnie inne konsekwencje niż błędnie opracowany lub zaimplementowany algorytm. To właśnie algorytmy decydują o czasie wykonania skomplikowanych operacji przez programy komputerowe, a ich odpowiednia implementacja niejednokrotnie decyduje o sukcesie lub porażce projektu wartego fortunę.

Dzięki tej książce masz szansę uniknąć typowych programistycznych błędów i porażek. Jej lektura zapozna Cię z najpopularniejszymi algorytmami, ich licznymi zaletami oraz słabymi stronami. Sprawdzisz, do czego można je zastosować, a w jakich miejscach lepiej zrezygnować z ich wykorzystania. Ponadto nauczysz się analizować działanie algorytmów, mierzyć ich wydajność oraz dobierać dane testowe. W książce zostały omówione klasyczne algorytmy sortowania, wyszukiwania, operacji na grafach oraz kompresji danych. Jej ogromnym atutem są przykładowe implementacje algorytmów w języku Java oraz to, że przedstawiony kod jest gotowy do natychmiastowego użycia! To obowiązkowa pozycja dla każdego programisty, któremu zależy na najwyższej wydajności tworzonych rozwiązań.

- Podstawowe pojęcia
- Struktura programu w języku Java
- Instrukcje, typy danych, wyrażenia w języku Java
- Korzystanie z abstrakcyjnych typów danych
- Analiza algorytmów
- Algorytmy sortowania i wyszukiwania
- Wykorzystanie grafów
- Znajdowanie najkrótszej ścieżki
- Operacja na łańcuchach znaków
- Algorytmy kompresji danych

Nr katalogowy: 7517

Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

Cena 149.00 zł
ISBN 978-83-246-3536-8
9 788324 635368